Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays

by

Jason Luu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays

Jason Luu
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2014

As technology scaling, human creativity, and other factors open new markets for FPGAs, the architectures of such chips must continue to evolve to meet changing demands. However, public domain software tools available to explore future FPGA architectures have not kept pace with advances in the field. Furthermore, these tools often have strong architectural assumptions embedded within the source code itself. Thus, short of major software rewriting, this limits the use of these tools to simple variations of particular architectures. In this thesis, we describe contributions to a large open-source collaborative project, called Verilog-to-Routing (VTR), that relaxes such limitations by providing an extensive software infrastructure for FPGA architecture exploration and CAD research. This infrastructure includes modern benchmarks, sample architecture description files, and a CAD flow that can target a broad space of architectures. We then describe new techniques in the packing stage of the CAD flow, which allow the packer to both target FPGAs with modern architectural features, as well as adjust computational effort based on architectural complexity. Finally, we conduct an architecture experiment on hard adders and carry chains to show the new capabilities of the software infrastructure and to quantitatively answer questions about the actual effectiveness of these classical architectural features.

Acknowledgements

There are several people and organizations that I would like to thank for their help and support throughout this dissertation.

I thank my advisors Jonathan Rose and Jason Anderson for their deep insights and advice on my thesis work and life in general. One of the things that I appreciate most is how Jonathan and Jason both sincerely care about the personal development of their students and the future of their students after graduation.

Much thanks is given to all those involved in our large collaborative vision to create a world class open source FPGA CAD tool, of which this dissertation forms a part. Of special mention include Vaughn Betz who acted as a third advisor, Kenneth B. Kent who stayed in this project since its inception 7 years ago, and Texas Instruments (notably Joyce Kwong and Jeff Rudolph) for a wonderful internship that spawned as a result of this project.

I express gratitude to my parents Can Luu and Huong Do for their encouragement throughout my university experience as well as to my wife, Jun Ye, for her support.

I would like to acknowledge Braiden Brousseau, Henry Wong, Daniel DiMatteo, Alex Rodionov, James Choi, Andrew Canis, Safeen Huda, Charles Chiasson, Kevin Murray, and many others in the lab. Their feedback and input were well appreciated.

Lastly, I thank NSERC, SRC, Texas Instruments, Altera, and the Rogers Scholarship for their financial support in my degree.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are integrated circuits whose functionality may be configured by the end user. They have grown from their humble beginnings as simple glue logic devices to complex systems targeting a diverse space of applications, each with distinct needs. Today, FPGAs may be found in internet routers which require massive data bandwidth, video/audio processing which require efficient on-chip memory and arithmetic functions, mobile devices which require low power operation, among many others. As technology scaling and human creativity continue to open up new markets for FPGAs, so too must the architecture of the FPGA itself evolve to meet these diverse demands.

Modern FPGA architecture design is usually done using an empirical approach. An architect evaluates hypothetical FPGA architectures by using CAD tools to map benchmark circuits to each architecture and then measures the resulting speed, area, power, and other quality metrics for each mapping. The knowledge gained from these experiments is then used to guide the design of the next set of hypothetical architectures. This process is repeated until a suitable candidate meets the objectives of the designer. The reliability of the data used to drive this process depends on having representative benchmarks, accurate architecture models, and software tools that can efficiently map circuits to the architectures being investigated. Thus, software infrastructure plays a critical role in FPGA architecture research.

Approximately 15 years ago, Betz *et. al.* released an open source software infrastructure for FPGA architecture exploration and CAD research [9]. This infrastructure included a curated set of technology-mapped benchmark circuits, some sample architecture description files, a packer (T-VPack), and a place-and-route tool (VPR). Betz's infrastructure provided great value to the research community serving as the foundation of much subsequent academic research [77] [11] [48] [70] [17] . However, in academia, new software written by one group that advances the field often does not cross institutional boundaries for use by another group; thus, over time, the open source software infrastructure became increasingly out of step with modern architectures. Consequently, the software development effort now required to target state-of-the-art architectures is very large.

With limited development resources, academic researchers are stuck targeting simple architectures for simple glue logic circuits even though FPGAs today can be two to four orders of magnitude larger, contain specialized logic such as embedded block memories, have far more complex soft logic, among other important differences. Unlike academia, the major FPGA vendors have more development resources which allows them to create their own internal software infrastructure capable of targeting large, complex architectures. However, those tools are closed source which limits their use in the academic community.

Hence, there is a need for a new open source baseline for architecture exploration. This open source software should be capable of targeting a broad range of different architectures so as to reduce, if not eliminate, code modifications normally required to investigate modern and future architectures.

This PhD dissertation makes major advances towards providing an open source, flexible, software infrastructure for FPGA architecture exploration, called Verilog-to-Routing (VTR), which is based on Betz's original VPR infrastructure. This infrastructure consists of larger benchmark circuits from real applications, CAD tools that are capable of mapping to a diverse range of architectures, sample architecture descriptions, scripts that simplify the use of the full toolchain, and finally developer support such as documentation, regression tests, and bug tracking. In addition to its value as an architecture exploration platform, this large-scale infrastructure creates opportunities for CAD research. One reason for this is that new architectural features provide unique challenges to CAD algorithms. A more subtle, yet equally important reason is the complex interactions within a project of this scale, for algorithmic optimizations in one part of the system cause profound, perhaps unexpected, effects in other parts. Thus, the first key contribution of this dissertation is an infrastructure for enabling future CAD and architecture research [80] [59].

Within the large space of CAD and architecture for FPGAs, our work focuses on CAD for FPGA *logic blocks*. Logic blocks are responsible for computation and storage. Examples of logic blocks commonly found in commercial FPGAs include configurable memories for on-chip storage, configurable bus-based arithmetic blocks, and soft logic blocks for general purpose computation [97] [6]. When designing a logic block, one needs to choose the logic, hierarchy, interconnect, and modes of operation available within the block. Designing a well-optimized logic block is difficult because many decisions in the vast design space are coupled – some choices synergize, while others conflict. The diverse set of interdependant axes along which an architecture may be explored gives rise to a rich and complex space of interesting candidate architectures. These different exploration points require CAD tools that can efficiently map to them. In a typical FPGA CAD flow, the packing stage is responsible for mapping a flattened, technology-mapped representation of a circuit to the physical logic blocks of the FPGA. Historically, packing algorithms embed strong architectural assumptions about the logic blocks into the software itself. Though this may provide good runtime and quality of results for the architectures being targeted, it also means that targeting new architectures often requires substantial software development effort. If a packer is capable of mapping to any logic block without software changes, then that would enable a more extensive exploration of the design space.

Our first attempt at generalizing the packer was done as part of my Master's work [56]. There, we showed that the general packing problem creates a local intra-logic block placement and routing problem that is challenging to solve. The packing solution developed as part of that work served as a proof-of-concept but suffered from practical issues including long runtimes, inconsistent quality, and a lack of timing-driven capability. In this PhD dissertation, we revisit the general packing problem with better success. Though we fell short of creating a packer that will pack to any architecture, we successfully created a packer that can target a very wide space of logic block architectures [57] [62] [80]. This forms the second contribution of this dissertation.

We observe that common interconnect structures in logic blocks, such as crossbars, carry chains, dedicated signals, and others, have unique properties that simplify the local intra-logic block placement and routing problem encountered by a general packer. Naively using a general-purpose routing algorithm, such as a negotiated congestion router, to route these structures is often unnecessarily time consuming.

Instead, when these structures are encountered, the packer should automatically reduce its computational effort because the routing of these structures can be done with much less effort. To accomplish this, we employ three techniques in this work: *speculative packing*, *pre-packing*, and *interconnect-aware pin counting*. We show that these techniques result in improvements to runtime and quality of results across a spectrum of architectures, while simultaneously expanding the scope of architectures that can be explored. Techniques for interconnect-adaptive packing forms the third major contribution of this dissertation [62].

Finally, the fourth contribution of this dissertation is an architectural study that uses the new CAD capabilities mentioned earlier to investigate hardened addition and subtraction in FPGAs. Hard adders, along with carry chains (the dedicated interconnect for carry logic that join hard adders together), are widely used in commercial FPGAs to improve the efficiency of arithmetic functions. There are many design choices and complexities associated with such hardening including circuit design, FPGA architectural choices, and CAD decisions. There have been few published studies, however, on these choices despite a general consensus in the FPGA community of their importance, and hence we explore a number of possibilities for hard adder design. We show that hard adders and carry chains, when used for simple adders, increase performance by a factor of four or more, but on larger benchmark designs that contain arithmetic, improve overall performance by roughly 20%. We measure an average area increase of 5% for architectures with carry chains but believe that better logic synthesis should reduce this penalty. Interestingly, these performance ratios stay fairly close across many different hard adder and carry chain designs [61].

This thesis is organized into the following chapters: The background chapter briefly describes FPGA architecture and the general FPGA CAD flow before going into detail on the packing stage of the CAD flow. The third chapter describes the VTR architecture exploration infrastructure. The fourth chapter formally defines the packing problem and presents our overall approach to solving this problem. The chapter following describes the interconnect-aware advances made to the packer. The sixth chapter presents an architectural study of hard adders and carry chains. The final chapter presents conclusions and avenues for future work.

Chapter 2

Background

This chapter provides the relevant background material necessary for understanding this thesis. This thesis discusses advances made to the VTR software infrastructure for exploring FPGA logic block architectures. Thus, we begin with material on FPGA architecture, focusing on logic blocks. Afterwards, we present background on the infrastructure used to conduct architecture exploration. Finally, we describe in detail one crucial part of that infrastructure, called *packing*, that comprises a major focus of this thesis.

2.1 FPGA Architecture and Logic Blocks

An FPGA architecture is composed of programmable logic blocks, I/Os, and interconnect. Logic blocks implement functions and storage, I/O interfaces the FPGA with the extra-chip world, and interconnect connects signals to and from the logic blocks and I/Os. This thesis focuses on FPGA logic block exploration so this section describes logic block architecture in greater detail.

FPGA logic blocks play a crucial role in the performance, cost, and energy efficiency of an FPGA. This section describes the different types of logic blocks, their key features, and their intended functionality. We also highlight the key challenges that these logic blocks pose to the CAD software.

2.1.1 Soft Logic Blocks

Soft logic blocks are designed to both implement general boolean functions and provide local data storage. Figure 2.1 shows an example of a basic FPGA soft logic block architecture with some typical parameters [9]. This logic block contains a group of *basic logic elements* (BLEs) where each BLE consists of a *look-up table* (LUT) with an optionally registered output. A full crossbar provides local connectivity, joining any logic block input pin and any BLE output to any BLE input.

There has been some prior work optimizing the various parameters of a basic soft logic block. Betz [9] and Ahmed [2] investigated how many input pins a basic soft logic block should have. Both studies concluded that the number of input pins should be just over half the total number of BLE inputs. Ahmed [2], and later Kuon [43], showed that there is a large range of reasonable values for the number of BLEs in a logic block. That number can vary from 4 to 12 while still providing similar overall quality. Ahmed and Kuon also showed that 6-LUT architectures result in better overall performance, 4-LUT architectures result in a better overall area-delay product, and 5-LUTs show characteristics that are in

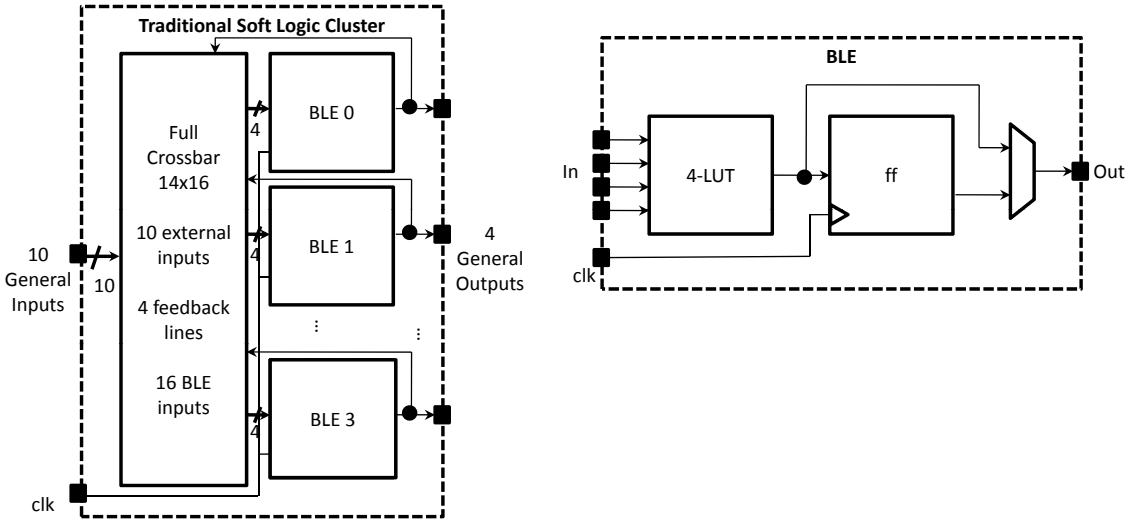

Figure 2.1: A basic soft logic block with typical parameters.

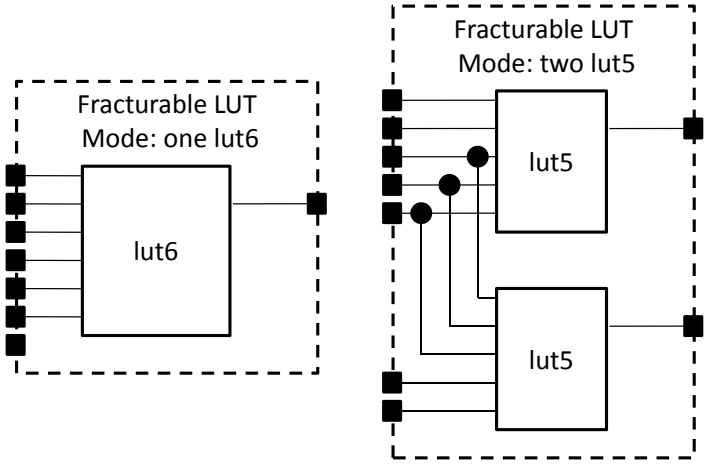

Figure 2.2: A fracturable LUT that can operate as one 6-LUT or as two 5-LUTs that share 3 inputs.

between. However, the quality of results that can be achieved from simple parameter sweeps alone is limiting. To obtain further gain, more advanced architecture optimizations must be attempted.

Fracturable LUTs

One technique to obtain the delay advantage of larger LUTs, while retaining the area efficiency of smaller LUTs is to use *fracturable LUTs*. A fracturable LUT is a LUT that can optionally implement two smaller LUTs with some shared inputs. If an architecture contains fracturable LUTs, and if the large physical LUT cannot be entirely filled, it may be possible to pair another small LUT from the user circuit into the partially filled large LUT. If this pairing can be done often enough, and if the area overhead to create a fracturable LUT is small, then fracturable LUTs result in an area savings while maintaining the delay performance of large LUTs.

Figure 2.2 illustrates a fracturable LUT that can operate in 6-LUT mode or in dual 5-LUT mode, where the 5-LUTs share three input pins. The total number of input pins used in dual LUT mode is

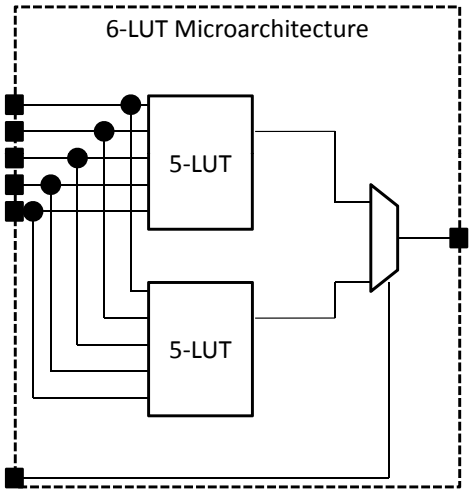

Figure 2.3: The microarchitecture of a 6-LUT consists of two 5-LUTs.

an important parameter that we define as *FI*. In this example, FI can range from a minimum of 5 to a maximum of 10 (in the figure, it is 7). A smaller value of FI constrains which LUTs may go together, while a larger FI provides more flexibility at the cost of more routing interconnect. An FI of 10 means that the two 5-LUTs may operate independently. The vendors have not come to consensus on what is the right value for FI. The fracturable LUTs in a Xilinx Virtex 6 FPGA have an FI of 5, while the fracturable LUTs in an Altera Stratix V FPGA have an FI of 8.

The fracturable LUT itself can be implemented with little area overhead because the microarchitecture of a LUT, shown in Figure 2.3, already contains within it two smaller LUTs. Therefore enabling a large LUT to optionally fracture into two smaller LUTs (with shared inputs) may be provided with low overhead. This notion of fracturing a larger LUT to multiple smaller LUTs can be generalized to a hierarchy of smaller LUTs that may compose together to form larger LUTs. However, a high degree of fracturing has drawbacks for increasing output pins also increases the demand on interconnect.

The Xilinx 2000 series FPGA [33] was one of the first FPGA architectures to have fracturable LUTs. However, at that time, LUTs in FPGAs were smaller than today, thus reducing the benefit of using fracturable LUTs. As LUT sizes in FPGAs trended larger, fracturable LUTs saw a resurgence in commercial devices and have now become commonplace [7] [34] [49]. Despite this, most literature on FPGA architecture today do not consider fracturable LUTs due to difficulties in software modelling.

Depopulated Crossbars

The full crossbar shown in Figure 2.1 for a basic logic block consumes significant area (roughly 50% of logic block area, depending on the architecture). One may reduce the area of a logic block by carefully removing some switch points to create what is known as a *depopulated crossbar*, but doing so may have global side-effects. A depopulated crossbar may increase the number of logic blocks needed to implement a circuit because some previously routable clusters of BLEs become unroutable in the new, less flexible interconnect. Furthermore, inter-logic block interconnect area might increase because of reduced routing flexibility from having less switch points. These global side-effects can be hard to quantify analytically so CAD tool support is essential for a proper study.

Lemieux [48] showed several different depopulated crossbar designs that provide an overall area savings. To conduct his study, Lemieux modified public domain software [9] to support these interconnect structures. But these important updates did not get integrated into the public infrastructure, resulting in studies afterwards neglecting depopulated crossbars.

2.1.2 Hard Logic Blocks

One of the central questions in FPGA architecture is determining which functions to harden and which to implement in soft logic [79]. Generally, a function makes a good candidate for hardening if it appears often in the set of used applications, and if there is a large advantage when that function is implemented in hard logic rather than soft. But this decision of hard versus soft is more nuanced than a simple, boolean, choice. Architecting some flexibility to a hardened function, such as by providing a programmable size versus quantity trade-off, may enable much better utilization with relatively low overhead. Logic blocks that are centred around hardened, dedicated, functions are called *hard logic blocks*.

Configurable Memories

I/O pins of ICs do not scale as quickly with improved process technology as on-die transistors. Consequently, applications today have limited off-chip data bandwidth compared to on-chip bandwidth. This drives the need for on-chip storage of local data. Although registers in soft logic blocks suffice as storage for some simple tasks, when an application requires buffering or larger caches, then block memories with significant amounts of storage, become preferable. Wong [95] makes a strong case for hardened memories in FPGAs by showing that hardened memories use two orders of magnitude less area than their soft counterparts. However, selecting the right quantity, size, depth, and width for hardened memories is not straightforward because these parameters vary based on the application.

Hardened memories are found today in commercial architectures [97] [6]. Though the major vendors are not in agreement on what is the best combination of memories to provide, they are in agreement that hardened memories should have configurable *aspect ratios* to accomodate applications with different memory aspect ratios. Since there is both a demand for, and a supply of, hardened memories in FPGAs, large modern designs often will contain one or more of these blocks. This makes CAD tool support for hardened memories essential, for a CAD flow that does not support these blocks is confined to targeting small, glue logic style, circuits.

Wilton [94] did an extensive study on various configurable memory architectures for FPGAs and developed CAD algorithms to map to these memories. Ho [30] followed up on this work with better algorithms to map memories in the user circuit to physical dual-port memories. However, their code did not integrate into the public infrastructure for FPGA architecture exploration, so many architecture studies afterwards did not consider hardened memories.

Bus-Based Arithmetic

Bus-based arithmetic operations, such as wide multiplication, are common especially in fields that require digital filters, like video and audio processing. Their implementation in soft logic is relatively much more expensive than a hard logic implementation making them a good candidate for hardening. However, selecting which operations, as well as the appropriate widths of each operation, to harden is difficult since requirements vary across applications. Thus, bus-based arithmetic hard logic blocks commonly

offer different modes of operation as well as some limited routing flexibility to improve utilization. These configurable arithmetic blocks appeared in FPGAs in 2002 [50] and have continued to evolve since then. We examine two examples of such blocks: an integer DSP logic block and a floating-point logic block.

Figure 2.4 [8] shows an example of one mode of a bus-based arithmetic block from a commercial FPGA. This is a variable precision DSP block from the Altera Stratix V device family operating in 18x18 bit mode. Here, one can see bus-based hardened multipliers, adders, and registers with optional bypass multiplexers. This DSP block has another mode that offers wider 27x27 bit operation, but with less computation units, to accomodate wider arithmetic functions. Software that aims to target future bus-based arithmetic logic blocks should, at minimum, support heterogeneous functions, different modes of operation, and unique routing flexibilities.

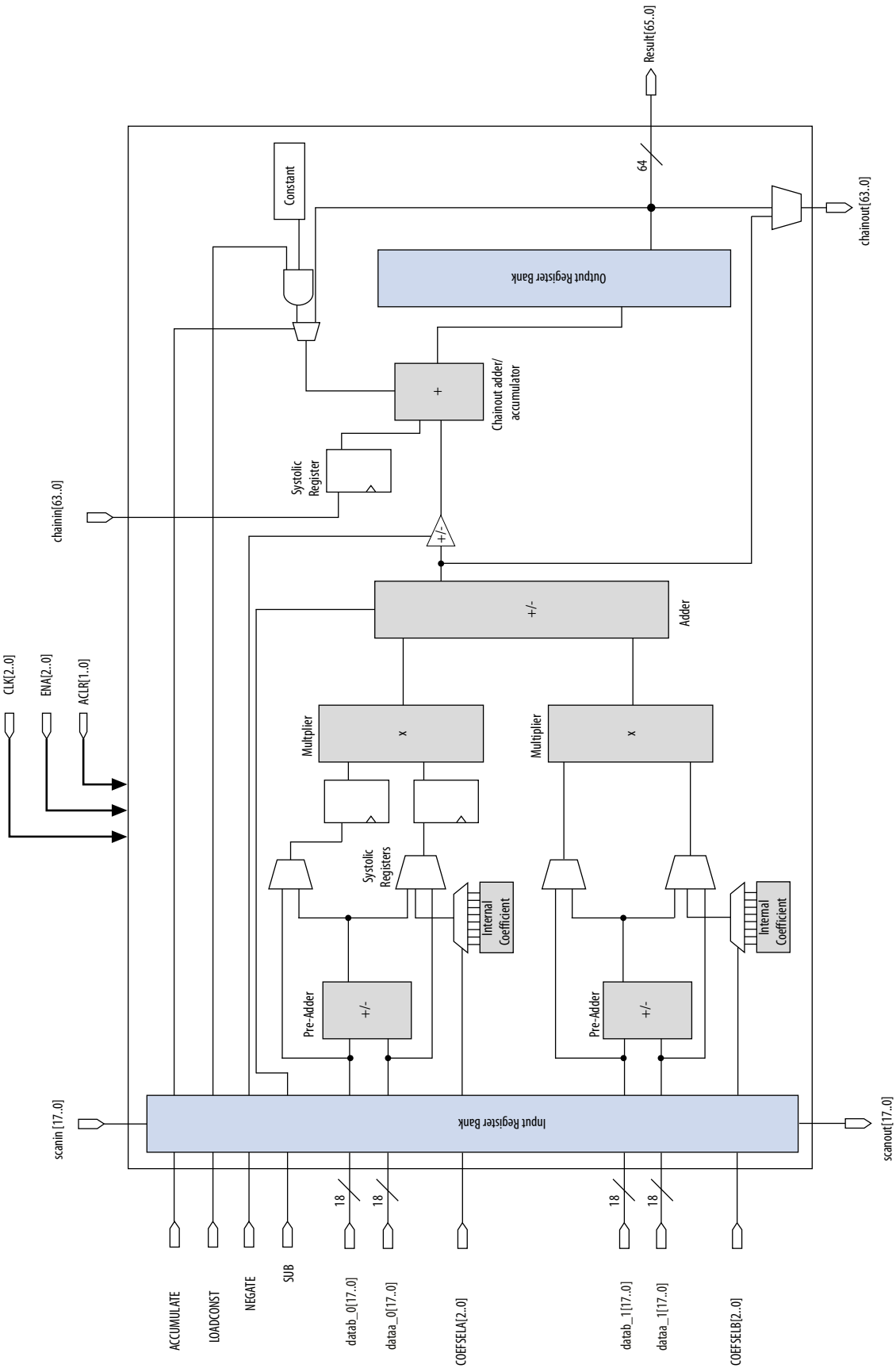

Figure 2.4: The 18x18 multiplier mode of the Stratix V variable precision DSP block [8].

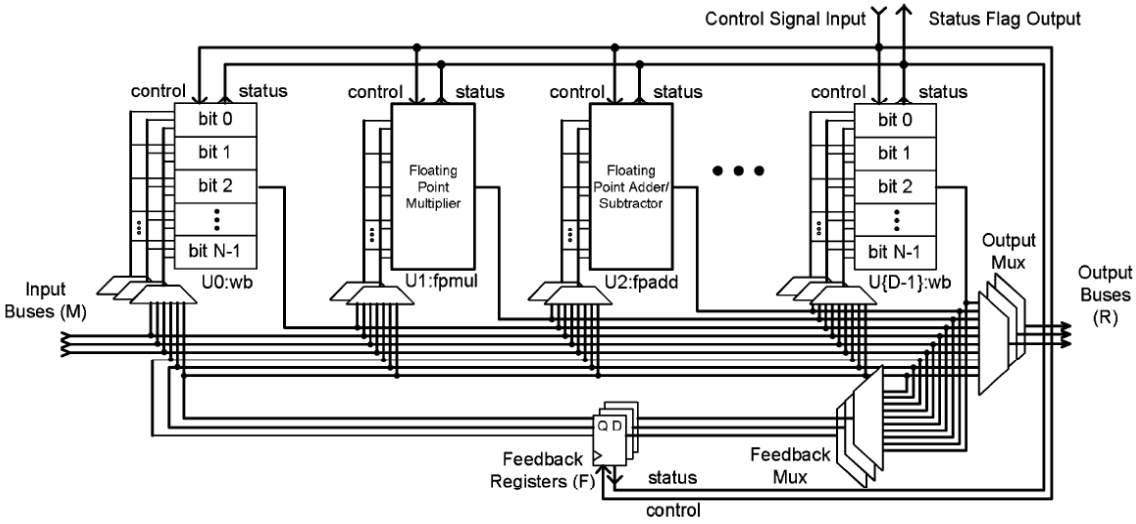

Figure 2.5: A floating-point hard logic block architecture by Ho [15].

Ho [15] proposed a hard logic block specialized for floating-point computation, as shown in Figure 2.5. This logic block contains bus-based registers, floating-point multipliers, and floating-point adders. Any computation or storage unit can drive any unit on its right. A small number of feedback lines limits how many units can drive a unit on its left. The authors did not have access to software that can map user logic to this logic block so they performed the mapping manually. The development effort required for manual mapping imposes a large barrier to entry in the research of new bus-based arithmetic logic blocks, a barrier that we would like to avoid with flexible software tools.

As emerging user applications place new demands on FPGAs, we foresee a greater diversity in future bus-based arithmetic blocks. Software tools will continue to play a crucial role in the design of these blocks.

2.1.3 Hardened Adders and Carry Chains

Architecting FPGAs to efficiently implement addition and subtraction poses an interesting architectural challenge. These operations are common and wide operations are especially slow. Thus, addition and subtraction are great candidate functions to harden. But, unlike memories and multipliers, the pin-to-area ratio for a hardened adder is quite large which makes the interconnect overhead for bringing signals to and from the adder relatively expensive. Thus, adders are typically not made into their own hard logic block. Rather, they are included in existing logic blocks to reuse the existing interconnect of those logic blocks.

Hard adders are found in both soft and hard logic blocks of commercial FPGAs. In bus-based arithmetic blocks, hard adders may optionally follow hard multipliers (to realize multiply and accumulate) and face the same considerations as discussed earlier. In soft logic blocks, hard adders provide a wealth of interesting design choices. One needs to decide how the adder interacts with the LUT, what granularity of adders to provide, among other choices. Commercial FPGAs employ dedicated routing paths, called *carry chains*, to join the carry signals of adjacent adders together to implement wide functions [31]. Figure 2.6 shows a simple soft logic block with hard adders that are joined together by a dedicated,

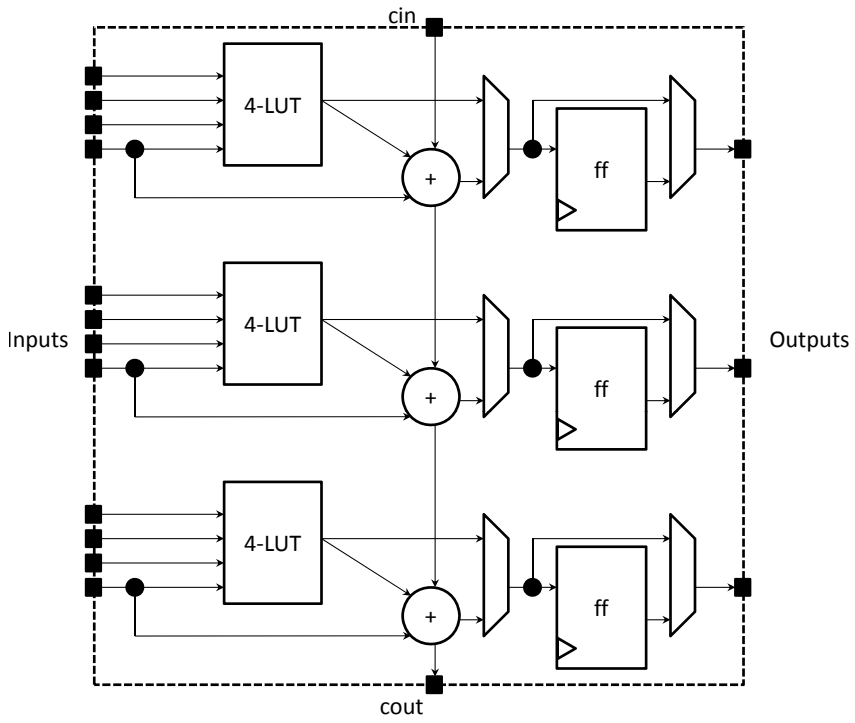

Figure 2.6: A simple ripple carry chain in a soft logic block.

inflexible, carry chain. There is a general belief in the field that both hard adders and carry chains are crucial for performance in FPGAs but there is little published literature measuring their usefulness in real applications.

Previous work in this area began in the early 90's, when Hsieh et al. [31] described the Xilinx 4000 FPGA that had soft logic blocks that were capable of implementing two independent adder bits per block. They employed dedicated carry logic and routing from adjacent logic blocks for the carry signals. Woo [96] proposed adding additional flexibility to the fast carry links between logic blocks to enable flexible tree-based mappings of addition/subtraction/comparison functions. FPGAs today have evolved substantially from those investigated by Hseih and Woo. Modern FPGAs tend to have larger, and more numerous, lookup tables in a soft logic block and also tend towards having less flexbility in the carry chain.

Since Hsieh et al. and Woo, there has been a few interesting works in this area. Xing proposed implementing carry-lookahead adders (in an FPGA architecture that contains just ripple adders) by using soft logic to do the carry-lookahead operation [99]. His case study on the Xilinx 4000 series FPGAs show that this approach is limiting because of the large area and delay penalty that results when soft logic is involved in carry-lookahead computations. Hauck et al. [28] evaluated different implementations for FPGA adders including ripple carry, carry-skip, and tree-based adders. He showed that a Brent-Kung adder achieves 3.8 times speedup vs. the basic ripple carry adder for 32-bit addition at the expense of between 5 to 9.5 times more area for the adder. Parandeh-Afshar et al. [75] proposed adding hardened compressors to soft logic blocks to speed up multi-input addition with a focus on DSP and video applications. The benchmarks used in that study appear to be on the order of a few hundred 6-LUTs [76]. Ultimately, more diverse and larger benchmarks need to be used to better quantify the

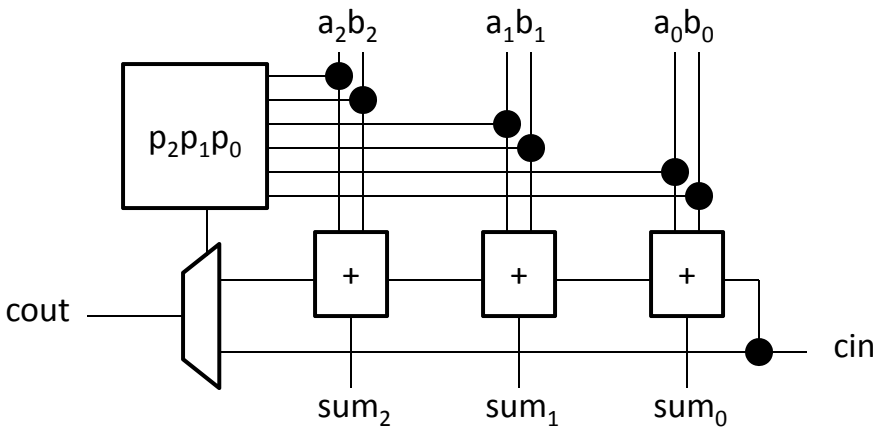

Figure 2.7: A 3-bit carry-skip adder.

effects of hard adders and carry chains.

Different FPGA vendors currently choose a variety of hard arithmetic architectures inside their soft logic blocks. The Xilinx Virtex-7 FPGA family [98] contains a basic ripple carry adder architecture where addition can only start on every 4^{th} adder bit. The interaction between the soft logic and the adder is flexible; the adder can either be driven by a 6-LUT and a logic block input pin or be driven by two 5-LUTs with shared inputs. The Altera Stratix V architecture contains carry-skip adders. An example of a 3-bit carry-skip adder is shown in Figure 2.7. A carry-skip adder contains a standard ripple-carry adder plus additional lookahead circuitry that determines whether to use or to skip the ripple-carry adder when generating the, assumed to be timing critical, carry out signal. The Stratix V architecture employs a two-level carry-skip adder architecture [51]. Each soft logic block contains ten 2-bit carry-skip adders that can be cascaded with dedicated links. Between two logic blocks, there is an additonal carry-skip stage that can skip 20 bits of addition. Lewis et al. [51] claims that this adder results in both a delay improvement and an area reduction compared to the basic ripple carry adder, as the increase in logic gates necessary for the carry-skip feature is more than offset by the area reduction made possible via optimizations during transistor design of the FPGA itself. Each of the two 5-LUTs in the fracturable LUT in Stratix V drives two bits of arithmetic. Each adder input is driven by a 4-LUT with input sharing constraints [7] as shown in Figure 2.8. Outside of microbenchmarks, neither vendor has published, in depth, the impact of the major design decisions for their hard adder and carry chain architectures.

Prior published work on hardened arithmetic focused on the implementation of arithmetic structures, and evaluated results on microbenchmarks such as adders and adder trees or very small designs. A full design, on the other hand, imposes many other demands on the FPGA and its CAD flow. In this thesis, we seek to measure the impact of different hard adder choices not only on microbenchmarks, but also on complete designs with a full CAD flow.

2.2 FPGA Architecture Exploration Infrastructure

New FPGA architectures are explored using an empirical approach. First, benchmark circuits are mapped onto a prospective architecture using a CAD flow; then, quality metrics from the implementation

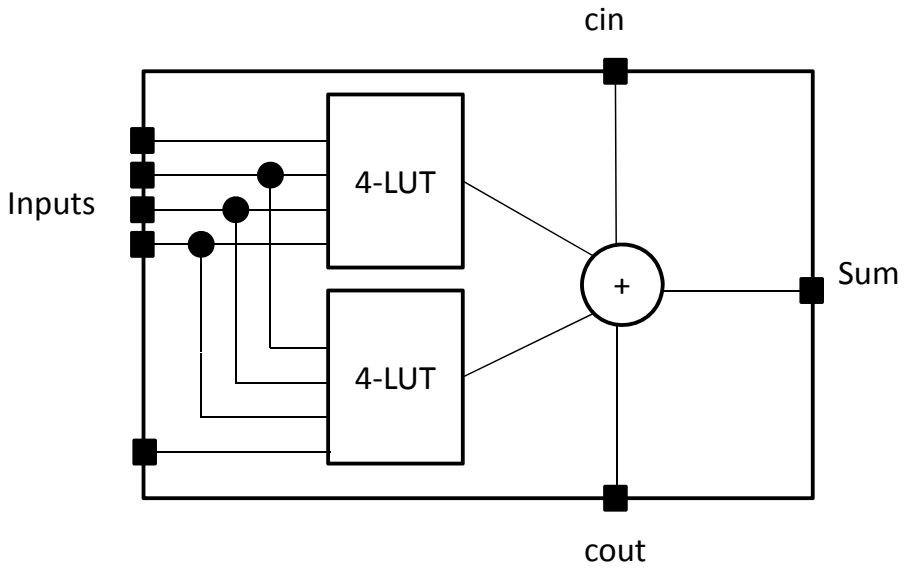

Figure 2.8: In arithmetic mode, a single adder bit in a Stratix V soft logic block is driven by two 4-LUTs that share inputs.

of each benchmark are obtained and compared with previous results. The knowledge gained from these experiments helps guide the creation of the next new architecture. This section provides background on the different parts of this exploration infrastructure starting first with benchmarks and then following with the FPGA CAD flow.

2.2.1 Benchmarks

As is the nature of any scientific study, the inputs used in an experiment can have a major impact on the resulting conclusions. Being a major input to an architecture experiment, one therefore wants high quality benchmarks that capture the space of applications of interest. Furthermore, to allow comparisons between different studies, a researcher would want to employ a public set of standard benchmarks before considering closed-source circuits.

Although there exists a large set of public benchmarks that may be used for FPGA architecture exploration [73], many of these benchmarks are specialized for particular domains (such as DSP or floating-point computations) so only a few benchmark sets are used for general study. Among the general benchmark set, the MCNC benchmarks [100] are arguably one of the most commonly used benchmarks in FPGA CAD and architecture studies. These benchmarks are convenient for they cover diverse functions and are compatible with open source CAD tools. But the MCNC benchmarks are more than two decades old, so the application space targeted by them is only a narrow slice of the applications that use FPGAs today. The newer IWLS 2005 benchmarks [5], originally meant for the related field of *ASIC* CAD, represent a much broader space of applications and have been used in FPGA CAD studies [10]. However, compatibility issues with open source tools limits the representation of these benchmarks entirely to soft logic which may not be realistic especially for large designs where one would expect to see, at the very least, hard block memories. This points to the need for modern public benchmarks that are compatible with open source tools. As we will show, our work has both created one such benchmark

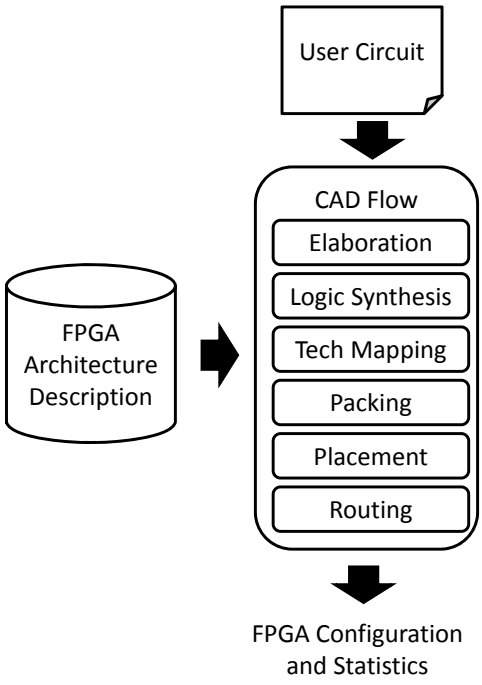

Figure 2.9: A generic FPGA CAD flow.

set [80] [59] as well as enabled others [69].

It is generally believed that the closed source benchmarks found in industry publications [52] are high quality, state-of-the-art, FPGA designs that are superior to what is available publicly today. But since these circuits are not available for public use, their use is necessarily limited in academic works. Altera has released some of their previously closed source benchmarks as part of QUIP [63]. There is currently an on-going effort to make these QUIP benchmarks, and other benchmarks, that target large industrial FPGAs to also be compatible with an open-source flow [69].

2.2.2 FPGA CAD Flow

A major challenge in FPGA architecture exploration work is changing the FPGA CAD flow to make use of the new features of the architecture being investigated. This section describes the FPGA CAD flow and discusses the prior work in this area.

An FPGA flow takes as input a user circuit, specified in a hardware description language (HDL) such as Verilog or VHDL, and a description of the FPGA architecture being targeted. The outputs of the flow are the configuration needed for the FPGA to implement the user circuit and statistics about that implementation.

Figure 2.9 show the major stages in a generic FPGA CAD flow. Elaboration takes as input a user circuit, described in HDL, then resolves all instantiations and parameters in the HDL to produce a flattened netlist representation of the user circuit. Logic synthesis performs technology independent optimizations on this flattened netlist. Technology mapping translates an optimized netlist into *atoms*. Atoms are specific, basic, components found in the FPGA architecture such as LUTs and flip-flops. The packing stage then groups the atomic netlist from technology mapping into the logic blocks available in

the architecture. Placement finds physical, geographic locations for each logic block. Routing determines the physical wires and switches needed to implement the connections of the user circuit. After routing, the information needed to configure the FPGA to implement the user circuit is complete. At this point, analysis can be done to figure out various statistics such as the area, delay, and power of the implementation.

The exploration of new logic elements often requires substantial support in some, if not all, stages of this CAD flow. To illustrate what is involved, we will use the configurable floating-point logic block described in Figure 2.5 to serve as an example. Given an architecture that contains these blocks, elaboration must decide whether a particular function in the user circuit should map to soft logic, use one of the hardened floating-point atoms within the floating-point logic block, or some combination of the two (which may be necessary when the user function is larger than the hardened floating-point functions supplied). Logic synthesis and technology mapping should understand the functionality and delay profile of the hardened floating point functions to better optimize the netlist produced by elaboration. Packing must determine where hardened floating-point atoms map to within the block. However, due to limited internal connectivity, this can be quite challenging. In this architecture, the floating-point logic block consumes far more area than a soft logic block, so the number of floating-point logic blocks is relatively scarce. The limited number of locations for these floating-point logic blocks in turn constrains optimizations during placement. Routing is unaffected in this particular example. As this example illustrates, the development effort needed to create a CAD flow that can explore new logic blocks can therefore be quite high. Thus, the capabilities of public domain FPGA CAD tools are often one of the limiting factors in the kinds of exploration a researcher is willing to conduct.

2.2.3 Publicly Available FPGA CAD Tools

An FPGA CAD flow requires substantial development effort to create from the ground up. Instead, researchers often start with existing CAD software then modify that software for their specific research. This section describes existing available FPGA CAD software. We discuss open-source CAD flows as well as publicly available, but closed-source, CAD flows.

For about a decade, many CAD and architecture researchers formed their own CAD flow using open source tools. Often, a logic synthesis tool, such as SIS [83], is used in conjunction with a physical design tool, such as T-VPack and VPR 4.30 [9]. This infrastructure, though very useful for its time, only supported architectures that contain LUTs and flip-flops which limits the space of circuits that can be investigated.

As part of my Master's work, we formed a collaborative team to build upon this initial public infrastructure adding Odin [37] for Verilog elaboration, replacing SIS with ABC [66] for logic synthesis and technology mapping, modifying T-VPack [60] for packing, and significantly improving VPR [60] for place-and-route. This work was publicly released under the name Verilog-to-Routing (VTR). We labeled the initial release as VTR 5.0. The value of this release included more optimized routing architectures, larger benchmarks, and some rudimentary support for hardened logic throughout the CAD flow. However, VTR 5.0 treated hard logic blocks as non-configurable, monolithic, objects that were either used or not used. This is, as we showed earlier, undesirable because inflexibile configurations of hard logic limit their use. In order to target flexible hard logic block architectures, the packing stage, which is responsible for mapping to logic blocks, needed major improvements. These improvements form a major contribution of this thesis.

Other groups have also worked on improving the public FPGA CAD infrastructure. Groups from four universities, headed by Steiner, released an open source FPGA CAD and architecture exploration software infrastructure called Torc [88]. Torc supports a wide variety of different netlist formats which in turn provides a convenient way to swap different tools, both academic and commercial, for the various stages of the CAD flow. Unlike VTR, which terminates after routing, Torc has a bitstream generation stage post-routing where the bitstreams can be used to program Xilinx FPGAs. The extensive development that the Torc team invested to support different netlists and to support the programming of commercial FPGAs meant that less resources were available to invest in the different CAD stages. As a result, Torc relies on other tools (such as VTR or commercial tools) for stages that are not available or not handled well by the Torc internal implementations. Currently, they appear to be missing their own technology mapper and packer.

RapidSmith is an open-source FPGA CAD infrastructure that enables open source CAD tools to directly target commercial Xilinx FPGAs. While the RapidSmith project is very helpful in enabling new CAD research, due to its focus on support for existing FPGAs, it is of limited use for exploring future hypothetical FPGAs.

Several open source tools exist that help automate the the creation of the FPGA architecture itself. Kuon [43] and Chiasson [18] created tools that optimize the electrical design of the FPGA. Smith [86] employs geometric programming techniques to simulateneously optimize lower level transistor sizing as well as higher level architectural parameters (such as LUT size and cluster size). As FPGA architectures become increasingly more complex, these kinds of tools will become increasingly more important. The output of these tools serve as inputs to the FPGA CAD flow.

Closed-source, but publicly available, CAD flows can be very useful for research. The two FPGA vendors, Altera and Xilinx, provide support for FPGA CAD researchers by releasing interfaces for third party software to link with their commercial CAD flows. QUIP is a package from Altera that gives researchers the ability to interface third party tools into various stages of the Quartus II CAD flow [63]. Xilinx provides descriptions for different parts of the architecture and different representations of the netlist to those who ask, so that third party flows can map user circuits to a Xilinx FPGA [88] [46]. The value of this is large. Using either vendors' flows, one has access to commercial front-end synthesis which makes available a far larger space of digital circuits than academic flows. Furthermore, the final analysis, such as timing and power, are accurate, taking into account many real world effects. However, vendor flows are designed to target logic found in vendor devices and are thus not well suited for exploring architectures that contain different logic. Also, since the main tools are closed source, optimizing the algorithms within the different CAD stages is near impossible. For these kinds of research, one still requires the flexibility offered by open source CAD tools.

2.3 Packing

The packing stage shown in Figure 2.9 forms a flat, technology-mapped netlist into a netlist of logic blocks found in the FPGA architecture. A core contribution of this dissertation is expanding the capabilities of the packing stage of a public CAD flow to enable the exploration of a much greater space of logic blocks. In this section, we review prior work in this field, starting with simple packers and then examine progressively more advanced packers.

2.3.1 Traditional Packing

Much of the prior work in packing is limited to only targeting basic logic blocks, such as the one shown in Figure 2.1, that consist of LUTs and flip-flops joined by full crossbar interconnect. However, modern designs often contain hard logic blocks, such as configurable memories and multipliers, because these specialized blocks are supplied in commercial FPGAs and because these blocks provide very strong area efficiency when used [95]. Thus, the inability to target hard logic blocks restricts these packing algorithms to older, small, glue logic, circuits. Despite this, a review of these algorithms is insightful because the issues they deal with are also found, in a more challenging form, in more general packing.

Marquardt, Betz, and Rose did important early work in the field of FPGA packing [64]. They released an open source, timing-driven, packer called T-VPack that targets basic logic blocks. Much prior work compares to this tool and a large proportion of prior work, including the packer created as part of this thesis, are based off T-VPack. T-VPack employs a two stage packing algorithm. The first stage groups LUTs and flip-flops together into the BLEs shown in Figure 2.1. The second stage groups BLEs together into logic blocks following a greedy algorithm. This algorithm first opens an unused logic block, fills the logic block with BLEs one BLE at a time (candidate BLEs are selected by a *gain* function), closes the logic block when it is full, and repeats the process until the entire input netlist is assigned into logic blocks. The gain function to select candidate BLEs is a weighted sum of a term that quantifies the shared connections of the candidate and another term measuring the timing criticality of the candidate. Though simple, the quality of results derived from T-VPack remain remarkably robust compared to the later BLE packers.

Several packers use T-VPack as a baseline then improve upon it by altering cost functions to better select what BLEs to pack together [11] [85]. The most notable of these packers is HDPack proposed by Chen et al. [17]. Similar to T-VPack, the HDPack's algorithm is greedy but with three stages instead of two. The first stage does the same BLE packing as T-VPack, the second stage seeds logic blocks with small groups of tightly connected BLEs, then the third stage performs greedy 1-by-1 packing of BLEs into the partially filled logic blocks. HDPack adds an interesting new *distance* term to the BLE selection gain function. To compute distance, HDPack first applies fast, min-cut, placement to put BLEs on a grid. BLEs that have a geographically closer distance on this grid have more affinity for being packed together than BLEs that are further. Over the 20 largest MCNC [100] circuits, the HDPack algorithm produces circuits that are on average 6% faster, have 24% lower minimum channel width, and uses 1% more CLBs than the T-VPack algorithm over approximately the same runtime. We consider this packer to be the current state-of-the-art in basic logic block packing.

There has been prior work on combined technology-mapping and packing. Lin et al. [53] investigated simultaneous technology-mapping and packing (SMAC) using techniques from FlowMap [20] (a technology-mapping alogrithm), while Ling et al. [54] investigated the application of techniques commonly found in technology-mapping to the packing problem. These methods attempt to find a logic depth optimal packing solution at the expense of duplicating logic. Overall, these methods produce better delay, worse area, and similar area-delay results compared to the T-VPack algorithm.

Others have explored coupling packing with placement. Singh et al. [85] proposed a packer, iRAC, that minimizes the worst-case Rent's exponent of a cluster. This packer is meant to be paired with a placer, iRAP, that also optimizes for the Rent's exponent but at a higher level of granularity. Their techniques result in a 24% reduction in overall area-delay product due to much better routability, but this comes with a caveat that the number of clusters increases by about 5% compared to T-VPack.

Some recent packers target basic logic blocks with even more lax constraints. Feng proposed two different packers [23] [24] that target basic logic blocks with no input pin constraints. With the constraints relaxed, techniques from the related field of circuit partitioning may be used to achieve results that might perform better than HDPack, though the experiments chosen by Feng make a direct comparison difficult. Currently, Feng's approach does not work well if the number of input pins becomes limiting.

Prior work in traditional packing show surprisingly strong area and delay results for greedy algorithms, which are simple to implement, versus more advanced algorithms, which are more complex. For example, despite the significant complexity of SMAC that guarantees optimal logical depth, SMAC appears to produce worse post-route area and worse post-route delay than the greedy packer HDPack [53] [17]. In the space that Feng's complex Rent's rule based packer works well, Feng implies (without actually doing a direct comparison) that his packer should produce 6% better delay and 15% lower minimum channel width than HDPack [24]. Thus, the simplicity and robustness of a greedy packer, with a well-tuned cost function, appears to be a reasonable choice for the goal of creating a general packer, versus using more complex heuristics.

2.3.2 Architecture-Specific Packing

The features of logic blocks found in commercial devices are too complex to be targeted by simple BLE packers. This led some researchers to create bespoke packing algorithms that are tailored for a specific architecture. Ahmed et al. proposed packing techniques for the Xilinx Virtex V FPGA [4]. These techniques include merging packing with placement when targeting soft logic blocks as well as packing memories in such a way that they align with the datapath of DSP blocks. Steiner created a packer that targets memories, multipliers, and soft logic slices of a commercial device, the Xilinx Virtex II Pro [89].

Although these packers are able to target very complex architectures, they do so by embedding architectural assumptions within the algorithm itself. This makes it difficult to apply these packers to target different architectures.

2.3.3 Flexible Packing

A general packer that can target a large space of different logic blocks addresses the lack of architectural complexity found in traditional BLE packers while at the same time avoiding the inflexibility of architecture-specific packers. There have been several different approaches to create these packers each providing useful insights on what an ideal general packer should (or should not) have.

Ni et al. [72] proposed expanding the notion of a BLE in a basic logic block to represent more arbitrary logic elements. Ni et al. created a tool for prepacking that is meant to be used with traditional BLE packers to enable these packers to target more complex architectures. Though this algorithm suffers from scalability issues, it does show the importance of having a flexible prepacker aid general packing. It also shows that limited interconnect flexibility makes placement within a logic element an important consideration during prepacking.

Depopulated crossbars in logic blocks are important architectural constructs that an architect would want to study. Wang et al. [92] as well as Lemieux and Lewis [48] proposed different ways to target logic blocks with depopulated crossbars. Their studies show that if one has depopulated crossbars in a logic block, then it may be necessary to have a router route nets within the intra-logic block interconnect. Work by Wang et al. suggests that the placement of BLEs within a logic block can also become an issue

for packing, given certain crossbar designs.

Cong et al. proposed a tool, RASP, where a variety of different interconnect in a logic block is permitted but the user is then responsible for creating custom heuristics to map to that logic block [21]. This approach defers too much development effort to the user. An ideal packer should encapsulate, or at least greatly simplify, the hints given to it by the user.

Paladino and Brown [74] proposed a packer that focuses on modelling control signals and carry chains within a soft logic block. The architect describes a set of design rules which governs the legality of how logic elements may or may not pack together. The packer then uses these design rules to filter out illegal intermediate packing solutions. This is an excellent work that is capable of targeting a wide space of different architectures without requiring the highly specialized expert knowledge required to operate Cong's RASP tool. Although Paladino's packer was only proven on soft logic blocks, we believe that such a tool, given sufficient modifications, could be made to target hard logic blocks too. However, the design rule language is still quite complex. An ideal, general, packer should instead infer these design rules based on the structure of the architecture itself.

Sharma et al. [84] proposed a placement algorithm that can explore FPGAs with arbitrary general interconnect. That work could be extended to target arbitrary interconnect within a logic block. Sharma's approach is to regularly sample the underlying interconnect during annealing-based placement by employing detailed routing repeatedly. However, this approach results in extremely long runtime. This suggests that detailed routing enables the exploration of a wide space of architectures but blindly relying on it is intractable.

Similar to the work done by Sharma et al. but in the context of FPGA packing, in my Master's and early portions of this thesis work, we proposed a greedy packer, called AAPack [57], that recognized the need for intra-logic block placement and routing when targeting a large space of different logic block architectures. This algorithm employed first-fit placement and negotiated-congestion routing to check the legality of all intermediate (partial) packing solutions. Compared to the former, the AAPack algorithm benefited from targeting a much smaller routing problem, namely working at the logic block level, but it still ran two orders of magnitude slower than T-VPack [64] on the simple FPGA architectures (recall though that T-VPack is specialized to only target these simple architectures).

In summary, the different approaches before this work targeting different aspects of general packing provide important insights for creating a general packer. We know that intra-logic block placement and routing becomes a concern once we allow freedom in specifying interconnect. We know that a detailed router for addressing the interconnect problem is powerful but can be very slow; thus simpler, but less powerful, design rule checks need to be used too. Finally, we know that prepacking is a robust technique for addressing situations of limited flexibility in packing and that this technique can be generalized to target more than just pairs of LUTs with flip-flops.

Chapter 3

VTR: FPGA Architecture Exploration Infrastructure

This chapter describes a flexible, robust, open-source, public infrastructure for FPGA architecture exploration and CAD research, called Verilog-to-Routing (VTR). The VTR project is a large scale endeavour that involves contributions from many collaborators around the world. We, the research teams of Jonathan Rose, Jason Anderson, and Vaughn Betz, at the University of Toronto, stand in an unusual position for this project because we are responsible for the VTR system as a whole. Thus, substantial work done as part of this thesis touches upon all parts of the VTR project. Due to this unique circumstance, the content of this chapter will describe the different parts of the VTR project that I was involved with, or oversaw, while giving credit to the many important contributors of each part.

Figure 3.1 shows the main components of VTR. These components include realistic benchmarks, sample architecture description files, a CAD flow, experiment scripts, and regression tests. The CAD flow consists of the Odin II tool for Verilog elaboration, ABC for logic synthesis and technology mapping, and VPR for packing, placement, and routing. We describe each of these different components in more detail before describing the process of engineering the VTR system as a whole.

3.1 Benchmarks

Due to the empirical nature of an FPGA CAD or architecture study, the benchmarks used in experiments can have a significant impact on that study's conclusions. One therefore wants a large number of benchmark circuits that represent the space of applications that one intends to optimize for. Gathering these benchmarks comes with challenges. There are far fewer, and far less substantial, open source circuits in the digital hardware community than in the software community. But gathering benchmarks is only one small part of the challenge, as one also needs to port the benchmarks so that they are compatible with the limited feature set of an open source flow. This can be quite labourious. Thus, there is value in having one research group create a set of benchmarks that are compatible with an open source flow, then make that set publicly available for other researchers to use.

In VTR, we make our best attempt at curating a representative benchmark set, while keeping the functionality of each circuit as close as we can to the original work. Benchmarks are chosen from a variety of different applications from a variety of different sources. This process is made possible only

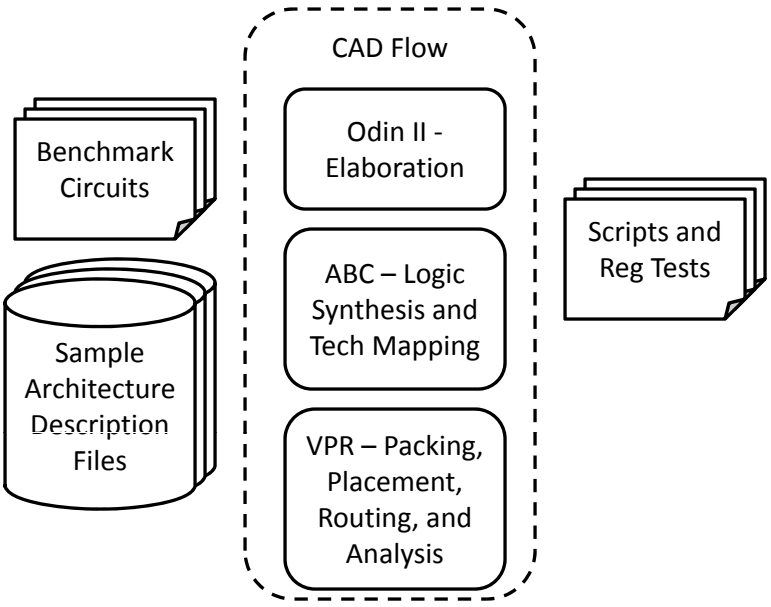

Figure 3.1: The VTR infrastructure includes benchmarks, architecture description files, CAD flow, experiment scripts, and regression tests.

through the goodwill of the community. Thus, we acknowledge and thank the many people who donated applications directly to VTR and to those who donated circuits to the open source hardware repository, OpenCores, from where we also gathered some of our benchmarks [44].

Benchmarks are, themselves, a form of software. If benchmarks are not maintained, then it is very easy for even basic runs of the tool to fail, let alone more sophisticated experiments. It is thus important that each release of the VTR CAD flow is paired with a set of benchmarks compatible with that release. Hence, before any release, we perform regular testing of the benchmarks on a variety of different architectures and flows.

Contributors involved with assembling these benchmarks and changing the benchmarks to be compatible with the VTR flow include Kenneth Kent, Cong Wang, Peter Milankov, Opal Densmore, Jingjing Li, and myself [80] [59]. I also acted as the gatekeeper of the benchmarks deciding which benchmarks to keep and whether or not a particular benchmark was properly converted. During a release, I was responsible for testing that the benchmarks are compatible with the release software.

Table 3.1 shows the statistics for the VTR benchmarks gathered using the VTR CAD flow that will be described later in Section 3.3. The columns from left to right are as follows: The name, number of 6-LUTs, number of multiplications, number of memory bits, number of 1-bit addition/subtraction operations, maximum addition/subtraction length, and application domain of the circuit. The benchmarks demonstrate a great diversity of different properties. Circuits range in soft logic size from a few hundred LUTs to nearly 100,000 6-LUTs. In terms of hard logic, circuits range from having no multipliers to having hundreds of multipliers, and from having no memories to having over 5 million bits of memory.

To measure the ratio of arithmetic functions to soft logic in the benchmarks, the CAD flow was configured such that addition/subtraction operations were always hardened. Later in this thesis, we will show that there are situations when some addition/subtraction operations are better off implemented in soft rather than hard logic.

Table 3.1: VTR benchmarks statistics gathered using the VTR CAD flow.

Circuit	# 6LUTs	# Mult	# Mem Bits	# Add	Max Add Size	Domain
arm_core	13,812	0	305,152	537	35	Proc
bgm	32,337	11	0	5,438	25	Finance
blob_merge	7,843	0	0	3,754	13	Image Proc
boundtop	2,846	0	32,768	309	19	Ray Trace
ch_intrinsics	425	0	256	0	0	Memory Init
diffeq1	294	5	0	132	33	Math
diffeq2	202	5	0	132	33	Math
LU8PEEng	21,668	8	46,608	3,251	47	Math
LU32PEEng	73,828	32	673,328	8,249	47	Math
mcml	94,048	27	5,210,112	24,302	65	Med Physics
mkDelayWorker32B	5,405	0	532,916	816	33	Packet Proc
mkPktMerge	225	0	7,344	45	5	Packet Proc
mkSMAdapter4B	1,819	0	4,456	431	33	Packet Proc
or1200	2,813	1	2,048	534	65	Soft Proc
raygentop	1,778	18	5,376	580	32	Ray Trace
sha	1,994	0	0	309	33	Cryptography
stereovision0	8,282	0	0	2,920	18	Comp Vision
stereovision1	7,845	152	0	2,388	19	Comp Vision
stereovision2	11,006	564	0	13,843	32	Comp Vision
stereovision3	174	0	0	28	11	Comp Vision

In addition to these comprehensive benchmarks, we also included other benchmarks to aid researchers in particular experiments. We include the traditional MCNC benchmarks [100] for use when evaluating against older works, and a set of floating-point benchmarks curated by Yu et al. [15].

3.2 Architecture Modelling

In this section, we describe two contributions to architecture modelling. First, we describe advances to the language used to describe architectures in VTR. Second, we describe a curated set of architectures that may serve as baselines for researchers conducting CAD and architecture experiments. The most notable of these is an architecture based on a commercial Stratix IV FPGA [19], that we call the *Extensive Archiecture* (EArch).

3.2.1 Architecture Description Language and Modelling

VTR employs an XML-based language to describe an FPGA architecture [60]. This language assumes an island-based architecture where logic blocks are defined on a discrete 2D grid and are surrounded by a sea of interconnect. An architecture description file begins with a set of parameters that describe general properties of the FPGA and the interconnect architecture. The logic blocks themselves are defined using a netlist-style language that was created as part of my master's thesis [56]. This logic block description language allows the architect to specify an arbitrary internal hierarchy of subclusters within a logic block, different exclusive modes of operation at any point in the hierarchy, *primitives* such as LUTs, flip-flops, memory slices, and other basic blocks, and interconnect that joins the different intra-logic block components together. The physical locations of different types of logic blocks are organized into columns. All logic blocks in a column must be of the same type.

In the present work, I extended the language used to describe a logic block to include information

about delays [80]. Interconnect delay is specified as point-to-point delays between pins inside the logic block. *Primitives* within a logic block, such as LUTs, flip-flops, memory slices, and other basic blocks, have their timing information specified based on whether the primitive is combinational logic or sequential logic. For combinational logic, the language allows the architect to specify a graph of input-to-output pin delays. To make this process less onerous for specifying large logic primitives (which would have many input-to-output paths), the architect may specify a constant delay between a set of primitive input pins to a set of primitive output pins. For sequential logic, the architect specifies setup time for input pins and clock-to-Q time for output pins. For sequential primitives that have some internal pipeline stages, we allow the maximum operating frequency of the primitive to also be specified. Hold time support, though important, is left to future work.

Below, we show a code example of how delays for a simple memory primitive may be specified:

```
<pb_type name="mem_512x64_sp" blif_model=".subckt single_port_ram" num_pb="1">
  <input name="addr" num_pins="9"/>
  <input name="data" num_pins="64"/>
  <input name="we" num_pins="1"/>
  <output name="out" num_pins="64"/>
  <clock name="clk" num_pins="1"/>
  <T_setup value="509e-12" port="mem_512x64_sp.addr" clock="clk"/>
  <T_setup value="509e-12" port="mem_512x64_sp.data" clock="clk"/>
  <T_setup value="509e-12" port="mem_512x64_sp.we" clock="clk"/>
  <T_clock_to_Q max="1.234e-9" port="mem_512x64_sp.out" clock="clk"/>
<\pb_type>
```

In this example, the *pb_type* XML tag describes an architecture primitive, named *mem_512x64_sp*, that can implement any user netlist block of type *single_port_ram*. The following *input*, *output*, and *clock* tags specify the input, output, and clock ports, respectively, of this memory along with their widths. The new timing extensions to the architecture description language include *T_setup* for specifying the setup time to the input ports of the memory and *T_clock_to_Q* time for specifying the delay from the arrival of a clock edge to valid data at the memory output. The *value* attribute specifies delay in seconds. The *port* attribute specifies which port this delay applies to. Finally, the *clock* attribute specifies which clock functions as the clock for this port. This last feature was created in anticipation that future logic primitives may have more than one clock.

We also extended the language to allow the architect to provide hints to the CAD tool to aid the tool on particularly difficult logic block constructs. As this "advanced user" feature is necessarily coupled with the CAD algorithms, we describe this extension later in Section 4.3.3.

Jeff Goeders at UBC further extended the language to support power specification [25]. Our experience integrating this feature into the VTR architecture modelling language is interesting because it illustrates an important difference between features that are standalone with features that are well integrated. Initially, Goeders' models ignored hard blocks, such as memories and multipliers. In addition, these models duplicated key parameters leading to risks that the models may become internally inconsistent. Vaughn Betz drove the need for power specification for hard blocks (through the use of optionally simpler models) and better synchronization of architectural parameters. Betz also verified that power modelling reasonably matched with the other area and delay models. I assisted in these tasks. One may now choose among multiple different levels of abstraction for power modelling in the architecture description file ranging from detailed transistor level models of wires and buffers to simpler models that measure power based exclusively on pin activity [25] [59].

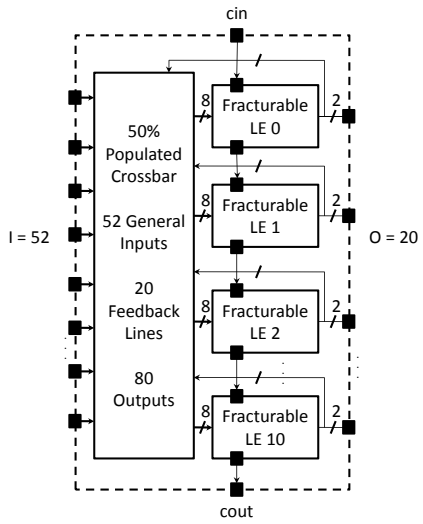

Figure 3.2: Architecture of the soft logic block in the flagship architecture.

3.2.2 The Extensive Architecture File

There is great value in having one, quality, architecture file as the "goto" architecture for a researcher new to the field. We wanted this architecture to resemble a modern, commercial chip. Such an architecture should contain the features that one would find in a modern FPGA including hard fracturable multipliers, hard memories with configurable aspect ratios, and soft logic blocks that have fracturable LUTs, carry chains, and a depopulated crossbar. Among the state-of-the-art commercial devices available, we decided to base our architecture off an Altera Stratix IV device [6] for practical reasons - we had available more device information for the Stratix IV device than any newer device at the time. We call this "goto" architecture, the Extensive Architecture (EArch).

Modelling for this architecture was done by Vaughn Betz, Charles Chiasson, and myself. Delay information was gathered primarily by using the Quartus II CAD tool to sample the different components of a Stratix IV device. Area information for the different logic blocks was scaled from Wong et al. [95]. Where more information was needed, such as when describing the resistance and capacitance of wire segments, we scaled the Berkeley predictive models [14] for 45 nm process to 40 nm to match Stratix IV.

Soft Logic Architecture

Figure 3.2 shows a soft logic block in EArch. This soft logic block has 52 general inputs, 20 general outputs, one clock pin (not shown), and dedicated *cin* and *cout* pins for fast carry arithmetic. The *cout* pin of a soft logic block drives the *cin* pin of the soft logic block directly below it. These carry signal pins do not have access to the general (inter-block) interconnect. This soft logic block has 10 fracturable logic elements and a 50% populated crossbar provides internal connectivity from the general inputs to the fracturable logic elements as well as between logic elements. The crossbar parameters are set to closely resemble a Stratix IV device [19]. A dedicated carry chain connects the fracturable logic elements vertically downwards.

A fracturable logic element is shown in Figure 3.3. It contains a fracturable LUT (explained in

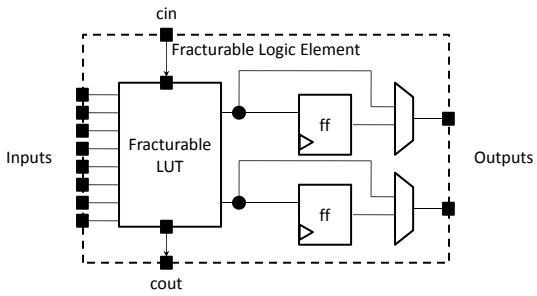

Figure 3.3: Fracturable logic element with optionally registered outputs.

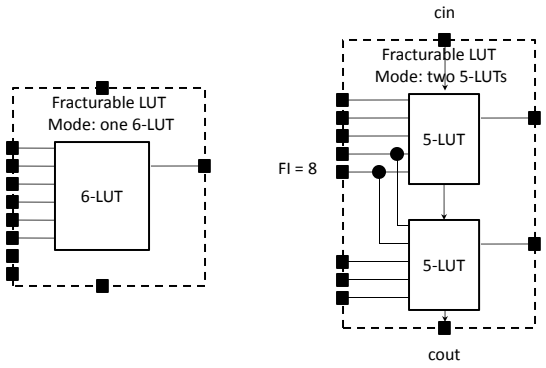

Figure 3.4: Fracturable LUT with an FI of 8. Each 5-LUT has an optional arithmetic mode that can implement one bit of addition.

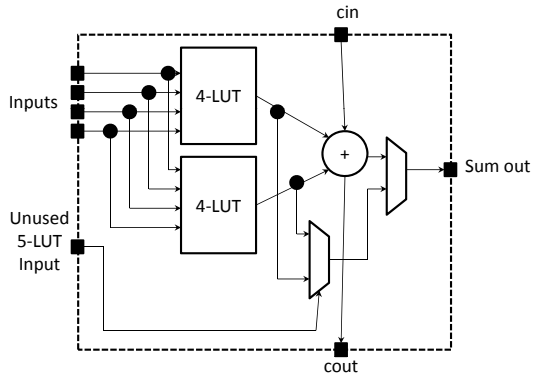

Figure 3.5: In arithmetic mode, the 5-LUT fractures into two 4-LUTs where each 4-LUT drives an adder input.

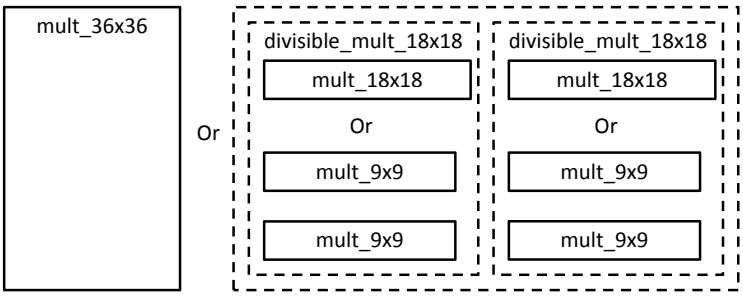

Figure 3.6: A fracturable multiplier can operate as one large 36x36 multiplier or two smaller 18x18 multipliers that can each further fracture to two 9x9 multipliers.

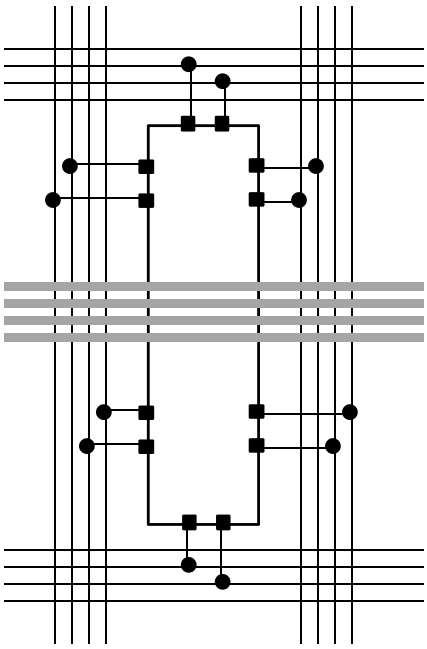

Figure 3.7: General interconnect wires run over tall logic blocks.

Section 2.1.1) with two optionally registered inputs. The fracturable LUT has two modes of operation, as shown in Figure 3.4. It can operate as either one 6-LUT or as two 5-LUTs that share two inputs (FI = 8). When operating as a 6-LUT, two LUT input pins are unused and the carry chain is unused. When operating in dual 5-LUT mode, each 5-LUT can optionally implement one bit of addition, which is implemented by a 1-bit hard adder. Figure 3.5 shows that hard adder along with the interaction between the LUT with the adder. In arithmetic mode, the 5-LUT is further fractured into two 4-LUTs with all inputs shared. Each 4-LUT drives one adder input. Fast, dedicated, interconnect for carry logic joins adjacent adders together.

Fracturable Multipliers

Figure 3.6 shows the architecture of a hardened multiplier block in EArch. This fracturable multiplier can operate as one large 36x36 multiplier or two smaller 18x18 multipliers that can each further fracture to two 9x9 multipliers. This style of fracturability is a simplified version of a Stratix DSP block [50].

Table 3.2: Configurable memory modes and aspect ratios.

Single-Port RAM	32Kx1 16Kx2 8Kx4 4Kx8 2Kx16 1Kx32 512x64
Dual-Port RAM	32Kx1 16Kx2 8Kx4 4Kx8 2Kx16 1Kx32

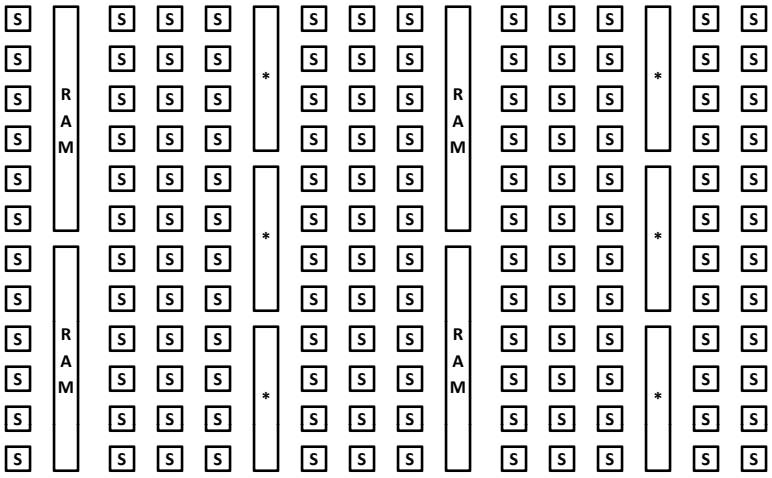

Figure 3.8: Core logic block layout for EArch.

In the core of the architecture, every eighth column, starting from the sixth, contains block multipliers exclusively and each multiplier block is four soft logic tiles high. The supply of multipliers was selected such that they become the limiting factor when determining the size of the FPGA for our most multiplier-intensive benchmark (stereovision2). In VTR, general interconnect wires run over tall logic blocks. This is shown by the bolded, gray, wires in Figure 3.7. Logic block pins do not connect to these wires. Since the multiplier is four tiles high, every multiplier block has three interconnect channels that run over it.

Configurable Memories

Stratix IV has block RAM sizes of 9 Kb and 144 Kb [6]. We chose to simplify our architecture by supplying one block RAM size of 32 Kb in EArch (32 Kb is the closest power of two to the geomean of the two Stratix IV RAM sizes). The configurable block memory in EArch has different modes of operation to better fit with the varying memory demands placed upon it by different user circuits. Table 3.2 shows the various modes of operation of the memory. It can operate in either single-port mode or dual-port mode. In single-port mode, the memory can have an aspect ratio varying from 32Kbits x 1 to 512x64. In dual-port mode, the memory can have an aspect ratio from 32Kbits x 1 to 1024x32. There is no 512x64 mode for dual-port mode because this mode requires an excessive number of pins, which are very costly.

As was the case with fracturable multipliers, general interconnect wires may run over the memory block.

In the core of the architecture, every eighth column, starting from the second, contains block memories exclusively and each memory block is six soft logic tiles high. Since embedded multipliers are also found on every eighth column (but starting from the sixth), this configuration results in no conflicts regarding whether or not a column is a memory or a multiplier. For our most memory intensive benchmark, mcml, the number of memories is close to being the limiting factor for determining the size of the FPGA.

Table 3.3: Routing architecture parameters.

Type	Single-Driver
Fc_{in}	0.15
Fc_{out}	0.1
L	4
Switch Block Type	Modified Wilton [60]

Table 3.4: Architecture files included in VTR.

Architecture Name	Description
$k4\_N4$	Legacy style soft logic only architecture
$k6\_N10$	Legacy style soft logic only architecture
$hard\_fpu\_arch\_timing$	Contains hardened floating-point logic blocks
$k6\_N8\_ripple\_chain$	Heterogeneous architecture with carry chains and non-fracturable LUTs
$k6\_N8\_unbalanced\_ripple\_chain$	Heterogeneous architecture with carry chains and non-fracturable LUTs
$EArch$	Heterogeneous architecture with carry chains and fracturable LUTs

Figure 3.8 shows the core logic block layout for EArch. Soft logic blocks are labelled s, multipliers are labelled $*$, and block RAM are labelled RAM.

Routing Architecture

The routing architecture parameters are listed in Table 3.3. We use a single-driver routing architecture [35] [50] [47]. The fraction of wire segments in a channel that are connected to a logic block input pin (Fc_{in}) is 0.15. The fraction of wire segments in a channel that are driven by a logic block output pin (Fc_{out}) is 0.1. All wire segments are length 4. The switch block pattern is a modified Wilton pattern [94] that attempts to balance mux sizes [60].

3.2.3 Other Architecture Files

We created a series of architecture files to serve as examples for other researchers to use. Table 3.4 shows some key architectures that are included in VTR. I created simpler variations of EArch including architectures with/without fracturable LUTs, carry chains, and hard blocks. Chiwai Yu created architecture files to explore hardened floating-point logic blocks [80].

A full summary of the different architectures released in VTR 7.0 [59] is provided in Section A. EArch itself is not in the VTR 7.0 release, will be made available in the next release.

3.3 CAD Flow

3.3.1 Odin II: Verilog Elaboration

Odin II is the Verilog elaborator used in VTR [37]. It was originally written by Peter Jamieson at Miami University in Ohio, but research and development of the tool is now primarily done by the research team headed by Kenneth B. Kent at the University of New Brunswick. The contributors in Kent's team were/are Andrew Sommerville, Sen Wang, Konstantin Nasartchuk, Ash Furrow, Jingjing Li, Bo Yan, and Conor McCullough. Since every other stage of the VTR CAD flow depends on elaboration, I worked

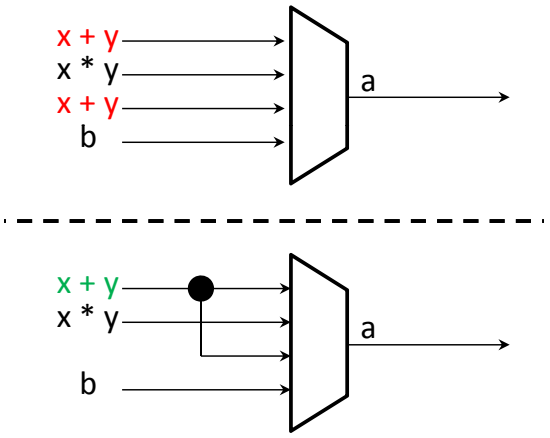

Figure 3.9: Example of common subexpression elimination for hard logic in Odin II. The duplicated $x + y$ operation may be merged together to save area.

closely with the Odin II team throughout my Master's and PhD to ensure a correct and compatible link between the elaborator and the downstream tools. Here, I briefly discuss my involvement with Odin II.

A key capability of Odin II is its ability to directly synthesize parts of a user circuit into the hardened logic (see Section 2.1.2) found in the FPGA architecture. But elaborating hard blocks requires more than just a direct translation. If the logical function in the user circuit is bigger than the hardened function available in the architecture, then the CAD tool must automatically *split* the user function into chunks that can physically fit. Odin II has the capability to automatically split multipliers, memories, and adders. Of these features, multiplier splitting was implemented by the Odin II team during my Master's. Memory and adder splitting was implemented during my PhD. I oversaw and verified the work done by Andrew Sommerville and Ash Furrow on memory splitting [87] as well as Sen Wang on adder splitting [59].

The introduction of hard logic into a user circuit creates boundaries at hard logic instances that make logic synthesis difficult. The logic synthesis tool that we are using, ABC, is unable to optimize across these boundaries. The VTR team lacked expert developers in ABC, thus, the responsibility of performing important, basic logic optimizations fell to Odin II. Odin II can perform basic logic synthesis operations such as sweeping away unused cones of logic, merging duplicate hard blocks together in a process formally known as common subexpression elimination, and propagating constants through hard blocks. An example of merging duplicate hard blocks is shown in Figure 3.9. The operation $x + y$ is needed in two different mux inputs. Rather than create two adders, the output of one adder may be used twice.

For certain situations, one may want to avoid hard logic altogether because the penalty from the boundaries imposed by hard logic may exceed the gain from using them. Thus, Odin II allows the user to specify conditions that, when satisfied by a particular hard logic instance, will replace that instance with soft logic.

Though hard logic block optimizations are performed during elaboration, the effects are more pronounced at the system level. Thus, being the one responsible for the VTR system as a whole, I discovered the initial problem caused by the boundaries imposed by hard blocks, proposed many of the optimizations later implemented by the Odin II team, then oversaw the implementations of these optimizations and

verified their correctness. Sen Wang, Conor McCullough, and Bo Yan implemented these optimizations under the supervision of Kenneth Kent.

Andrew Sommerville and Ash Furrow implemented functional simulation in Odin II [80]. The primary purpose of this simulator is to help verify the correctness of the circuits generated by Odin II. I played the role of a consultant for this work.

3.3.2 ABC: Logic Synthesis and Technology Mapping

One of the main uses of VTR is to explore FPGA architectures that contain new logic elements using realistic benchmark circuits that may be quite large in size. Thus, we need a logic synthesis and technology mapping tool that allows for the representation of black boxes and can scale to large circuits. We chose the ABC tool, developed by Mishchenko et al. [66], for logic synthesis and technology mapping, precisely for those reasons. Through the use of local, scalable, optimizations, ABC can match the quality of results of other open source logic synthesis tools while acheiving one to two orders of magnitude speedup [67]. Mischenko's team added rudimentary black box support to ABC so that arbitrary logic can pass through synthesis *completely* untouched. But, as mentioned earlier, this implementation introduced a host of quality issues that we handled in Odin II.

Although we do not directly collaborate with the developers who work on ABC, we do modify ABC to tailor it for certain specific needs in VTR. Jason Anderson added the *wiremap* [39] technology mapping heuristic to ABC. Rafi Rubin [81] and myself also made build changes to better deploy the tool to different platforms.

3.3.3 VPR: Packing, Placement, and Routing

The VPR tool [9] [60] [80] performs packing, placement, and routing in the VTR CAD flow. Packing maps a technology mapped circuit to the logic blocks found in an FPGA architecture. Placement finds physical locations for those logic blocks. Routing connects different logic blocks together. After routing, VPR then performs timing and power analysis of the resulting implementation.

Packing creates a netlist of logic blocks from a flat netlist of atoms. In line with the vision of a single CAD tool that can target many different architectures, we want a packer that can target a broad space of different logic blocks without code changes. In my Master's work, we proposed a simple packer, called AAPack [57] that is capable of legally mapping to a broad space of architectures, but the packer had excessive runtime and may perform poorly on certain complex architectures. For my PhD work, we made major contributions to the packing problem. These contributions will be discussed in detail in Chapter 4.

Placement assigns packed logic blocks to physical locations on the FPGA. The greatly expanded scope of architectures targeted by VTR created new challenges for placement. Work done before this thesis by Ted Campbell and myself involved initial support for heterogeneous logic blocks during placement [60]. Work done as part of this thesis involved supporting carry chain structures in placement through the use of relative placement. Thien Yu and Noruddin Ahmed did much of the initial development work, while being supervised by Vaughn Betz and Jonathan Rose [59]. But, the term ended before Thien and Noruddin were able to complete carry chain support so I was responsible for finishing the work. This included bug fixes to carry chain discovery and placement, as well as new features such as graphical visualization of the carry chain and carry chain interconnect modelling [59]. Tim Liu and myself further

refined placement for heterogeneous logic blocks. I improved accuracy for costing intermediate placement solutions while Tim further improved and sped up that computation.

Routing implements nets using the physical wires and switches available on the FPGA. The original timing-driven router in VPR, designed by Betz et al. [9], was very robust. Few changes needed to be made to target the benchmarks and architectures that we are interested in for VTR. The changes that we did make focused on speeding up minimum channel width search using the router. Minimum channel width search is the slowest stage in the VTR CAD flow [9] [59] [90]. The problem became especially noticeable with larger circuits where runtimes can exceed a week. I made some runtime optimizations to routing for handling high fanout nets based on techniques by Swartz et al. [90]. The rationale is that although high fanout nets tend to have a large bounding box, the bounding box for any particular target pin of that net should be much smaller. By restricting the area of the bounding box for each target, one can dramatically reduce the search space that the router explores, which in turn can reduce route time several fold for high fanout nets. The overall speedup on routing is approximately 5% to 20% depending on the circuit and architecture. Andre Pereira added more intelligence in the router for predicting when routing is impossible. This allows the router to exit earlier rather than waste time solving a problem that it cannot solve. I further refined Andre's heuristics to allow more precision. The ability for the router to exit early has a substantial impact on our largest circuits that also have the longest route times. This optimization reduced the total time to run all VTR benchmarks on one architecture from over a week to just a few days.

A flexible logic block architecture implies that the VPR models for timing must likewise be flexible. VPR has two models for timing, one for *before* packing (when the user netlist has no hierarchy, as explained in Section 2.2.2) and another for after packing (when the user netlist is assembled into hiearchical logic blocks). Originally, the pre-packing timing model used two constants to model delay: delay for a primitive and delay between primitives. I updated the timing models in packing to include actual delay information gathered from the architecture file. The delay graph now uses the timing information for each primitive [80]. Michael Wainberg improved upon this by assigning a constant delay for connections between primitives based on inter-logic block interconnect delays [59]. The user may optionally override this with a different value using a command-line option.

Originally, the post-packing timing model in VPR contained hard-coded architectural assumptions regarding the delay model within a logic block. This model assumed a two-level hierarchy for logic blocks that no longer holds true with the broader architectural space for logic blocks. I generalized the timing models to use architect-defined delay information from primitives, intra-logic block interconnect, and inter-logic block interconnect [80].

Jeff Goeders added power analysis to VTR. After routing, VPR applies the activities of various nets, as specified in a user-provided activity file, to the power models, specified in the architecture description file, to measure dynamic and static power consumption of the final mapping [25]. Vaughn and I were heavily involved with the integration of power analysis with VTR. Vaughn drove for features that enable power modelling of hardened logic and he spent time validating power measurements. I assisted Vaughn in these activities.

There were multiple important advancements to VPR where I had minimal involvement. Michael Wainberg added more advanced timing features to VPR including multi-clock timing analysis and clock constraints. He also added support for specifying clock constraints using the industry standard called SDC [59]. Miad Nasr and Jason Anderson added a feature that allows one to extract a netlist, with

Table 3.5: Statistics from mapping the VTR benchmark circuits to EArch through the VTR CAD flow.

Circuit	Min W	Crit Path Delay (ns)	Num CLB	Total VTR Runtime (s)
arm_core	136	19.49	1067	2359
bgm	108	30.07	2916	3822
blob_merge	92	13.79	560	412
boundtop	74	6.51	200	85
ch_intrinsics	60	4.01	26	11
diffeq1	70	16.81	25	28
diffeq2	56	12.85	15	24
LU8PEEng	124	78.29	1852	3840
LU32PEEng	176	81.89	6208	65012
mcml	170	43.82	6044	30243
mkDelayWorker32B	92	7.29	422	1140
mkPktMerge	48	4.92	15	46
mkSMAdapter4B	88	6.03	152	76
or1200	114	8.58	214	336
raygentop	88	4.97	162	81
sha	74	10.11	187	57
stereovision0	66	4.29	860	275
stereovision1	122	5.92	854	1669
stereovision2	136	13.94	1906	12446
stereovision3	34	2.98	13	4

delays annotated, after routing. This allows users to perform accurate timing simulation on their circuit [59].

3.4 Example Result Runs of the Entire Infrastructure

This section shows the quality of results when the VTR benchmark circuits (described in Section 3.1) are mapped to EArch (described in Section 3.2.2) using the full VTR CAD flow. The machine used in this experiment is a 64-bit Intel Xeon 5160 running at 3 GHz in single core mode with access to 8GB of physical memory. Table 3.5 shows the final stats. The leftmost column lists the circuit name. Following that are the minimum channel width to route the benchmark, the critical path delay in nanoseconds from a channel width 30% greater than min W, the total number of soft logic blocks used, and total VTR runtime in seconds.

These statistics show the current state-of-the-art for open-source FPGA architecture exploration. It shows that VTR can target architectures with depopulated crossbars, fracturable LUTs, carry chains, memories, and multipliers on real benchmarks that approach 100K 6-LUTs in size.

Investigating the multiple instances of poor performance in these results reveals the holistic nature of VTR. For example, a large portion of the 19 hour compile time for LU32PEEng, a circuit with just over 70K LUTs, is in minimum channel width search that we feel can be further sped up through better algorithms. On the other hand, the low clock frequencies for LU32PEEng, mcml, and other circuits primarily arose from a slow implementation of division in the benchmarks themselves. Our implementation for division in the benchmarks is much less efficient than the proprietary dividers used in the original benchmarks. Rather than being just one-off cases, these cases resemble the kinds of issues involved with, large, system-level, FPGA CAD infrastructures because architecture, benchmarks, and CAD are all, ultimately, interrelated.

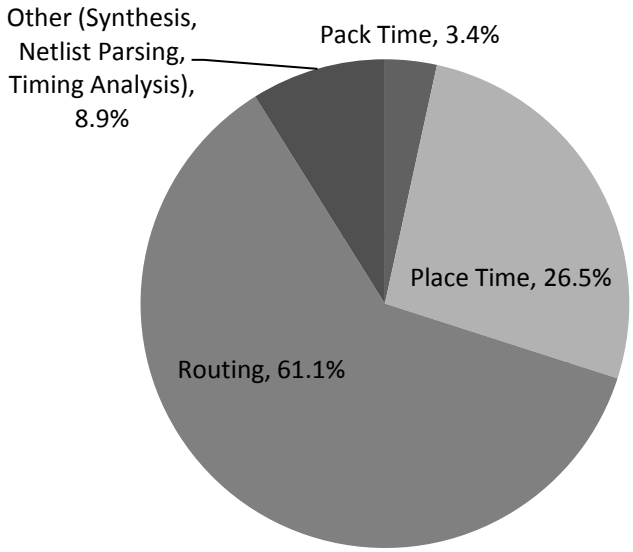

Figure 3.10: Breakdown of percent of total runtime for the different CAD stages for a minimum channel width search experiment. Values shown are the arithmetic mean of the relative runtime for each stage across the VTR benchmarks.

Figure 3.10 presents a breakdown of the percent of total runtime for each of the major stages of the CAD flow. Minimum channel width search occupies more than half of total runtime for each circuit on average. Placement time takes up just over one quarter of total runtime. Other stages of the CAD flow, such as elaboration, logic synthesis, packing, and timing analysis, occupy less than 15% of total runtime. In this experiment, the placement option inner_num was set to 10, which is higher than the recommended value from [59], because we found that increasing placement effort led to an overall reduction in runtime because a better placement results in faster min W search time.

In addition to its use as an architecture exploration tool, some users employ the VTR CAD flow in a production environment to target a specific FPGA architecture [71] [68]. It is thus interesting to measure how the CAD flow performs in a production style environment. First, a fabricated FPGA must have a fixed channel width. Thus, for this experiment, we ignore minimum channel width search time and instead take fixed-channel width route time at 1.3 times minW as total route time. Second, since minimum channel width search time is no longer a factor, placement effort can be substantially reduced with only a marginal degredation in results, so inner_num is set from 10 to 1. Table 3.6 shows the results of mapping the VTR benchmarks to EArch using these settings. We see a 6.8-fold speedup for a 2% delay penalty when comparing a production style run of VTR to an architecture exploration run. Figure 3.11 shows the runtime breakdown for a production flow. Without minimum channel width search, the stages before place and route consume a relatively larger pecentage of total runtime.

As issues that affect runtime and quality continue to be improved upon, and as researchers explore new avenues of research opened up by VTR, we foresee a future where quality of results will be significantly better than the results presented here.

Table 3.6: Statistics from mapping the VTR benchmark circuits to EArch with settings tuned for a production environment.

Circuit	W	Crit Path Delay (ns)	Num CLB	Total VTR Runtime (s)
arm_core	198	19.71	1067	218.93
bgm	146	29.39	2916	530.30
blob_merge	130	14.14	560	59.24
boundtop	102	6.58	200	20.29
ch_intrinsics	78	3.99	26	3.36
diffeq1	70	17.10	25	4.86
diffeq2	68	13.06	15	4.54
LU8PEEng	172	82.18	1852	402.61
LU32PEEng	242	81.09	6208	2614.80
mcml	222	47.34	6044	2594.10
mkDelayWorker32B	128	7.59	422	82.38
mkPktMerge	64	4.98	15	10.93
mkSMAdapter4B	108	5.91	152	16.65
or1200	136	8.86	214	29.76
raygentop	114	4.95	162	20.83
sha	94	10.27	187	18.43
stereovision0	86	4.61	860	86.16
stereovision1	164	6.05	854	110.25
stereovision2	196	14.23	1906	401.70
stereovision3	44	3.07	13	2.49

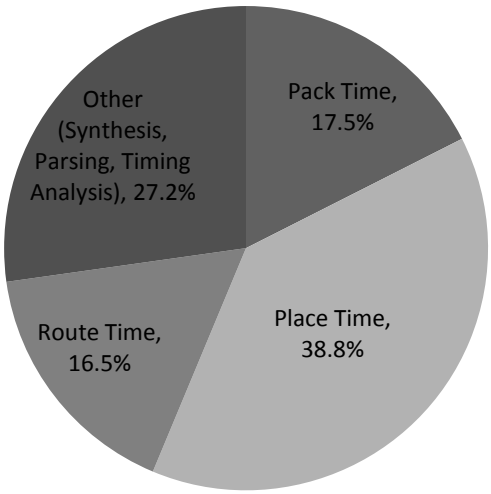

Figure 3.11: Breakdown of percent of total runtime for the different CAD stages for a production environment. Values shown are the arithmetic mean of the relative runtime for each stage across the VTR benchmarks.

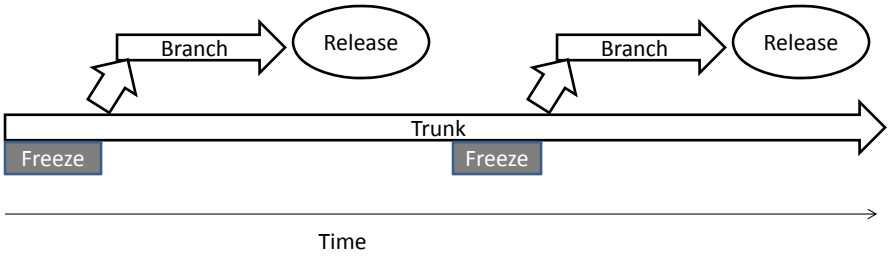

Figure 3.12: VTR software development follows a typical trunk/branch process.

3.5 Software Engineering

This section describes the software engineering behind the VTR project. We briefly illustrate the problems and solutions that we encountered with the intention that this may help others with the software engineering behind their projects.

The unique circumstances of academia give rise to interesting challenges in the creation of large-scale software. First, contributors to the VTR project are distributed around the world, which can make collaboration difficult. Second, researchers choose, rather than are assigned, what to research; therefore the VTR project is susceptible to losing key experts if those experts change interests or priorities. Third, many contributors have a short turnover time compared with industry: a summer student contributes for 4 months, a Master's student contributes for 1-2 years, and a PhD student contributes for 3-4 years. This distributed, ever-changing, at times fickle, staff can make the development, management, and testing of the VTR project difficult.

3.5.1 Software Development

We employ a typical *trunk/branch* software development process for VTR, as illustrated in Figure 3.12. The trunk represents the current and latest shared code in active development. Changes are made incrementally to the trunk. When sufficient new features are added to justify a full release, at approximately every two to three years, we start the release process. This process starts by putting the trunk into a state called *code freeze*. In this state, only bug fixes are allowed, no new features are added. Once the team has sufficient confidence in the quality of the code, a branch is started. A branch is a completely separate snapshot of the trunk which may now diverge from the trunk in code. The branch undergoes testing in preparation for a release while the trunk is *unfrozen* so that developers can continue development of new features independently of the release process. When the branch reaches a suitable state, we release a packaged version of the branch. Unlike larger companies, we do not have spare development capacity for patching releases, so code changes to the branch stops at the release.

I led the VTR 6.0 and 7.0 release as part of my dissertation work. To aid in this process, I automated the process of creating and validating a release candidate. However, manual spot checking of the release still needs to be done because automation lacks the scope of judgement that a human has.

Due to the large number of people on the project, we placed a requirement that the trunk must always be stable. If this condition was not met, then people responsible for making the current state of software unstable are responsible for fixing it.

Figure 3.13 shows the source-controlled, software organization of the trunk of VTR. Every solid

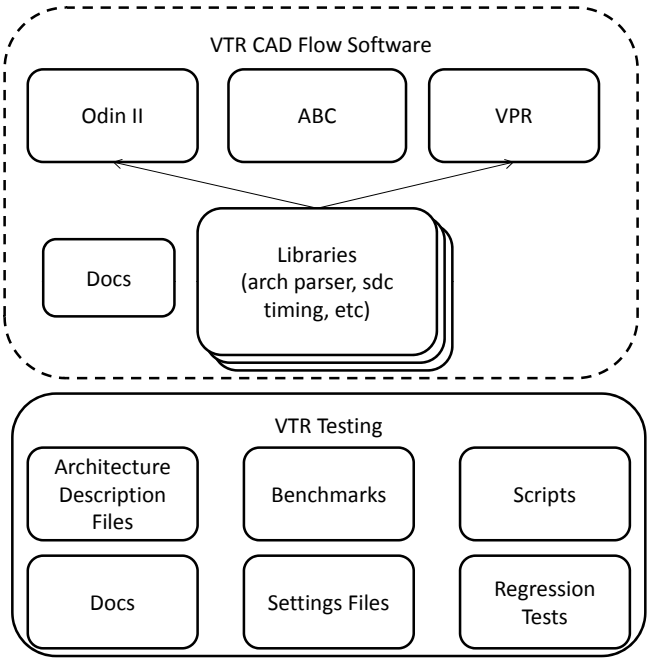

Figure 3.13: Organization of the VTR 7.0 release.

block represents a folder. VTR has a folder for each of the three major CAD tools (Odin II, ABC, VPR). Various libraries, such as libraries to read the architecture file, that support Odin II or VPR, have their own separate folders. We provide documentation in the base folder that describes the overall organization and use of VTR. Within each major folder, there exists documentation that describes the function and organization of that folder. VTR has a test folder, called *vtr_flow*, that contains architecture description files, benchmarks, scripts for running and parsing VTR, documentation, settings, and the tests themselves.

3.5.2 Management

To work with people around the world, we held, and continue to hold, weekly meetings, initially over Skype, then later over Google Hangout. Each research site would then have smaller meetings for each local team. Jonathan Rose took a leadership role running the meetings and driving the common vision for the VTR project. This is essential for keeping the team cohesive. Vaughn Betz also played a leadership role taking on increasing responsibility to drive the project forward.

Managing the people in the VTR project goes beyond just division of responsibility. Often times, system level issues can arise that are not detected from looking at any one part in isolation. The high turnover rate of contributors further exacerbates this problem because often times contributors do not have the time to understand the nuances of the system. Hence, there is a need for one person who is responsible for the project as a whole. I decided to take this role which includes performing integration work, testing the full infrastructure, and providing support where help was needed to ensure good operation of the VTR project.

Due to the short turn-over time of contributors, it becomes especially important that faults are detected early in the software development process rather than later. Suya Liu [55] worked on changing

the build process to detect problems early. She added stricter checks in the compile process. Suya and I also cleaned VPR of memory leaks, though such work is currently ongoing in Odin II. Daniel Chen [16] and Jeff Rudolph [82] were involved in improving error logging and messaging in VTR to help both developers and users detect and solve problems quicker.

3.5.3 Testing

Early on, I would test VTR manually but this does not scale well and is prone to human error. Hence, we moved towards a more automated way to check for the good operation of the system. Jeff Goeders and myself worked on regression tests that tie the different components of VTR together. These tests let a user automatically map different circuits to different architectures using various CAD settings, automatically parses out the results, then compares these results to a set of golden results. *CAD noise* creates a complication to testing. Due to the nature of the heuristic algorithms used in the CAD flow, something as simple as changing the order of nets or changing how rounding is done can change quality of results while still generating correct solutions. So forcing equivalent results in testing may end up creating many false positives due to this "noise" in quality of results. To manage this problem, guardbands are placed around golden results. These guardbands, and the golden results themselves, need to be updated and tuned, as the CAD algorithms get optimized, so that the tests capture real errors while avoiding false positives.

Using this infrastructure Norrudin Ahmed [3] created an automatic test infrastructure using BuildBot [1] that regularly runs regression tests at various levels of comprehensiveness that check both correctness and quality of results. In addition, Norrudin added measurements on average quality of results to this infrastructure to track performance. The VTR automated infrastructure was then further augmented by Andrew Sommerville and Kenneth Kent, who added unit tests for the Odin II elaborator. With this system, new changes to the source code are verified automatically so that developers know when a problem happens at the system level. After Norrudin's summer term ended, I became responsible for maintaining the BuildBot automated test infrastructure.

Automated testing, though useful, does not cover new functionality so any new features still need to be tested by the developers manually before they can automated. Other manual work includes certain kinds of benchmarking. For example, Tim Liu did benchmarking work to check quality of results across VTR versions [59] to establish history of quality of results from before automated testing.

3.5.4 Public Deployment

The purpose of a release is to create a known stable version of the software that the public can depend on. We discussed the process leading up to a release in Section 3.5.1. Here, we discuss the deployment considerations surrounding a release such as licensing, support, and actual deployment.

For VTR to be widely useful, we want an open source license that maximizes what others can do with the tool, while at the same time minimizes our liability. This led us to choose the MIT license [36] for VTR. In short, with the MIT license, one can use VTR for free but we are not liable for the outcome. Since VTR is built from multiple different tools with different licenses, we negotiated with various authors to consolidate the release into one license (with the exception of ABC, but the license for that software is in practice identical to the one we use).

Software that is not maintained will eventually fall into disuse as various features become outdated

or incompatible. To combat this, we have a simple bug reporting/tracking system using the google code infrastructure. At this current snapshot in time, we have received 70 issues with 27 issues still open.

The source code itself can be downloaded directly from the google code repository at `https://code.google.com/p/vtr-verilog-to-routing/`.

3.6 Contribution

The primary contribution of VTR is in enabling CAD and architecture research. We can begin to see its impact through some recently published works. Zgheib et al. [101] studied a newly proposed soft logic block based on programmable and-inverter cones using VTR as part of her experimental infrastructure. Murray et al. [69] released an even larger set of benchmarks for FPGA research along with a CAD flow to map those benchmarks to a VTR compatible flow, which together is called TITAN. Brant and Lemieux [13] proposed virtualizing FPGAs by creating an FPGA overlay over an existing commercial architecture, then using VTR to map to the virtual FPGA. Purnaprajna and Ienne [78] investigated a mix of LUTs and hardened muxes for logic using the VTR infrastructure. Hung et al. [32] performed a head-to-head comparison between the VTR CAD flow versus a commercial flow and created a way for the output of VTR to interface to a commercial tool. Hartig et al. used VTR to map circuits to a structured ASIC [27]. Lingli Wang's group at Fudan University is using VTR to target an FPGA that they are fabricating [93]. The diversity of research that VTR already enables speaks well to its future value.

VTR is/was used as part of commercial CAD infrastructure. Texas Instruments used VTR to map to an internal FPGA that they were investigating [68]. A new FPGA startup, called Efinix, use VTR as part of their backend [71]. One of the founders, Ngai [71], says that VTR was invaluable to the startup for it enabled a much shorter time to market.

Reproducibility is a major cornerstone of the scientific method but reproducibility can be difficult to achieve in FPGA CAD and architecture research. The FPGA CAD flow has become so complex that most publications must carefully curate what implementation details are important to show. If a researcher omits a particular parameter or optimization that happens to be important, then it can make reproducibility extremely difficult. VTR helps address this concern by reducing infrastructure differences among researchers. At minimum, the results packaged with VTR should be easily reproduced simply by running the experiment flow packaged with the tool. Hence, by improving the reproducibility of future studies, VTR helps with the scientific rigour that is vital in good research.

From 2012 to date, VTR has accumulated over 700 unique downloads and its 2012 publication [80] has accumulated 89 citations in just two years. The VTR project has further underscored the value that large collaborative endeavours, though challenging, is necessary for overcoming certain barriers that affect the research community that are too large for a single group to resolve.

Chapter 4

Architecture-Aware Packing for FPGAs

To study a variety of novel logic block architectures, one needs a flexible packer that can target a broad architectural space. We begin this chapter with a formal definition of the architecture-aware packing problem for FPGAs. We then describe a packing algorithm to solve the packing problem.

4.1 Problem Definition

Figure 4.1 shows the inputs and output of the packing problem. The inputs to packing are a technology-mapped netlist and a description of the architecture. The output of packing is a netlist of logic blocks that are configured to implement the user netlist. This section formally defines each of these terms.

4.1.1 Architecture Definition

A description of the FPGA architecture is one of two inputs to packing (the other input being the user netlist). It defines the *logic blocks* in the FPGA. A logic block performs computation, storage, or some combination of both. Logic blocks that have the same properties are classified as belonging to the same *type*. Modern FPGAs have multiple types of logic blocks, such as DSP blocks, configurable RAM, and soft logic clusters, where each type is available in different quantities. The total number of logic blocks

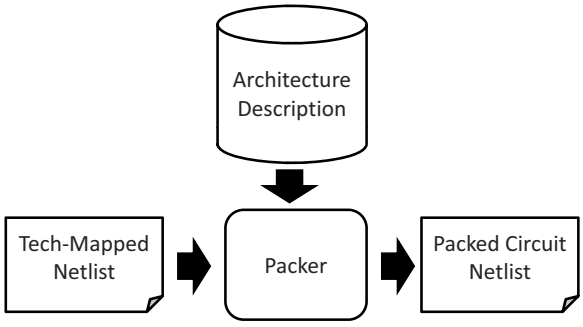

Figure 4.1: The inputs and outputs of the packing problem.

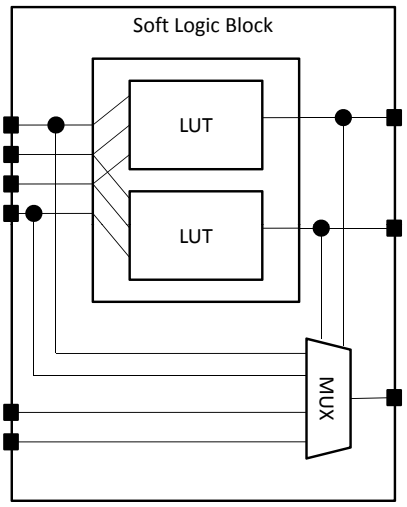

Figure 4.2: Examle of a simple logic block

is represented by the variable Z. The architect may choose a fixed Z or a variable Z. A variable Z is a convenience feature for the architect; it follows a long history [91] [9] of allowing the CAD algorithms to optimize for certain architectural parameters (in this case, FPGA size). Let the logic blocks be $\mathbf{B} = \{b_1, b_2, ...b_Z\}$.

Logic blocks contain *primitives*, where each primitive can implement a single basic block, that we call an *atom*, of the input user netlist. Let primitive p_{ij} represent the j^{th} primitive in logic block b_i. The particular sequential/combinational logic functionality that a primitive implements, such as a LUT, flip-flop, memory slice, or adder bit, is called a *logic model*. The function $MODEL(p_{ij})$ returns the single logic model of the primitive p_{ij}. Let $\mathbf{P}$ be the set of all primitives in the FPGA architecture.

All logic blocks and primitives have input/output pins that connect these blocks/primitives to other blocks/primitives via configurable interconnect. Configurable interconnect is modelled as a directed graph consisting of nodes and edges. Nodes in this network are either primitive input/output pins or intermediate nodes that represent interconnect resources (such as muxes or wires). Let v_{ij} represent the j^{th} interconnect node in logic block b_i. Let $\mathbf{V}$ be the set of all interconnect nodes across all logic blocks in the FPGA.

We allow the architect to specify any arbitrary interconnect graph within a logic block, with any number of intermediate nodes or edges. The logic block pins themselves are represented as intermediate nodes in this graph. A detailed interconnect model *between* logic blocks is unknown because the packing stage of the CAD flow precedes placement-and-routing. Thus, we define a simplified model of that interconnect for packing – we assume that any logic block pin has an edge (can connect without restriction) to any other logic block pin.

Figure 4.2 shows an example of a simple logic block. This logic block consists of two 3-input LUT primitives and one mux primitive. The mux primitive is intended to be part of the *logic* function, not the internal-to-the-block interconnect. The LUT primitives share two of their three inputs. The logic block has six input pins and three output pins.

The interconnect graph representation of this simple logic block is shown in Figure 4.3. Black squares are intermediate nodes, nodes labelled 't' are sink nodes, nodes labelled 's' are source nodes. Each pin

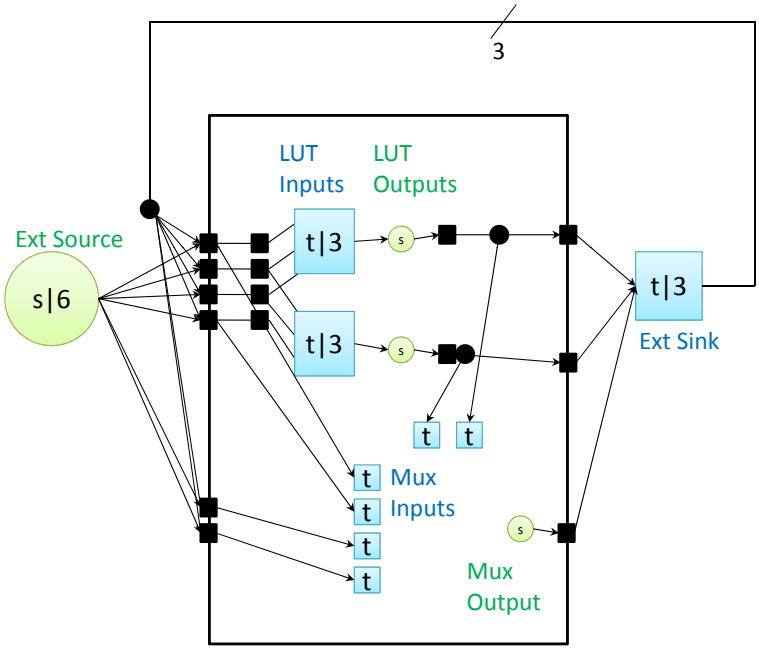

Figure 4.3: Graph representation of the interconnect of the simple logic block

in the logic block is an interconnect node. In addition, the external source and the external sink nodes represents interconnect outside the logic block. The number of nets that are allowed to drive a node is called *capacity*. All nodes have a capacity of one unless otherwise specified. The external source node has a capacity of 6 because up to 6 inter-logic block nets can drive the logic block and the external sink node has a capacity of 3 because the logic block can drive up to 3 inter-logic block nets. The mux has six sink nodes (data and select) and one source node. The two LUTs require special modelling. Since a LUT has logically equivalent inputs, we model all three input pins as one sink with a capacity of 3. Also, since a LUT can be configured to implement an interconnect wire, we add an edge that joins the inputs of the LUT to the output. We will revisit this simple logic block again when we describe how to pack a netlist into it.

Modern logic blocks have hierarchy as well as different modes of operation. A mode is specific functionality in a logic block, or part of the logic block, that when used, prevents certain other functionality from being used. A soft logic block that contains BLEs, where each BLE contains a LUT and flip-flop pair, is an example of hierarchy. Memory blocks in modern FPGAs have different modes of operation and may be configured to different aspect ratios. A soft logic block that contains fracturable LUTs, where each fracturable LUT can operate as either a big LUT or two smaller LUTs with shared inputs, is an example of both hierarchy and modes in the same logic block. Thus it is important to capture hierarchy and modes in the definition of an architecture.

Hierarchy is represented using *subclusters*. A subcluster is a group of primitives, interconnect nodes, and/or other subclusters. The earlier example of Figure 4.2 shows a logic block that contains one subcluster of two 3-input LUTs. Any subcluster, primitive, or interconnect node has one parent. That parent must be another subcluster or the logic block itself. Logic blocks are the exception as they do not have a parent; they are top level subclusters. Sibling subclusters are not allowed to share descendants. Let subcluster s_{ij} represent the j^{th} subcluster in logic block b_i. Let $\mathbf{S}$ be the set of all subclusters

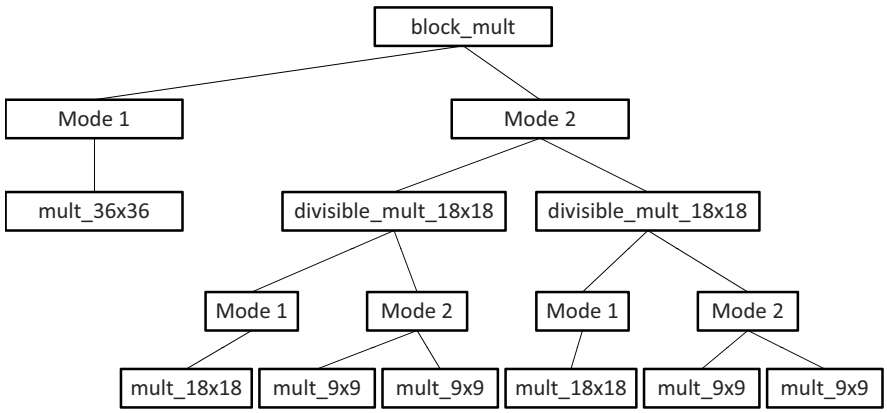

Figure 4.4: Example of a configurable multiplier that has multiple levels of hierarchy and multiple modes of operation

across all logic blocks in the FPGA. Apart from its use as a convenience feature, hierarchy enables the specification of modes.

A mode is modelled as a special subcluster that can contain other subclusters, primitives, or interconnect nodes. To maintain mutual exclusivity, no two sibling modes may be *active* at the same time. A mode is considered *active* if any of its descendant subclusters or primitives are used. Let mode m_{ijk} represent the k^{th} mode of subcluster s_{ij}. Let $\mathbf{M}$ be the set of all modes in the FPGA architecture.

Figure 4.4 provides an example of how modes and subclusters may be used in the modelling of an FPGA logic block. Suppose that a multiplier logic block can operate as one large 36x36 multiplier or as two 18x18 multipliers, and that each 18x18 multiplier can further optionally operate as two 9x9 multipliers. Figure 4.4 shows how the subclusters and modes of such a multiplier would be defined. The logic block is called block_mult. The logic block can operate in mode 1, which contains one 36x36 multiplier primitive, or it can operate in mode 2, which contains two divisible 18x18 subclusters. Each 18x18 subcluster can operate in a mode that contains one 18x18 multiplier primitive or two 9x9 multiplier primitives.

4.1.2 Netlist Definition

A technology mapped netlist is a flattened view of a user circuit that serves as the second of the two inputs to packing. The netlist consists of *atoms* and *nets*. Atoms are basic blocks that, during packing, will be assigned to physical *primitives* (defined above) found in the FPGA. Typical examples of atoms include LUTs, flip-flops, memory slices (one bit data memories that may be grouped together to form a wider memory), and adder bits (one bit additions that may chain together to form a wide addition). Let the set of all atoms in the netlist be $\mathbf{A} = \{a_1, a_2, ..., a_N\}$.

An atom has one *logic model* that identifies the specific sequential/combinational logic functionality of that atom. Each primitive in the architecture can only implement atoms of a particular logic model. For example, suppose an FPGA contains LUTs and flip-flops, then atoms of logic model LUT can only be assigned to LUT primitives while atoms of logic model *flip-flop* can only be assigned to flip-flop primitives. The function $MODEL(a_i)$ returns the logic model for the i^{th} atom.

Nets join atoms together. Let the set of nets be $\mathbf{E}$. Every net e_i in $\mathbf{E}$ is driven by exactly one atom

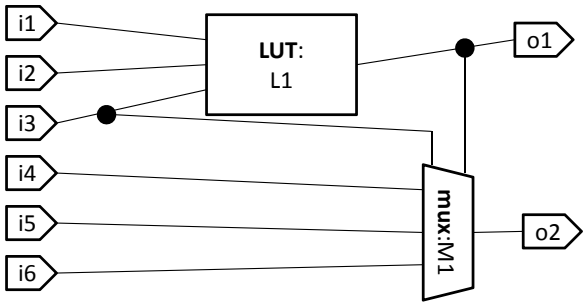

Figure 4.5: Example of a simple netlist.

output pin and connects to one or more input pins of its current atom or other atoms.

Figure 4.5 shows an example of a netlist with 10 atoms and 8 nets. This netlist has six atoms of logic model *input*, one atom of logic model LUT, one atom of logic model *mux*, and two atoms of logic model *output*. The 8 nets join these atoms together. We will revisit this example below to show how this netlist gets packed to an architecture.

4.1.3 The Packing Problem

The packing problem is defined as the determination of a *legal* assignment of input netlist atoms A to input FPGA architecture primitives P, in such a way as to optimize one or more aspects of the result such as area, speed, or energy.

We leave the definition of the cost function flexible, as it is a subject of investigation itself. A good cost function should give a lower cost for packing solutions that result in a smaller FPGA, lower critical path delay, lower minimum channel width, lower energy usage, and lower wirelength usage after placement and routing.

A legal solution is an assignment that satisfies the following conditions:

1. All atoms are assigned to a unique primitive.

   $\forall a \in \mathbf{A}$, $\exists$ exactly one p in $\mathbf{P}$ *s.t.* $a \mapsto p$.

2. The primitive that is assigned can implement that atom.

   If $a \mapsto p$ then $MODEL(a) = MODEL(p)$.

3. No two atoms map to the same primitive.

   If $a_i \mapsto p_{rs}$ and $a_j \mapsto p_{tu}$ then $p_{rs} \neq p_{tu}$ for all $i \neq j$.

4. The packed solution is routable assuming full connectivity between logic blocks.

   Given a logic block interconnect graph, used primitive input/output pins map to sink/source nodes on the graph. Since we assume full connectivity between logic blocks, the connectivity of a logic block may be checked independently of other logic blocks, because nets either route inside the logic block or to/from a logic block pin. Hence, nets with connections outside the logic block map to external source and external sink nodes, as shown earlier in Figure 4.3. A solution is routable if and only if for all nets e_i in $\mathbf{E}$, there exists a *directed path* from each source to each sink of e_i in

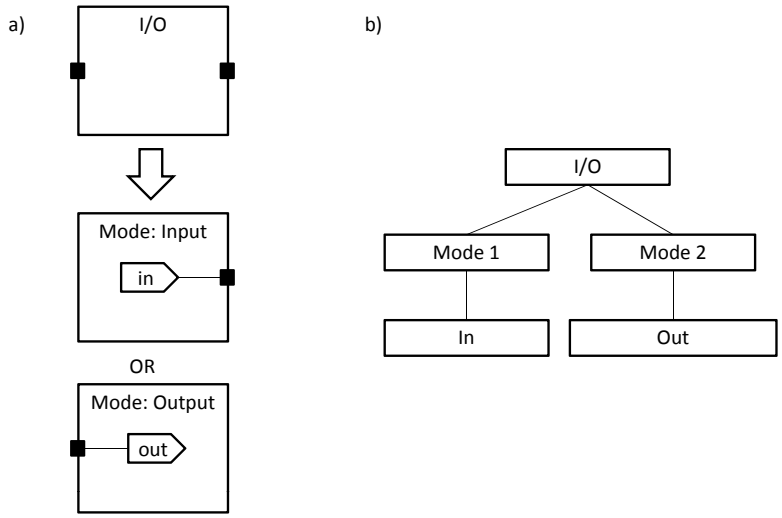

Figure 4.6: a) Example of an I/O logic block. b) The hiearchical representation of that I/O block.

V such that the number of nets that share the same node v does not exceed the capacity of that node, where $v \epsilon \mathbf{V}$. A path is defined as a sequence of edges which connect a sequence of vertices.

5. The mutual-exclusivity of modes is respected

   A mode is called *active* if in the final packed solution, the mode contains a primitive p that has an atom mapped to it or the mode has an interconnect node v that has a net that routes through it. The mutual-exclusivity of modes are respected if all subclusters have at most one active mode. More formally:

   $\forall m_{ijk}$ in **M**, if m_{ijk} is active then m_{ijt} is not active for all $t \neq k$.

4.1.4 Example of Packing

This example illustrates what constitutes a packing solution. Let the input netlist be the one shown in Figure 4.5. Let the FPGA architecture consist of one simple logic block defined in Figure 4.2 and 8 I/O logic blocks. In this example, we will use the I/O logic block shown in Figure 4.6. This logic block can operate as either an input pad or as an output pad.

Figure 4.7 shows an example of a packing solution. There are 10 logic blocks $\{b_1, b_2, ..., b_10\}$ total. The I/O logic blocks are set to the modes in which they are operating. The shaded primitives show which netlist atom was mapped to which primitive. The bolded edges show the physical interconnect edges that are used by nets. Figure 4.8 shows the hierarchy of the mapped logic block. Shaded regions show the used regions of the hierarchy. Note that although the typical operation of a LUT is to implement logic, a LUT can also be configured to act as an interconnect wire. As a result, a LUT is modelled as a subcluster with two modes of operation. One mode as a LUT primitive and the other as interconnect only.

4.1.5 Subproblems in Packing

From our new definition of packing, a few related subproblems arise. We have identified them as follows:

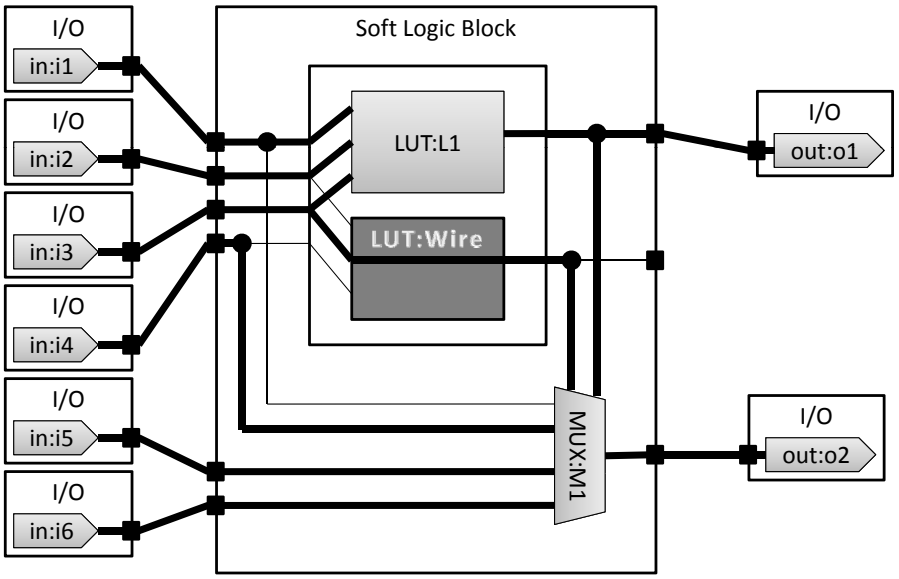

Figure 4.7: Example of a packing solution mapping a netlist to a set of logic blocks.

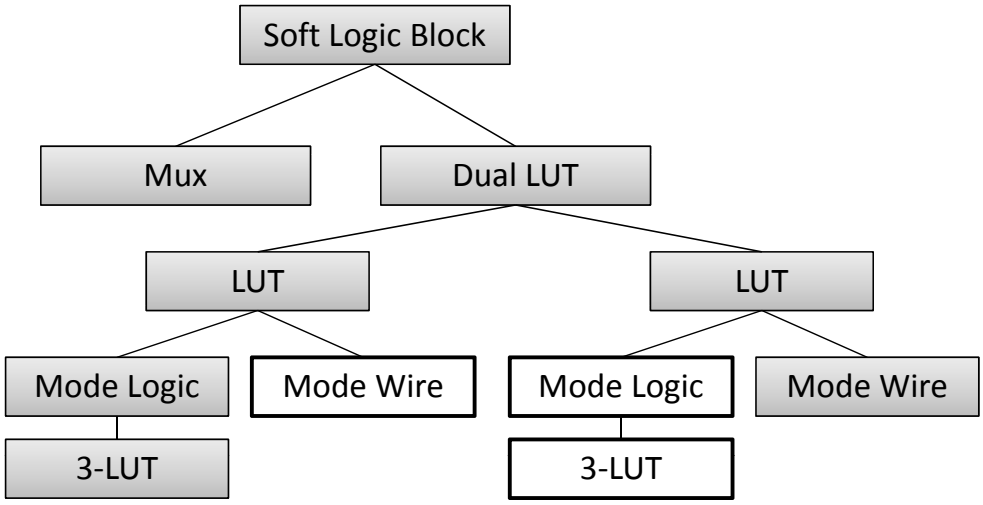

Figure 4.8: Soft logic block hierarchy for the packed solution of Figure 4.7. Shaded regions show used sections of the hierarchy.

- Partitioning: The packer needs to determine what atoms should be grouped together. This is a classical problem that all prior FPGA packers consider but now must be reconsidered in the context of heterogeneous cluster and primitive types as well as constraints, such as interconnect and modes.

- Resource balancing: When more than one type of logic block can implement (ie. contain primitives that have the same logic model) one particular atom, then the packing algorithm must make a choice when selecting what type of logic block to use. The packer should choose in such a way that the final packed solution fits a given architecture (if the size of the architecture is given) or minimizes the size of the final architecture (if the size is left variable). The resource balancing problem is often seen in commercial architectures having memory blocks of different sizes or flip-flops available in different logic blocks. We note that there has been some prior work on resource balancing for FPGAs with memories [94] and for FPGAs with different LUT sizes [29]. But, the more general problem of balancing supply and demand given different ratios of physical and netlist blocks is not well explored in packing. We do not address this issue well in this thesis. It is left for future work.

- Intra-logic block placement: In much prior academic work, the logic blocks are architected such that the selection of which particular primitive an atom maps to inside a logic block did not matter, because there was full connectivity between all internal inputs and outputs. Our new packing problem may target architectures where this assumption no longer holds (because of more limited internal connectivity within the logic block) so the packer must now select specific primitives for atoms.

- Intra-logic block routing: Since the interconnect may now be arbitrarily defined, routability needs to be checked in order to ensure a legal solution. There is little prior work on intra-logic block routing.

4.2 Architecture-Aware Packing Algorithm

This section describes an algorithm, called the Architecture-Aware Packing (AAPack) algorithm, that is a step towards solving the problem defined in the previous section.

An ideal packer would discover the globally optimal solution, to any architecture, for any input netlist, instantly. Reality dictates that one or more of these four ideals needs to be relaxed. In our particular case, we want our packing algorithm to be useful to other researchers, which in turn led us to relax all four ideals in specific ways. We propose an algorithm that can accept any architecture and any netlist as input, but we focus on the quality of results for specific sub-classes of architectures. The types of architectures we focus on are the EArch architecture described in Section 3.2.2 and the many simpler variations of this architecture described in Section 3.2.3. We chose these architectures because they contain useful features found in modern FPGAs, such as carry chains, fracturable LUTs, and configurable memories, that are often ignored in much academic work and we believe should be considered in future studies.

There are certain advantages that arise from a packer that can accept any architecture but is only vetted for some. First, the architectural space that is well-supported by the algorithm can be immediately useful. Second, for architectures that are not well supported, this algorithm serves as a starting point,

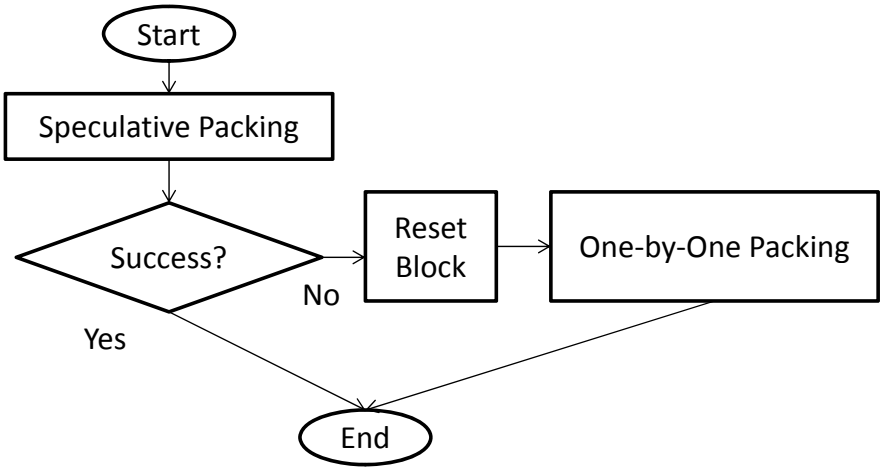

Figure 4.9: Flow chart showing the operation of the try_fill function.

for it takes less development effort to tune and optimize an existing packer that can already model the target architecture, than it is to design a packer entirely from scratch.

This section begins with an overview of the algorithm, followed by details on the main stages, some key optimizations, and then experiments and results. We will be presenting many tuning parameters in the presentation of this algorithm. Unless otherwise specified, results for each tuning parameter are obtained from mapping the VTR benchmarks to the EArch architecture using the VTR CAD flow described the in previous chapter.

4.2.1 Overview

The AAPack algorithm follows a greedy approach, using T-VPack as a base [64]. The inputs to the algorithm are the technology mapped netlist and the FPGA architecture. First, a pre-packing stage identifies certain patterns of netlist atoms that require special handling (details described later in Section 4.3.3), then a greedy general packing stage assigns atoms to logic blocks. The pseudocode for the general packing stage is as follows:

```
1: pack(atom_netlist, architecture) {
2:   logic_block_netlist = empty
3:   while(exist_unpacked_candidates(atom_netlist)) {
4:     current_logic_block =
         open_new_logic_block(atom_netlist, architecture)
5:     try_fill(current_logic_block, atom_netlist, architecture)
6:     logic_block_netlist.append(current_logic_block)
7:   }
8:   return logic_block_netlist
9: }
```

Logic blocks are filled iteratively. Line 4 shows the start of a new iteration where an empty logic block is "opened". The logic block is then filled with atoms. When the logic block cannot be further filled, the logic block is then "closed", which makes the configuration of the logic block immutable. This filled logic block is then added to the netlist of packed logic blocks on line 6. The process repeats until all atoms are packed into logic blocks, after which packing is complete, so the logic block netlist is returned.

Figure 4.9 shows the operation of the try_fill function that fills a logic block with netlist atoms. Section 4.3.1 describes the speculative packing optimization in detail. We focus first on the base operation of one-by-one packing. Pseudocode on the operation of one-by-one packing is shown below:

```
1: one_by_one_packing(current_logic_block, netlist, architecture) {
2:   candidate =
      get_next_candidate(atom_netlist, current_logic_block)
3:   try_lb_place_and_route(candidate, current_logic_block, success)
4:   if(success) {
5:     add_candidate_to_logic_block(candidate,current_logic_block)
6:   }
7: }
```

A logic block is filled by selecting a candidate unpacked atom from the netlist, locating a primitive for the candidate in the logic block, then routing within the logic block to connect the nets of the partially filled logic block. We now describe in detail the key parts of the algorithm starting with new logic block selection.

4.2.2 New Logic Block Selection

The process of filling a logic block must first begin with the selection of an unused, empty logic block. We call this *opening* a logic block. A general principle in greedy packing is to start with the most timing critical and the hard-to-pack parts of the netlist first [64]. For example, consider a long carry chain, if LUTs and flip-flops connected to a chain are packed first, then those atoms might get packed in a way that prevents the carry chain from going into the same logic block; however, if the chain is packed first, then it becomes much easier to figure out which other atoms should pack with the chain and where precisely in the logic block those atoms should go. Hence, a logic block is opened by first scoring unpacked atoms, selecting the unpacked atom with the highest score in the netlist, then returning the first logic block in the architecture that can legally pack that atom. We call the selected atom the *seed* and the score the *seed value*.

Equation 4.1 shows the equation for determining seed value.

$$seed\_value = seed\_fac * timing\_criticality + (1 - seed\_fac) * used\_inputs \tag{4.1}$$

The seed value is a weighted sum of timing criticality and the number of used inputs of the atom (normalized to the maximum number inputs of all atoms). The parameter $seed\_fac$ is set to 0.5. Timing criticality is the max criticality of all pins. We approximate atoms with many used inputs as being more difficult to pack. The seed values themselves are precomputed for all atoms before packing, then sorted for fast access.

The issue of resource balancing arises when a seed atom can pack to different types of logic blocks. If left unconstrained, our first-fit heuristic for selecting a logic block will select the same logic block type every time. To address this, we implemented a simple resource balancing method based on resizing the FPGA. At a specific FPGA size, the number of instances of each logic block type is fixed. When all instances of a particular type are used, then the first-fit heuristic will select an instance of the other types. If no unused instances exist, then we grow the FPGA to make available more instances. Thus, by starting with a min-sized FPGA, we acheive some rudimentary resource balancing. We leave a more elegant method of resource balancing to future work.

4.2.3 Candidate Selection for a Partially Filled Logic Block

After seed selection, candidate selection determines which unpacked atom the packer should try to put into a partially filled logic block. Candidate atoms are classified into two categories. The first category are unpacked atoms that have *strong* connectivity with the logic block. We define strong connectivity to mean that the unpacked atom has at least one shared, low fanout, net with a packed atom within the open logic block. A low fanout net is a net with fewer than 256 connections. For the purposes of strong connectivity, nets with more connections, *high fanout nets*, are ignored, because considering high fanout nets greatly increases the number of potential candidates, which in turn adversely affects runtime without helping quality of results. The second category consist of atoms that have weak or no connectivity with the open logic block. This category consists of atoms related to the open logic block by high fanout nets or other notions of relatedness. Since the number of atoms in this second category is usually very large, the packer always considers atoms with strong connectivity first.

Unlike conventional packing that assumes full connectivity within a logic block, the computational cost required to guarantee, in the general case, that an unpacked atom can legally pack into a logic block is large. Thus, candidate selection only applies fast, basic, legality checks. The checks are as follows:

- The candidate has not been packed.
- There exists at least one available primitive that can implement the candidate, ignoring connectivity constraints with already packed atoms.

Candidates that pass these checks are placed in a list sorted based on an *attraction* value computed for each candidate to the logic block. The candidate with the strongest attraction is selected first. If the later intra-logic block place and route stages cannot find a legal fit for this candidate, then the next candidate in the list is selected. The number of candidates in the priority queue of strongly connected atoms is kept constant to keep runtime reasonable. For EArch, a queue length of 1000 doubles pack time compared to a queue length of 2 in return for 3% better logic density and 2% fewer external nets. We set the queue length to 30 as this provides almost the same quality as a queue length of 1000 for only a 15% increase in runtime when compared to a queue length of 2.

The attraction function is a weighted sum of the timing gain (an estimate of the effect of this choice on the critical path) and the connection gain (an estimate of how many connections will be absorbed by this choice). The attraction function is based off [64] and is defined as follows:

$$attraction(atom) = (T * timing\_gain + (1 - T) * connection\_gain) \tag{4.2}$$

$$timing\_gain = \max(timing\_criticality) \tag{4.3}$$

$$timing\_criticality_{edge\_i} = 1 - slack_{edge\_i}/crit\_path\_delay \tag{4.4}$$

$$slack = required\_time - arrival\_time - node\_delay \tag{4.5}$$

Timing gain of an atom is a normalized value from 0 to 1 and is equal to the timing criticality of the most critical edge connecting that atom to the currently open logic block. A value of 1 means that the atom has at least one critical edge with no *timing slack*, where timing slack is defined as the total delay, normalized to critical path delay, that may be added to a pin before that pin becomes critical. A value close to 0 means that all the pins of the atom have plenty of slack. Timing analysis for packing is

described earlier in Section 3.3.3. T is a constant, determined empirically, with a default value of 0.75. Connection gain is defined below:

$$connection\_gain = \frac{AB * net\_absorption\_gain + (1 - AB) * num\_shared\_nets}{num\_used\_pins\_of\_candidate} \tag{4.6}$$

The connection gain metric for an atom is similar to my Master's thesis [56], which in turn is a simplification of the much more complex attraction functions of [12]. It is a weighted sum of *net absorption gain* and number of shared nets. Both these values are normalized to the number of used pins of the candidate. The number of shared nets is a count of the total number of nets shared between the atom and the current set of atoms packed in the open logic block. Net absorption gain is a sum of $1/(number\_pins\_of\_net\_outside\_open\_logic\_block)$ over all shared nets. The idea behind net absorption gain is that atoms connected to a logic block by lower fanout nets should have higher attraction than atoms connected a logic block by higher fanout nets. AB is a constant that is tuned, through empirical tests, to 0.9.

When the queue of strongly connected candidates is exhausted before the logic block is completely filled, which happens more regularly for heterogeneous architectures, then atoms with weak (or no) connectivity to the logic block need to be considered. We select these candidates using the following techniques (in order of priority): 1) shared connections with high fanout nets, 2) transitive connections, and finally 3) just based on number of inputs.

Shared Connections with High Fanout Nets

Once a logic block is opened for packing and atoms are packed into it, a record is kept of the lowest-fanout high fanout net (defined as nets with 256 or more connections). When no unpacked atoms with strong connectivity to a logic block remain, the atoms connected to this high fanout net are then selected from. This is particularly important for packing memories, where composing a very wide memory from 1-bit data memory slices requires considering high fanout address or control signals.

If no legal candidate among this set can be found, then the next selection strategy, *transitive connections*, are considered.

Transitive Connections

Figure 4.10 illustrates an example of transitive connections. The multiplier in this figure has registered inputs but this particular multiplier logic block does not supply flip-flops so those flip-flops must be packed in a different logic block. Those flip-flops do not directly connect to each other but rather are related indirectly because of their shared connectivity with the multiplier – the flip-flops are *transitively connected*. Thus, it makes sense to pack these flip-flops together over completely unrelated logic. We consider unpacked atoms as transitively connected with the current logic block if those atoms share connections to already packed logic blocks that in turn have connections with the current logic block. The more transitive connections an unpacked atom has, the higher the attraction value for that atom. To prevent too many candidates from being explored, the number is limited to transitive connections for nets that have less than four sinks. If this fails to find a candidate, then completely unrelated atoms are considered.

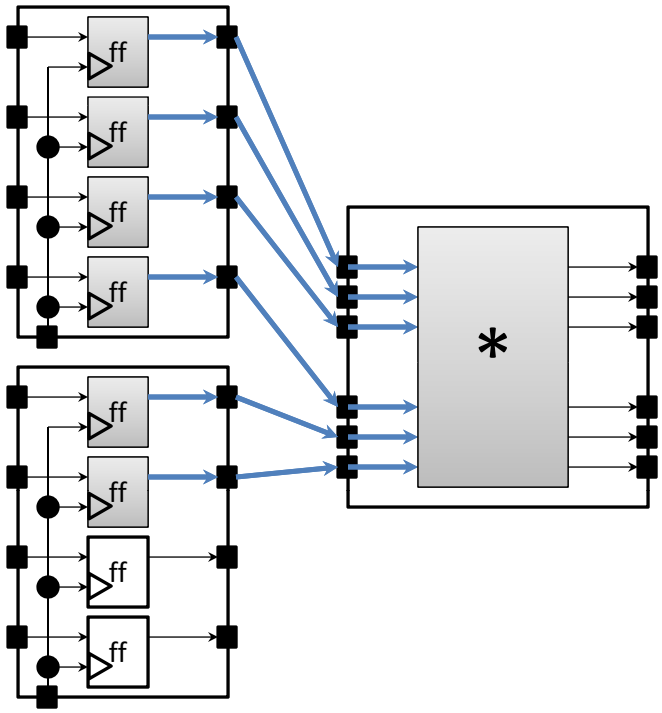

Figure 4.10: Registered multiplier inputs show the importance of transitive connections during packing.

Unrelated Logic

The final technique selects an unpacked atom with the largest number of inputs as the best candidate, provided some simple feasibility tests pass. These feasibility tests are checking that an available primitive exists that can implement that atom (ignoring connectivity) and that there exists enough free pins on the logic block to pack the atom. This technique of packing unrelated logic is ported over from the original T-VPack [64]

4.2.4 Intra-Logic Block Placement

During packing, an intra-logic block placement subproblem arises because, given a candidate atom, there may exist many primitives in the open logic block that can implement that atom, and the choice of which primitive to assign that atom matters. Consider, for example, the fracturable LUT described in Section 2.1.1, the figure of which is reproduced again in Figure 4.11 for the reader's convenience. It is illegal to assign two completely independent 5-LUTs in this one fracturable LUT because the fracturable LUT can accomodate at most 8 unique inputs and not 10. Much prior work in packing avoided the intra-logic block placement subproblem by simply not supporting architectures where intra-logic block placement is an issue. However, with the need to explore more complex architectures, this subproblem can no longer be ignored.

The intra-logic block placement subproblem is as follows: Given a logic block and all atoms to pack into that logic block, legally assign the atoms to primitives in the logic block such that the overall objective function is optimized. Since the overall packing algorithm we employ is greedy and iterative, we simplify the intra-logic block placement subproblem as follows: Given an already partially filled logic

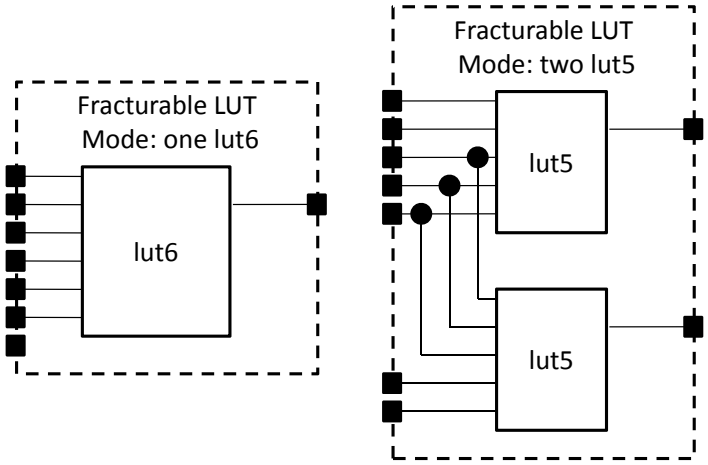

Figure 4.11: A fracturable LUT that can operate as one 6-LUT or as two 5-LUTs that share 3 inputs.

block and a candidate atom to pack into that logic block, return a *potential* unused primitive such that area is minimized.

We employ a simple, greedy, best-fit (for a simple cost function) heuristic for intra-logic block placement. Every primitive in the logic block has a *cost* value and a *valid* flag. Primitives with a lower cost are considered first before primitives with a higher cost. The cost function is initially set to be equal to the number of inputs of a primitive, so that smaller primitives are considered first before larger primitives. The cost of each primitive is modified during placement to prioritize filling partially used subclusters over unfilled subclusters. More detail on this will be described later. The valid flag stores whether or not this primitive is available for use. A primitive that is used by another atom or a primitive that belongs to a conflicting mode with the current state of the logic block will have its valid flag set to *FALSE*.

Three lists keep track of state information, they are as follows:

- *Available Primitives List* - Stores valid primitives.
- *Attempted Primitives List* - Stores primitives that have been attempted for the current candidate atom but did not result in a legal assignment.
- *Invalid Primitives List* - Stores invalid primitives.

A valid primitive for a candidate atom is selected from the available primitives list. The available primitives list is organized as shown in Figure 4.12. Primitives are sorted into bins based on type. A primitive type is defined by the architect and all instances of a particular type implement the same logic model, have the same number of pins, and the same delay profile. If an atom can be implemented by a particular primitive type, then the primitive with the lowest cost of that type is selected. Since an atom may fit into multiple different types of primitives, this process is repeated for all primitive types after which the placer returns the overall lowest cost primitive. Lazy removal is used, so primitives that are flagged as invalid will be removed when encountered.

The atom is tentatively placed in the lowest cost primitive selected, then this assignment undergoes two legality checks: a pin counting check (described in Section 4.3.2) and a routability check (described in the next subsection). If the assignment fails a check, then the placement is reverted, the primitive is

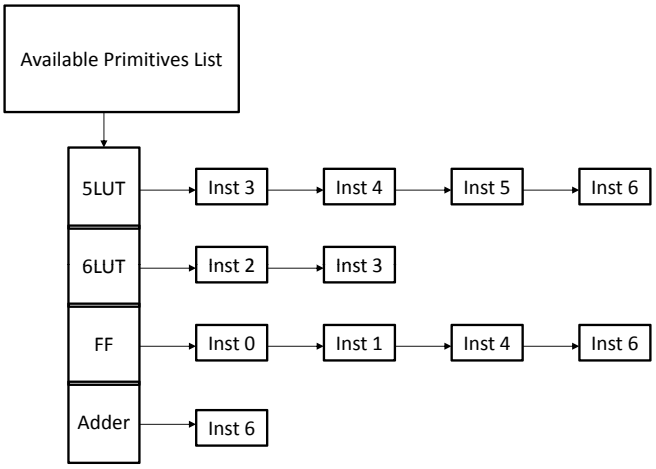

Figure 4.12: Example list of available primitives for intra-logic block placement.

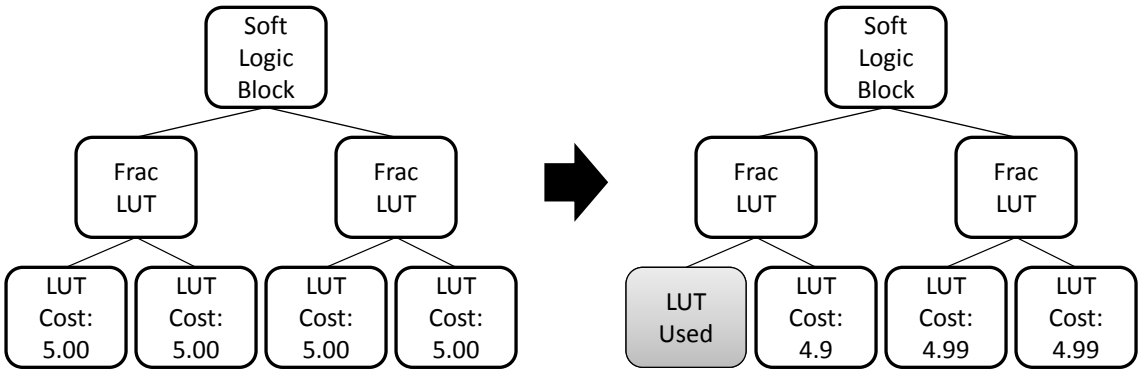

Figure 4.13: Example showing how the intra-logic block placement cost function changes, as a logic block is filled, to favour selecting primitives in used subclusters first before unused subclusters.

removed from the *Available Primitives List* and stored in the *Attempted Primitives List*. If the intra-logic block placer exhausts all potential placements, then the candidate atom is rejected and primitives in the *Attempted Primitives List* list are moved back to the *Available Primitives List*.

If the legality checks pass, then that assignment is *committed*. A commit results in a series of actions. First, primitives in the *Attempted Primitives List* list are moved back to the *Available Primitives List*. Second, the selected primitive becomes invalid as it is now used to implement the atom. Third, the modes in the logic block that must be set in order for the primitive to be used in turn invalidate primitives that belong to the other, mutually exclusive modes. For example, if a fracturable LUT is set to dual 5-LUT mode, then the 6-LUT primitive cannot be used so the 6-LUT primitive would then be invalidated. Finally, each primitive that remains valid has its cost reduced by 0.1^a, where a is the depth of the used primitive to the closest ancestor of the unused primitive, and the value 0.1 is an empirically derived parameter. This encourages the usage of partially used subclusters before unused subclusters.

Figure 4.13 shows a simple example of how primitives in partially-filled subclusters are favoured over unfilled subclusters, and how this minimizes area by encouraging tight packing. The logic block in this example contains two subclusters where each subcluster has two LUTs. When the logic block is unfilled, all LUTs have the same cost. When one LUT is used however, the cost of the unused LUT closest in

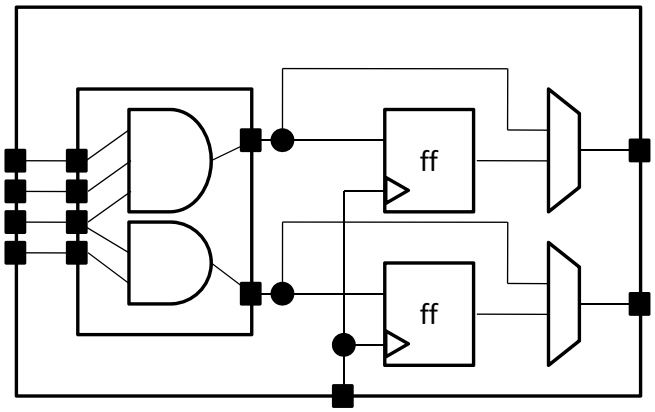

Figure 4.14: Example of a simple logic block.

the hiearchy to the used LUT is reduced more than the cost of the LUTs in the unfilled subcluster.

4.2.5 Intra-Logic Block Routing

Driven by a need to explore increasingly sophisticated logic blocks, intra-logic block interconnect is now an input to packing, which gives rise to an intra-logic block routing subproblem. The intra-logic block routing subproblem takes as input a partially filled logic block and outputs the interconnect used to implement nets within the logic block. It is part of the process to determine if the assignment of atoms to primitives, selected by the earlier placement stage, is legal. If intra-logic block routing finds a solution, then the specific assignments made by intra-logic block placement is legal. If intra-logic block routing fails, then the placement is rejected. This section describes how interconnect is modelled, followed by a description of the routing algorithm used.

Interconnect Model

The interconnect supplied within a logic block is modelled as a directed graph of nodes and edges. Every pin of a logic block, subcluster, or primitive is modelled as a node in this graph. Primitive input pins are modelled as *sink* nodes that may be the end destination of a net connection while primitive output pins are modelled as *source* nodes which may drive a net.

In packing, external-to-logic block interconnect is simplified to a full crossbar that connects all logic blocks together. To represent this model of external interconnect, the intra-logic block interconnect graph adds both an *ext_source* node to represent signals that come from outside the logic block and an *ext_sink* node to represent signals that go out to other logic blocks. The ext_source and ext_sink nodes may accommodate as many nets as there are logic block inputs and outputs, respectively. The ext_sink connects to all logic block inputs to model the use of external interconnect in the event intra-logic block interconnect is insufficient for routing signals.

Figure 4.14 shows an example of a logic block that contains a 3-input AND gate, a 2-input AND gate, and two flip-flops. This logic block has some statically-controlled muxes that specify whether to use the flip-flop or AND gate outputs. Observe there is some pin sharing for the AND gates. Figure 4.15 shows how the interconnect of this logic block is modelled. In this figure, source nodes are represented by empty circles, sink nodes by empty squares, and other pins are represented as filled squares. The

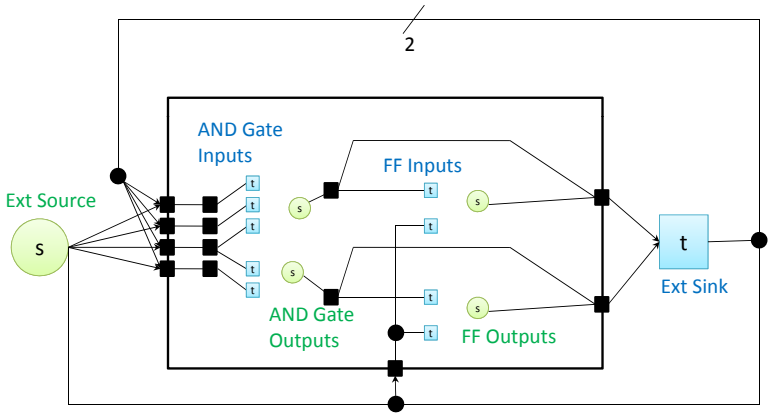

Figure 4.15: Example of the interconnect graph for the logic block described in Figure 4.14.

ext_sink node is represented by a large square with a 't', and the ext_source node is represented by a large circle with an 's'. Notice that the link from the ext_sink node to the input pins of the logic block represents external interconnect which must be used if the flip-flop drives an AND-gate in the same logic block.

Routing Algorithm

The intra-logic block router employs a non-timing-driven variant of the PathFinder negotiated congestion routing algorithm [65]. Nets are routed using a minimum cost search while allowing for overuse. Overuse is gradually removed by iteratively re-routing nets and increasing the cost of congestion after each *routing iteration*. A routing iteration finishes when all nets have be visited once. Routing completes when there are no overused nodes, in which case a valid routing solution is returned, or when the number of routing iterations exceed a threshold, in which case the router failed to find a legal answer.

The cost of a node n is determined by the following equation:

$cost_n = (b_n + h_n) * p_n$

The term b_n represents the base cost of a node. This cost depends on the node and net being considered; it is independant of the routing iteration. The term p_n represents the present congestion of a node. If the number of nets currently using this node is less than the maximum *capacity* supplied by the node, then p_n is set to 1 because using that node does not create congestion. Otherwise, p_n is set to $(1 + NetsUsingNode - Capacity) * CongestionFactor$ where $CongestionFactor$ is a term that increases with each routing iteration to increasingly penalize overused nodes. Present congestion has the most immediate effect towards removing congestion. The term h_n represents historical congestion. It is the sum total of overuse of the node over the course of the routing algorithm execution. This term serves to "coax" the router to explore alternative paths not previously chosen for heavily congested areas. Similar to p_n, the historical usage count for h_n is multiplied by a historical use factor that increases with each routing iteration.

We made two tuning optimizations to the PathFinder algorithm. First, we applied a standard technique where only *illegal* routes are re-routed – nets that are legal do not get rerouted. This dramatically speeds up the algorithm. Second, we applied some cost tuning that is specific to intra-logic block packing. Logic block input pins have a base cost set to 1,000 times more than pins inside the logic block so

that the router strongly prioritizes intra-logic block routing over using external routing. We also set the base cost of a node to nominally 1 but adjusted by a small fanout factor. For interconnect nodes with a fanout of 1, the fanout factor penalizes nets with higher fanout, while favouring nets with a fanout of 1. For interconnect nodes with a fanout of 2 or more, the fanout factor penalizes nets with a fanout of 1 while favouring nets with a higher fanout. This biases the intra-logic block router to using shared pins for multi-fanout nets during earlier routing iterations rather than waiting for negotiated congestion to discover pin sharing. The equation for the fanout factor is as follows:

$$fanout\_factor = \begin{cases} 0.85 + \frac{0.25}{net\_fanout}, & \text{if } node\_fanout \geq 2 \\ 1.15 - \frac{0.25}{net\_fanout} & \text{otherwise} \end{cases} \tag{4.7}$$

For architectures with depopulated crossbars and fracturable LUTs, only re-routing congested nets speeds up pack time by 26%, while fanout factor speeds up pack time by an additional 8%. For simpler architectures, both these techniques have limited, if any, effect. We speculate that for even more complex intra-logic block interconnect structures than those investigated here, more sophisticated optimizations may be needed.

4.3 Interconnect-Aware Enhancements to Packing

The VTR logic block architectural description language (described in [58] and in Section 3.2.1) provides the architect great freedom when specifying logic block architectures, including the ability to specify any arbitrary interconnect structure within a logic block. Support for arbitrary interconnect enables the natural expression of a wide range of architectural constructs. These include carry chains, crossbars, optionally registered inputs/outputs, and control signals, which can be expressed by simply stating how various components are connected together. However, this level of customization creates a computationally challenging packing problem. The packing algorithm must determine if the internal connectivity within a logic block can successfully route the portions of the netlist that are assigned to that logic block. The algorithm described in the previous section deals with this problem by solving the full routing problem. However, though robust, detailed routing alone can be very slow. This section describes techniques to avoid the blanket use of detailed routing when "lighter weight" approaches may work.

Our vision is a packing tool and algorithm that runs quickly for architectures with simple interconnect, spends medium computational effort on architectures with moderately complex interconnect, and only uses heavy computational effort on architectures with very complex interconnect. Our approach is to use a faster, simpler algorithm when interconnect structures that are easier to deal with are encountered. For example, if an architecture contains full crossbars, then computationally intensive routing checks within the logic block are not necessary because routing is guaranteed as long as the number of pins to be connected is below a certain threshold. Similarly, if an architecture has an inflexible carry chain, then we know that the blocks that form that chain must be kept together in a strict order.

We introduce three techniques that enable the packer to adjust computational effort based on interconnect complexity [62]: First, *speculative packing* attempts to save runtime by optimistically skipping detailed legality checks at intermediate steps and then checking all legality rules after a logic block is full. Second, *interconnect-aware pin counting* reduces the more complex routing problem to a simple counting problem, which is inferred from the architecture. Third, *pre-packing* groups together netlist blocks that should stay together as one unit during packing. This helps the packer deal with interconnect

structures with limited or no flexibility, such as carry chains and registered input/output pins.

4.3.1 Speculative Packing

Speculative packing is a technique to avoid unnecessary invocations of detailed routing. This technique first attempts to optimistically pack a logic block without invoking detailed routing until the logic block is filled. We call the optimistically filled logic block the *speculated solution*. If detailed routing of the speculated solution succeeds, then the solution is accepted. Otherwise, the packer rejects the speculated solution and reverts back to the conservative method described in Section 4.2 that invokes detailed routing for every partial packing for that logic block.

The runtime impact of speculative packing depends heavily on how often the final route of a speculated solution succeeds. In the best case, the final route always succeeds, resulting in speedup. In the worst case, the final route never succeeds, resulting in wasted speculation time. Thus, if a logic block contains simple interconnect from which the packer can form routable speculated solutions, then speculative packing enables the packer to spend less computational effort routing. If a logic block contains more complex interconnect, then the computational effort expended by the packer depends on how often the packer assembles a routable speculated solution – the more often a routable speculated solution is found, the less computational effort expended.

4.3.2 Interconnect-Aware Pin Counting

As discussed above, there are some circumstances in which the routing problem can be abstracted into a much simpler counting problem - for example, when logic blocks have a full internal crossbar for routing. Pin counting is a technique that approximates the routability problem with a simpler counting problem. Pin counting checks if a particular assignment of atoms to a logic block/subcluster uses more pins than supplied by the logic block/subcluster. If pins are overused, then that assignment is proven unroutable. If pins are not overused, then in the pin counting approach, we optimistically assume that the assignment is routable. Pin counting is a check performed during intra-logic block placement. During speculative packing, when detailed routing is skipped, pin counting becomes the only check for routability. Therefore, more accurate pin counting reduces computational effort by increasing the chance that speculated solutions will route.

Interconnect-aware pin counting is a more precise implementation of pin counting than what was done during my Master's work [56]. In addition to analyzing pins, interconnect-aware pin counting also analyzes the underlying physical interconnect with the intention of capturing clues about how those pins are related. We begin by describing what information this technique extracts from the interconnect, and then we describe how packing uses that information.

Prior to the packing stage, we analyze the architecture of each logic block, and group the pins of each block and subcluster into separate *pin classes* based on the interconnect structures. Intuitively, pin classes are an attempt to approximate arbitrary interconnect with a set of non-overlapping full crossbars. Input pins of the same class drive the same crossbar. Output pins of the same class are driven by the same crossbar.

Two pins of a subcluster/logic block belong to the same pin class if they belong to the same, connected, interconnect graph within the subcluster/logic block and satisfy two constraints. First, pins in the same pin class must be on the periphery of the same subcluster/logic block. Second, pins of the same

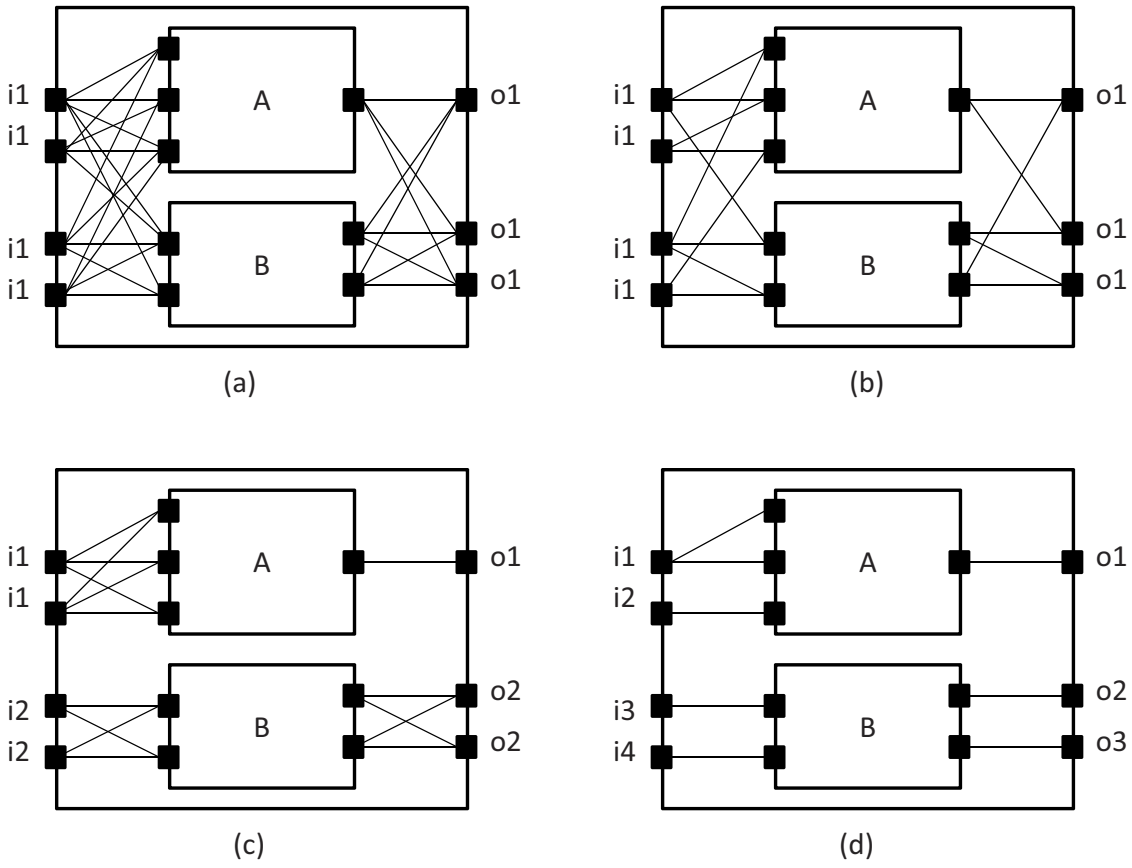

Figure 4.16: Examples on how pins are grouped into pin classes. Input pin classes are labelled with an i followed by a number while output pin classes are labelled with an o followed by a number.

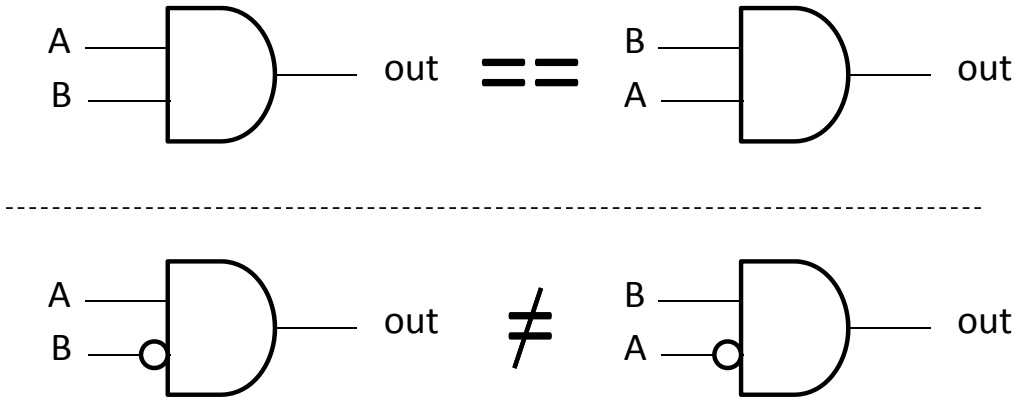

Figure 4.17: An AND gate is logically equivalent because its inputs can be swapped without changing functionality. The gate below is not logically equivalent because functionality changes if the inputs are swapped.

pin class must either be all input pins or all output pins. Note that although primitives are involved in the determination of pin classes, the primitives themselves do not have pin classes because primitives do not contain interconnect and are terminal points for nets.

Figure 4.16 illustrates different examples of pin classes on a subcluster with two primitives, four input pins, and three output pins. The labels on the subcluster pins show which pin class each pin belongs to. Figure 4.16 (a) has a large, well populated crossbar at the inputs and outputs. The subcluster input pins all belong to the same pin class *i1* and the subcluster output pins all belong to another pin class *o1*. Figure 4.16 (b) has a sparser crossbar than (a). Our technique optimistically approximates these cases as the same, thus (b) has the same pin classes as (a). Figure 4.16 (c) has disconnected smaller crossbars. This is reflected in the two pin classes for the inputs and two pin classes for the outputs. Finally, Figure 4.16 (d) has no interconnect flexibility so all subcluster pins belong to separate pin classes.

We include a special case in the event that the primitive has logically equivalent pins. Pins that are logically equivalent means that connections to those pins may be swapped arbitrarily without changing the functionality of the primitive. Figure 4.17 show that an AND gate has logically equivalent inputs while the the gate below it does not. Logically equivalent primitive pins are considered as one pin for the purposes of determining pin classes. Currently, logical equivalence is only supported for LUTs but may be easily extended for other gates too.

Prior to packing, pin classes are determined for each logic block type and subclusters within. We begin by selecting an unclassified input pin of the logic block. Starting from that pin, the interconnect graph is traversed forwards and backwards, ignoring directionality, until all reacheable nodes have been visited once. If any input pins are encountered, then those pins belong to the same pin class. Afterwards, the next unclassified input pin is selected. This is repeated until all input pins belong to pin classes, then repeated again for output pins. Once all pins are classified for the logic block, the process is repeated for each subcluster inside the logic block. To provide fast pin class lookup during packing, a lookup table that maps each primitive pin to pin class is created for each logic block type during the construction of pin classes. Similarly, a lookup table records whether or not a primitive output pin can connect to a primitive input pin using intra-logic block interconnect for each logic block type/subcluster type.

During packing, every time a candidate atom is placed inside a logic block, pin counting updates the

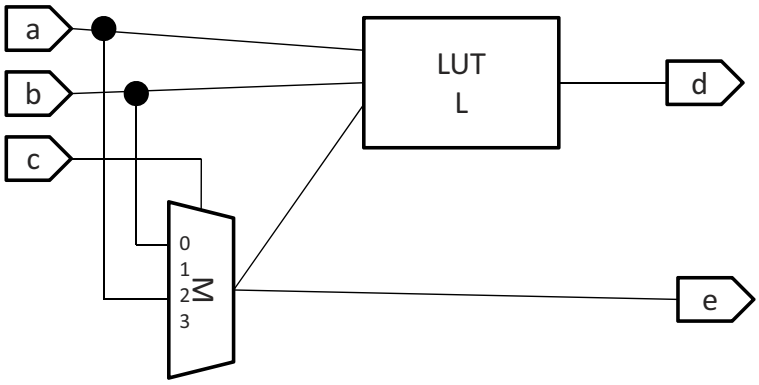

Figure 4.18: Example netlist to illustrate pin counting.

utilization of the pin classes of all used subclusters within the logic block and then updates the utilization of pin classes of the logic block itself. If, after the update, there exists a pin class that uses more pins than is supplied by that pin class, then pin counting declares the intermediate solution unroutable. Without loss of generality, we describe the update procedure for just the logic block. A net adds a count of one to a pin class of input pins if and only if the net drives a primitive input pin through that pin class and the driver of that net cannot reach the primitive input pin solely from within the logic block. A net adds a count of one to a pin class of output pins if and only if the primitive output pin of that net drives that pin class and there exists one primitive input pin driven by the net that cannot be reached solely from within the logic block.

To illustrate the nuances of pin counting, we revisit the logic block described in Figure 4.2. We show the effect of packing the netlist in Figure 4.18 to that logic block. This netlist consists of a LUT L and a logical mux M. Figure 4.19 shows an intermediate packing solution that placed the LUT L in the top LUT position and the mux M in the mux location. We start by describing the pin classes in the logic block, then we describe the utilization of each pin class.

The logic block input pin classes are I1, I2, and I3. The logic block output pin classes are O1 and O2. The dual-LUT subcluster input pin class is SI1 and the subcluster output pin class is SO1. There is some subtlety in determining pin class I1. All four top input pins of the logic block belong to the same pin class. A LUT has logically equivalent inputs so the top three input pins are grouped together and the second, third, and fourth input pins are grouped together. Moreover, since the second and third input pins are common to both groups, all four pins are merged into the same pin class. The capacity of each pin class is determined by the number of pins it grouped. We label this value in the figure as the denominator of the fraction displayed beside each pin class. For pin class I1, the capacity is 4.

The utilization of each pin class is determined by the nets connected to the primitives within the logic block. This value is displayed as the numerator in the fraction beside each pin class in the figure. Observe the following sublety: Net a requires two logic block input pins because of the lack of internal flexibility in the logic block. This behaviour is captured by consuming one pin each in pin classes I1 and I2. This is in contrast to net b which only needs to consume one pin in pin class I1 because of internal fanout within the logic block. Net M from the logical mux must traverse outside the logic block to reach the LUT input. This is represented as consuming one count of pin class O2 and one count of pin class I1. Net c illustrates how interconnect-aware pin counting is optimistic. Without the ability to

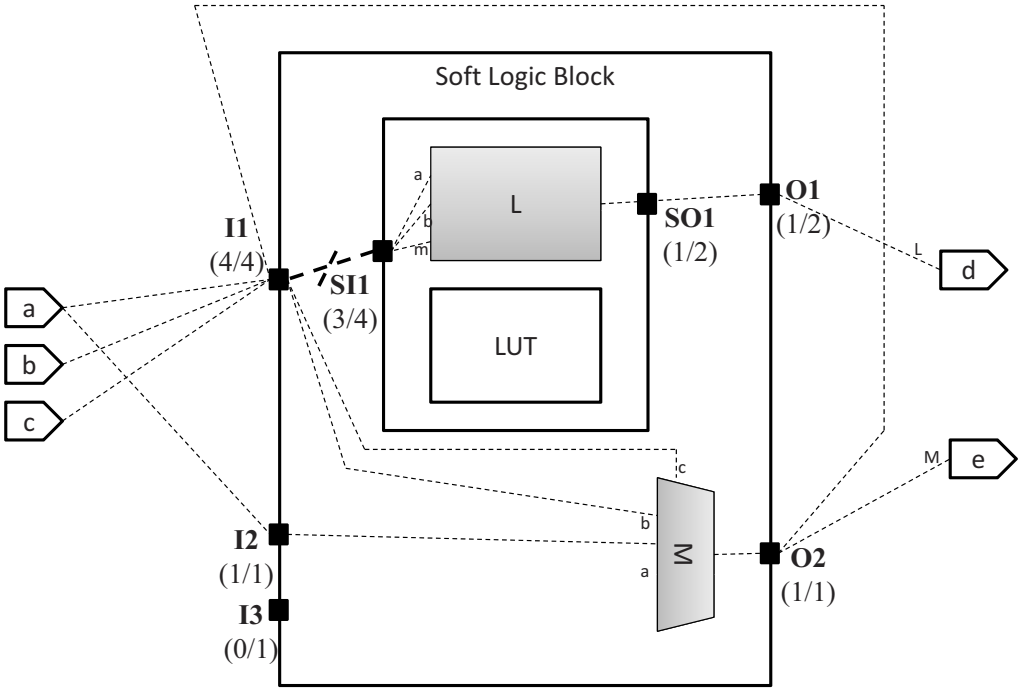

Figure 4.19: Example intermediate solution to illustrate pin counting. The pins in this figure are already grouped into pin classes. The utilization and supply of each pin class are shown below the pin class name.

detect that it is necessary to route through the dual-LUT subcluster to reach the mux select line, we see that our pin counting technique optimistically uses 3 of 4 pins in SI1, when in fact all 4 must be used in a detailed route. These examples illustrate which properties interconnect-aware pin can capture and which it cannot capture.

To summarize, we list the properties and limitations of interconnect-aware pin classes as follows:

- Acts as an optimistic filter. Cases that fail interconnect-aware pin counting will fail to route, while cases that pass may or may not successfully route.
- Sparse interconnect is approximated as fully flexible.
- Does not account for situations where a net routes through a subcluster without connecting to any primitive within the subcluster.
- Internal feedback and feedfoward connections within a logic block/subcluster can reduce the usage of a pin class and therefore must be checked.
- Only returns pass/fail. Does not give hints to guide future candidate selection.

4.3.3 Pre-Packing

Logic blocks sometimes contain inflexible routing structures. These structures can cause complications in a greedy packer because different stages of the packer become necessarily coupled. We illustrate this coupling using carry chains as an example. A carry chain is an important structure that enables the fast

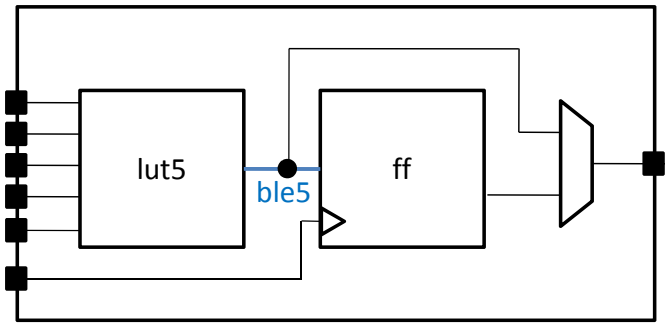

Figure 4.20: The pack pattern label, *ble5*, on the link that joins the LUT and flip-flop primitives in the architecture description file, forces LUT and flip-flop atom pairs to treated as a single unsplittable unit during packing.

computation of wide netlist adders by chaining together smaller physical adders using fast, inflexible carry links. In the packing stage of the VTR CAD flow, a netlist adder is represented by multiple smaller adder atoms that link together to implement wider addition. The packer must map the adder atoms to physical adder primitives in such a way that the physical chain can implement those logical links. An incorrect grouping or placement of the atoms during packing can result in failed (internal-to-the-block) routing because carry connections may become impossible to route. This example shows how inflexibility in interconnect can cause strong coupling among the candidate selection and placement stages in packing. This coupling is not unique to carry chains. We observe this coupling effect in multiple other logic block constructs including primitives with registered inputs/outputs, datapath arithmetic blocks with compound operations such as multply-add, and others.

We employ a pre-packing technique to capture coupling from restrictive interconnect in a generic and simple way. The architect is asked to identify (in the architecture file) groups of primitives joined together using inflexible interconnect by annotating the links in the interconnect. These groups and their links are called *pack patterns*. Figure 4.20 shows an example of a pack pattern for a BLE which consists of a LUT and flip-flop pair. The code below shows how a direct link is annotated with the pack pattern in the architecture description:

```
<direct name="direct2" input="lut5.out" output="ff.D">
  <pack_pattern name="ble5" in_port="lut5.out" out_port="ff.D"/>
</direct>
```

Before packing, groups of netlist atoms that match a pack pattern are grouped together into what is called a *molecule*. We call this stage the *pre-packing* stage. During packing, molecules are treated as though they are an atom and can only map to primitives that form the same pack pattern as the molecule.

Figure 4.21 shows a second example of the concept of pack patterns and molecules. This arithmetic logic block can perform both basic multiplication and addition, as well as combined operations such as multiply-add and registered arithmetic. If the architect intends for combined operations to be kept together during packing, then the architect should indicate that intent by specifying four pack patterns as follows: 1) Multiply-add, 2) Registered multiply, 3) Registered add, and 4) Registered multiply-add.

Certain pack patterns, such as a LUT or a multiplier with registered output(s), speed up packing and help with quality of results; however, other pack patterns, such as carry chains, are necessary in

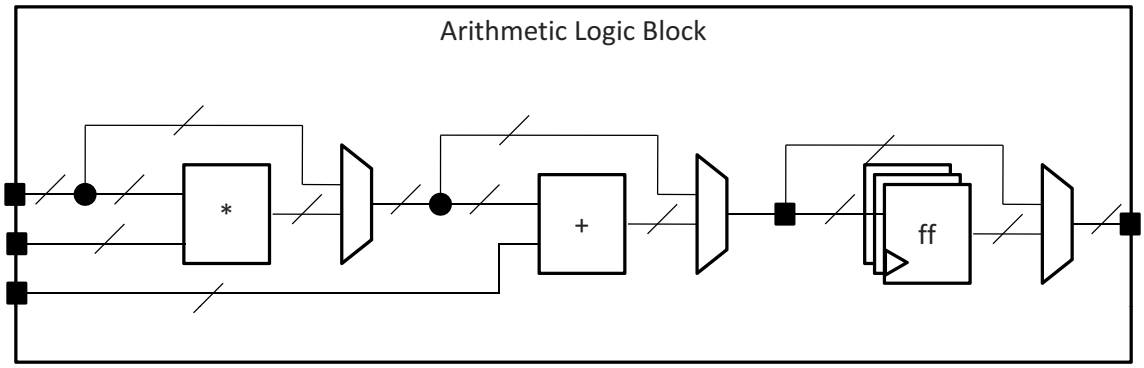

Figure 4.21: A bus-based arithmetic logic block.

order for the packer to find a legal solution. The necessity of a particular pack pattern is determined by whether or not one-by-one packing of atoms produces legal intermediate solutions. For carry chains, this is not true because of the lack of interconnect flexibility on the carry links, hence pack patterns is necessary for them.

Molecules have an impact on the attraction gain used to determine what atoms should be packed together. The attraction gain for a molecule is the sum of the gains of each atom in the molecule modified by two tie breakers. The first tie breaker penalizes the introduction of new input nets by a cost of 0.001 per net. The second tie breaker biases the packer to select molecules with more atoms by adding 0.0001 per atom.

We now describe the pre-packing algorithm used to group atoms into molecules. The netlist is traversed, once per pack pattern, starting with the largest pack pattern and ending with the smallest pack pattern. For each traversal, the pre-packer matches parts of the netlist to a pack pattern through a technique similar to the matching process in standard cell technology mappers [41]. A *root* primitive is selected in the pack pattern. If the root primitive can implement the current netlist atom selected, then another primitive connected to the root and belonging to the pack pattern is selected. The corresponding atom that matches that next primitive is located (if exists). This process is repeated until all primitives in the pack pattern have a corresponding match with an atom in the netlist, in which case those atoms are grouped together into a molecule. Or, if at any point, there is a mismatch (such as when no corresponding atom exists for a particular primitive), then the candidate atom cannot form a molecule from that root reference point and the next candidate is selected. The molecule creation process is greedy so atoms that are assigned to a molecule cannot be reassigned later to another molecule.

Carry chains create a special case for pre-packing. A carry chain has dedicated links that can extend across multiple logic blocks. So unlike other molecules, the pre-packer will assemble a carry chain by grouping a single chain of atoms into multiple molecules. The size of each molecule matches the length of the chain in a single logic block except the last molecule in the chain, which may have a variable size. This implies that, for wide addition that extends beyond one logic block, the carry chain head must always start at the beginning of the physical chain in the logic block. The code that handles carry chains is general to chain-like structures, such as linear mux chains, are also supported.

4.4 Experiments and Results

This section describes the experiments and results for the AAPack packing algorithm. The first set of experiments quantitatively break down the impact of key parts of the AAPack algorithm. Then, we measure how the AAPack algorithm changes computational effort based on architecture complexity. Afterwards, we compare AAPack performance against other packers for simple, legacy, soft logic blocks. Finally, we analyze the runtime behaviour of the algorithm.

4.4.1 Isolating the Impact of Optimizations

The following experiments measure the effectivness of the pre-packing technique, the speculative packing technique, and the intra-logic block router. These experiments use the EArch architecture described in Section 3.2.2. This is a heterogeneous architecture with soft logic, multipliers, and memories. The soft logic blocks contain fracturable LUTs, hard adders, carry chains, and a 50% depopulated crossbar. The experiment employs the VTR benchmarks described in Section 3.1. These benchmarks are from a variety of real applications, the largest of which contains almost 100,000 6-LUTs. The VTR CAD flow employed is described in Section 3.3. The software used is revision 4310 of the publicly available trunk which has advanced substantially beyond the latest, VTR 7.0, release. The placement option *inner_num* was set from a default of 1 back to the historical value of 10 because this produced both better quality of results and faster overall experiment runtimes.

All experiment results are normalized against the full flow, with all packing optimizations turned on, as described in Section 3.4.

Pre-Packing and Speculative Packing

The following experiments investigate the pre-packing and speculative packing techniques. In these experiments, pre-packing of carry chains is always on because this is necessary for the packer to handle the adders in the soft logic block. Thus, only for this section, *pre-packing off* means that pre-packing is turned off for LUT/FF pairs only.

The first experiment investigates packing without speculation and without pre-packing. Table 4.1 shows the results. The columns from left to right are as follows: the circuit name, packer runtime, minimum channel width (Min W), critical path delay at 1.3 times min W, number of soft logic blocks, and number of *external nets* (nets that connect two or more logic blocks). All values after the leftmost column are the geometric mean across all benchmarks normalized to the baseline EArch architecture (on the CAD flow with speculation and pre-packing on).

The results show that pack time increases by about 5-fold. The number of soft logic blocks also increases 7%. The increase in soft logic blocks arises because, without pre-packing, the packing algorithm may mistakenly split LUT/flip-flop pairs which reduces packing density. To see why this happens, consider the case where a 6-LUT drives a flip-flop. Assume that the flip-flop is packed first and the placement of that flip-flop sets the FLE to dual 5-LUT mode, then the 6-LUT cannot be packed into the same FLE thus wasting space. Note that speculation is a runtime optimization, while pre-packing is both a runtime and quality of results optimization. Therefore, if we rerun the same experiment without pre-packing but with speculation, then we expect to see similar quality of results but with faster runtime.

As such, the next experiment investigates packing with speculation and without pre-packing to isolate the impact of speculation. Table 4.2 shows the results. The columns are the same as previously

Table 4.1: Quality of results of packing without speculation and without pre-packing (of LUT and FF pairs) normalized to the fully optimized baseline.

Circuit	Pack Time	Min W	Crit Path Delay	Num Soft Logic Blocks	Num Ext Nets
arm_core	4.63	1.09	1.04	1.06	1.05
bgm	2.20	1.06	0.94	1.01	1.00
blob_merge	5.44	1.00	1.03	1.03	1.01
boundtop	7.90	0.97	1.05	1.11	1.05
ch_intrinsics	6.19	1.00	0.98	1.08	0.99
diffeq1	6.75	0.86	1.06	1.12	1.03
diffeq2	6.85	0.93	1.03	1.07	1.00
LU8PEEng	2.48	1.03	1.04	1.02	1.00
LU32PEEng	2.32	1.05	0.95	1.02	1.00
mcml	3.68	0.93	1.03	1.19	1.03
mkDelayWorker32B	6.38	1.09	1.01	1.03	1.01
mkPktMerge	8.19	1.04	0.88	1.00	1.01
mkSMAdapter4B	6.37	0.89	0.98	1.08	1.02
or1200	5.72	0.91	1.13	1.07	1.01
raygentop	4.23	1.02	1.02	1.04	1.01
sha	5.50	0.97	1.06	1.16	1.03
stereovision0	5.56	0.97	1.04	1.16	1.03
stereovision1	6.13	1.00	1.06	1.05	1.00
stereovision2	3.36	1.04	1.05	1.08	1.00
stereovision3	5.47	1.12	1.07	1.08	1.11
geomean	4.94	1.00	1.02	1.07	1.02
stdev	1.75	0.07	0.05	0.05	0.03

in Table 4.1. Here, as expected, the results show similar quality of results for just over 2-fold speedup. However, the runtime gap is still quite large compared to the baseline (which has both speculation and pre-packing on). There are a few reasons for this. First, when pre-packing is off, the packer must work with a larger number of packable units. Second, the number of failed attempts at speculative packing increases slightly versus the baseline, because interconnect-aware pin counting alone may mistakenly accept certain illegal intermediate assignments to a fracturable logic element.

To isolate the impact of pre-packing, the final experiment investigates packing without speculation but with pre-packing. Table 4.3 shows the results of the experiment. The columns are the same as earlier. Here, as expected, the quality of results are basically the same as with the baseline. Packer runtime is in between all optimizations on and both speculation and pre-packing off.

Though not shown in the tables, with all optimizations turned on, total pack time is 3.4% of total runtime on average. We therefore conclude that AAPack runtime is practical for architecture exploration. However, the bulk of architecture exploration runtime is taken up in the search for minimum channel width. If the channel width is known, as is the case for a manufactured FPGA, then pack time is 20% longer than fixed channel width route time (when routing at 1.3x min W), which is substantial. Thus, we conclude that once an architecture is decided upon for mass production, then it may make sense to specialize the packing algorithm to reduce runtime.

We conclude that pre-packing improves both quality of results and packer runtime while speculative packing only reduces pack time. We also conclude that both optimizations combined produce better quality of results and runtime than any optimization in isolation.

Table 4.2: Quality of results of packing with speculation but without pre-packing (of LUT and FF pairs) normalized to the fully optimized baseline.

Circuit	Pack Time	Min W	Crit Path Delay	Num Soft Logic Blocks	Num Ext Nets
arm_core	1.82	1.04	1.02	1.06	1.05
bgm	1.04	1.00	0.96	1.01	1.00
blob_merge	3.11	1.00	1.03	1.03	1.00
boundtop	4.35	0.97	1.08	1.11	1.05
ch_intrinsics	2.25	1.00	1.00	1.08	0.99
diffeq1	2.91	0.86	1.06	1.12	1.03
diffeq2	2.13	0.93	1.01	1.07	1.00
LU8PEEng	1.27	1.03	1.03	1.02	1.00
LU32PEEng	1.24	1.05	0.96	1.02	1.01
mcml	1.70	0.93	1.04	1.18	1.05
mkDelayWorker32B	1.87	1.09	1.07	1.03	1.02
mkPktMerge	1.36	1.04	0.88	1.00	1.01
mkSMAdapter4B	2.83	0.89	1.00	1.09	1.02
or1200	2.72	0.91	1.13	1.07	1.01
raygentop	2.29	1.02	1.02	1.04	1.01
sha	3.00	0.97	1.06	1.16	1.03
stereovision0	3.13	0.88	1.07	1.16	1.03
stereovision1	3.22	1.02	1.04	1.05	1.00
stereovision2	1.87	1.04	1.05	1.08	1.00
stereovision3	2.52	1.12	1.07	1.08	1.11
geomean	2.18	0.99	1.03	1.07	1.02
stdev	0.84	0.07	0.05	0.05	0.03

Table 4.3: Quality of results of packing without speculation but with pre-packing normalized to the fully optimized baseline.

Circuit	Pack Time	Min W	Crit Path Delay	Num Soft Logic Blocks	Num Ext Nets
arm_core	3.27	1.03	1.01	1.00	1.00
bgm	2.14	1.02	0.98	1.00	1.00
blob_merge	3.31	1.04	1.02	1.00	1.00
boundtop	3.58	1.03	1.03	1.00	1.00
ch_intrinsics	3.86	1.03	0.99	1.00	0.99
diffeq1	3.91	1.03	1.00	1.00	1.00
diffeq2	4.39	0.96	1.02	1.00	1.00
LU8PEEng	2.17	1.03	1.01	1.00	1.00
LU32PEEng	2.04	1.03	1.00	1.00	1.00
mcml	2.60	1.00	0.97	1.00	1.00
mkDelayWorker32B	4.49	1.02	0.98	1.00	1.00
mkPktMerge	7.55	1.00	1.00	1.00	1.00
mkSMAdapter4B	3.58	0.84	0.93	1.00	1.00
or1200	3.67	1.00	1.00	1.00	1.00
raygentop	2.22	1.00	1.00	1.00	1.00
sha	3.09	1.00	1.00	1.00	1.01
stereovision0	2.68	1.00	1.04	1.00	1.00
stereovision1	2.74	0.98	1.21	1.00	1.00
stereovision2	1.99	1.03	1.05	1.00	1.00
stereovision3	2.98	1.00	1.00	1.00	1.00
geomean	3.14	1.00	1.01	1.00	1.00
stdev	1.26	0.04	0.05	0.00	0.00

Table 4.4: Quality of results comparing old router with new router on EArch with all optimizations on. Values are normalized old over new.

Circuit	Pack Time	Min W	Crit Path Delay	Num Soft Logic Blocks	Num Ext Nets
arm_core	3.98	1.03	1.00	1.00	1.02
bgm	5.88	0.96	1.00	1.01	1.02
blob_merge	3.96	1.02	1.02	1.00	1.01
boundtop	7.76	0.97	0.98	1.00	1.00
ch_intrinsics	12.31	0.97	1.03	1.00	1.02
diffeq1	2.85	0.80	1.02	1.00	1.00
diffeq2	43.81	1.14	0.99	1.07	1.06
LU8PEEng	6.36	1.03	1.01	1.01	1.01
LU32PEEng	6.00	1.00	1.00	1.01	1.01
mcml	3.55	1.00	1.00	1.00	1.02
mkDelayWorker32B	5.11	0.98	0.99	1.00	1.02
mkPktMerge	2.44	1.00	0.93	1.00	1.00
mkSMAdapter4B	8.60	0.89	0.89	0.99	1.03
or1200	6.39	0.91	1.01	1.00	1.01
raygentop	2.28	1.02	1.00	1.00	1.01
sha	3.73	0.97	1.00	1.01	1.06
stereovision0	2.63	0.97	1.04	1.00	1.00
stereovision1	3.37	1.00	1.00	1.00	1.01
stereovision2	1.57	0.99	1.00	1.00	1.00
stereovision3	3.39	1.12	0.97	1.00	1.01
geomean	4.79	0.99	0.99	1.00	1.02
stdev	9.09	0.07	0.03	0.02	0.02

Impact of Intra-Logic Block Router

This experiment compares the new intra-logic block router with the old intra-logic block router employed in my Master's thesis [56]. The previous router used the breadth-first search router found in VPR [9]. This is the same router as the one used to route inter-logic block connections but with parameters tuned more for speed because the intra-logic block routing problem is a much smaller one than the inter-logic block routing problem. The new router, being completely decoupled from the inter-logic block router, uses smaller data structures, as well as some algorithm changes beyond simple parameter tuning (see Section 4.2.5). Thus, we expect the new, more specialized router to outperform the older, more general router.

Table 4.4 shows the results of the experiment. The columns are the same as earlier. The pack time for the old router is 4.8-fold longer than for the new router. Quality of results are about the same. There are two effects at play for runtime. First, the old router fails intra-logic block routing slightly more frequently than the new router resulting in failed speculation and longer runtimes. Second, the old router is slower than the new router for this architecture. For extremely small circuits, such as diffeq2 and ch_intrinsics, a single instance of a failed speculative route can cause a large spike in runtime. If one is conservative, then removing these two outliers results in the new router having a 4-fold (rather than 4.8-fold) speedup compared to the old router. When speculation is turned off, then packer runtime using the old intra-logic block router is approximately 3-fold slower than for the new router.

There is a caveat that these results depend on the complexity of the architecture. For very simple architectures, such as a legacy non-fracturable LUT architectures with full crossbar interconnect, there is no noticeable runtime differences between the new router and the old router.

We conclude that the new, specialized, intra-logic block router results in approximately 3-fold to

Table 4.5: Impact of interconnect difficulty on quality of results. Interconnect difficulty is approximated with crossbar flexibility.

Xbar Pop	Pack Time	Min W	Crit Path Delay	Num Soft Logic Blocks	Num Ext Nets
100%	1.10	0.81	0.99	1.00	0.99
50% (EArch Baseline)	1.00	1.00	1.00	1.00	1.00
25%	1.67	1.04	1.01	1.00	1.08
12.5%	5.57	1.20	1.04	1.02	1.22

5-fold faster pack time than the older, more general router for a complex architecture such as EArch.

4.4.2 Performance Across a Range of Architectures

The purpose of this experiment is to measure how well AAPack adapts to increasingly difficult interconnect. We keep the same experimental setup as in the previous section but vary the architecture (based on EArch) in the following way: the crossbar in the soft logic block is given population values of 12.5%, 25%, 50%, and 100% (fully populated). Lower population implies a more difficult intra-logic block interconnect which in turn should result in longer runtimes and poorer quality of results.

The depopulated crossbars themselves are designed with some simplifying assumptions because the focus of this experiment is on the packing algorithm. First, crossbar delay is kept constant regardless of crossbar size. Second, in an ideal world, the router would optimize both inter-logic block routing and intra-logic block routing together. But, due to limitations with VTR, the inter-logic block router cannot directly modify intra-logic block routing. The only optimization available to the inter-logic block router at the logic block block level in VTR is the selection of which logically equivalent logic block pins to use. Therefore, to compensate for this limitation, the depopulated crossbars are composed of smaller, fully populated crossbars. Input pins that share the same, small, fully populated crossbar are logically equivalent to take advantage of the limited flexibility offered by the inter-logic block router.

Commercial FPGA architects usually design the interconnect so that fracturable LUTs may be easily moved to a different location within the same soft logic block [49]. This allows the final, external-to-logic block routing stage of the CAD flow to decide which output pins a fracturable LUT should drive, which is advantageous because the external router has information about routing congestion that is not available during packing. Although we do not have this CAD capability to swap fracturable LUTs during external-to-logic block routing in VTR, it makes sense to impose such a constraint to improve the realism of the depopulated crossbar. Thus, the depopulated crossbar is designed so that if a fracturable LUT gets moved to a different position, each input of that fracturable LUT keeps access to the exact same set of logic block pins.

Table 4.5 shows the quality of results sweeping cross different crossbar values. The columns from left to right are as follows: the population of the crossbar, pack time, minimum channel width (Min W), critical path delay at 1.3 times min W, number of soft logic blocks, and number of nets that connect two or more logic blocks. All values after the leftmost column are the geometric mean across all benchmarks normalized to the baseline EArch architecture (which has a crossbar population of 50%).

The results show trends that are roughly what we expect. Pack time tends to increase and quality of results tend to decrease with decreasing crossbar population. A crossbar population of 12.5% show significant routability degredation (higher min W) due to an increase in the number of external-to-logic-block nets. The techniques used to avoid detailed routing when checking the routability of intermediate

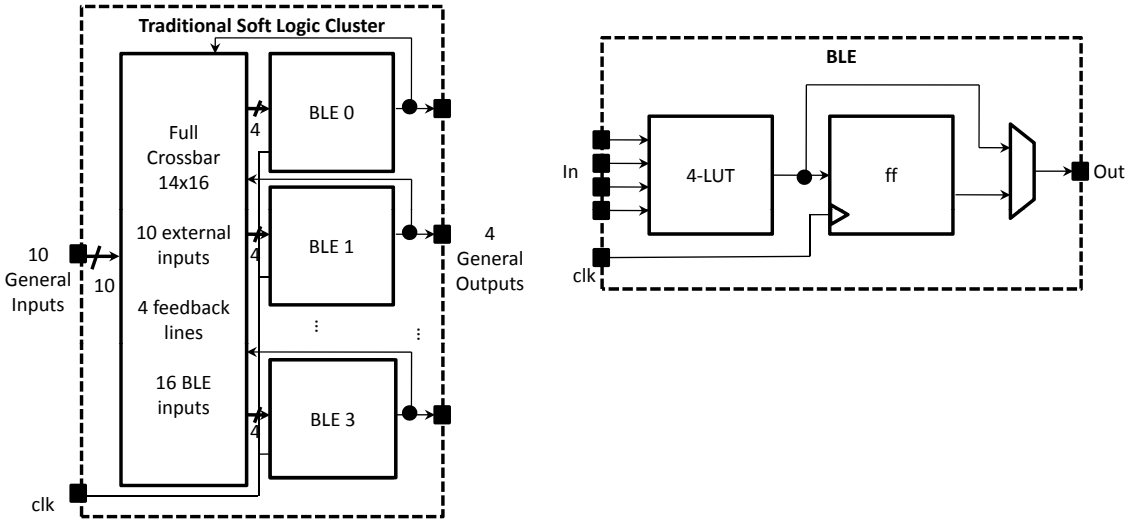

Figure 4.22: A basic soft logic block with typical parameters.

solutions fails at a crossbar population of 12.5% resulting in much higher pack time compared to crossbars with higher population. Perhaps a less expected result, logic density degrades only 2% at 12.5% crossbar population.

Note that in Table 4.5, minimum channel width values measured are conservative. We expect that if the inter-logic block router was allowed to route inside the logic block, then the CAD algorithm may find a better solution for the architectures with depopulated crossbars, because the external router would then have a wider optimization space available. Also, there are many ways to design depopluated crossbars beyond using smaller full crossbars [48].

The minor reduction in packer runtime from targeting the full crossbar architecture compared to the 50% populated crossbar architecture is interesting. For both architectures, there is sufficient interconnect for speculative packing to succeed, often within just a few intra-logic block routing iterations. So runtime differences between these architectures depend largely on how long each iteration of the intra-logic block router takes. Since the intra-logic block router employs a simple uniform-cost search (rather than a more advanced A-star search), the larger number of switches for the full crossbar case increases the runtime per iteration slightly more compared to the 50% populated case. That said, in the big picture, given the range of runtime differences across architectures, 10% difference is ultimately inconsequential.

We conclude that the packer runs faster for simpler interconnect and slower for more complex interconnect. We conclude that critical path delay and logic block density are both relatively insensitive to crossbar population down to as low as 12.5%.

4.4.3 Comparison to Other Packers on Simple Architectures

We now move onto the performance of AAPack on simple architectures. We compare AAPack to the open source packer T-VPack 4.30 [64]. Since much literature employs T-VPack as a baseline, the results obtained here may be used as a proxy to compare AAPack with other state-of-the-art packers for simple soft logic. The simple soft logic block itself was described in detail in Section 2.1.1. It is shown again in Figure 4.22 for the reader's convenience.

Table 4.6: Parameters of 4-LUT simple soft logic block architecture.

Parameter	Value
K	4
N	8
I	22
Fc_{in}	0.2
Fc_{out}	0.1
L	4
Switch Block Type	Modified Wilton

The key architectural parameters used in this experiment are presented in Table 4.6. The number of inputs to the LUTs (K) is 4. The number of LUTs per soft logic block (N) is 8. The number of inputs to a soft logic block (I) is set to 22 to match recommendations from prior research by Ahmed [2] for this architecture. The routing parameters match the closest architecture, named *N10K04L04.FC20FO10.AREA1DELAY1.CMOS45NM.BPTM*, in the IFAR repository [43]. This same IFAR architecture also provides the transistor-optimized area and delay values for the test architecture.

The 20 largest technology-mapped MCNC benchmarks, called the Toronto 20 [73], are used so that comparisons with other, closed-source, simple soft logic block packers may be made. The output of packing goes through the latest VPR 7.0 for placement and routing. For this experiment, T-VPack 4.30 was modified to output a netlist compatible with VPR 7.0. Thus, this study evaluates solely the packer – the rest of the flow is the same.

The standard VPR experiment flow is used. This flow starts with a technology-mapped circuit as input. This circuit undergoes packing, placement, and minimum channel width (minW) routing to find the minimum channel width. Afterwards, there is one more run of just routing at a higher channel width (more details on the exact values later) to measure critical path delay to simulate more realistic congestion conditions. All settings were left to default except the placement option *inner_num*, which was set to 10 to align these experimental parameters with prior literature. The machine used for these experiments is a 64-bit Intel Xeon 5160 at 3 GHz running in single core mode with access to 8GB of physical memory.

The quality of results is shown in Table 4.7. From left to right, the columns are as follows: 1) Name of the circuit, 2) Minimum channel width, 3) Critical path delay, 4) Number of external nets (nets that cross logic block boundaries) post-packing, and 5) number of soft logic blocks. For critical path delay, we set channel width to 1.3 times the largest min W of the two runs. This method of measuring critical path delay is unlike the previous experiments. Since the architecture is identical and the number of soft logic blocks is very close, we can obtain slightly better experimental control by using the same channel width for the same circuit when comparing AAPack vs T-VPack. All values are the measurement of the AAPack run divided by the T-VPack run. The last two rows show the geometric mean and standard of deviation of the results. For quality of results, AAPack demonstrates similar results to T-VPack. The change to the cost function of AAPack results in less external nets but this reduction in external nets does not have a significant impact on post-routed results for this architecture. AAPack, being a more flexible packer, is 8.3-fold slower than T-VPack. Not shown in this table is AAPack runtime, which is less than 5% of total runtime.

To measure the robustness of these results, we repeated the experiment on a different architecture. The second architecture parameters are shown in Table 4.8. This experiment uses larger LUTs and larger soft logic blocks. The IFAR architecture used as the base for this architectures is called

Table 4.7: Relative comparison of AAPack vs T-VPack on a simple soft logic architecture. Values shown are normalized measurements of AAPack/T-VPack.

Circuit	Min W	Crit Delay	Pack Time	Num Ext Nets	Num CLB
alu4	0.95	1.02	13.14	0.92	1.00
apex2	1.00	1.00	11.04	0.84	1.00
apex4	1.00	1.07	6.30	0.85	0.98
bigkey	0.94	1.12	5.23	1.21	1.00
clma	1.06	0.97	13.64	0.94	1.00
des	1.00	1.03	3.53	0.87	1.00
diffeq	1.00	1.12	6.92	0.89	0.99
dsip	1.00	1.04	4.02	1.05	1.00
elliptic	1.00	1.02	10.05	0.89	1.00
ex1010	0.91	0.99	14.80	0.90	0.98
ex5p	0.97	1.00	4.45	0.94	0.99
frisc	1.03	1.04	5.77	0.89	1.00
misex3	0.96	1.03	4.99	0.88	1.00
pdc	0.97	0.94	17.43	0.95	1.00
s298	1.00	0.93	6.48	1.01	1.00
s38417	0.92	1.02	8.99	0.90	1.00
s38584.1	1.00	0.96	6.77	0.96	1.00
seq	1.04	1.02	7.83	0.90	1.00
spla	1.00	1.05	14.21	0.94	1.00
tseng	0.85	0.99	6.63	0.74	0.99
geomean	0.98	1.02	7.76	0.92	0.996
stdev	0.05	0.05	4.10	0.09	0.01

Table 4.8: Parameters of 6-LUT simple soft logic block architecture.

Parameter	Value
K	6
N	10
I	33
Fc_{in}	0.15
Fc_{out}	0.1
L	4
Switch Block Type	Modified Wilton

N10K06L04.FC15.AREA1DELAY1.CMOS45NM.BPTM. The benchmarks employed are the Toronto 20, but these circuits are technology-mapped to 6-LUTs using the newer, *wiremap*, algorithm [39].

The results of this run are shown in Table 4.9. The columns are the same as before. Unlike the previous experiment, when compared to T-VPack, AAPack produces better quality of results post-routing where min W drops 5.8% and the number of external nets drops by 14% on average. Critical path delay is within noise.

Overall, these results show that a general packing algorithm can achieve comparable quality of results to a specialized algorithm for simple soft logic blocks with an under 10x packer runtime penalty. To put this runtime number in perspective, packing is typically only a small fraction of total CAD runtime. Despite being 10x slower than T-VPack, on average, AAPack runtime is under 6% of total runtime, on average, for these runs. On the largest circuit (clma on the 4LUT architecture), AAPack takes just 4.6 seconds. Therefore, we conclude that the runtime overhead for generality is reasonable, given its much greater explorative potential.

Table 4.9: Relative comparison of AAPack vs T-VPack on a larger simple soft logic architecture. Values shown are normalized measurements of AAPack/T-VPack.

Circuit	Min W	Crit Delay	Pack Time	Num Ext Nets	Num CLB
alu4	0.83	1.00	2.32	0.75	1.00
apex2	0.97	0.97	3.52	0.76	1.00
apex4	0.91	1.04	2.91	0.77	0.99
bigkey	1.06	0.97	8.17	0.95	1.00
clma	1.03	1.00	13.20	0.87	1.00
des	0.94	0.91	4.29	0.93	1.00
diffeq	1.24	1.02	4.09	1.06	1.00
dsip	0.94	0.93	8.14	0.88	1.00
elliptic	1.00	1.04	10.43	0.87	1.00
ex1010	0.96	1.02	16.51	0.91	1.00
ex5p	0.93	0.95	6.45	0.73	1.00
frisc	0.97	1.00	6.96	0.94	1.00
misex3	0.92	0.96	7.53	0.78	1.00
pdc	0.93	0.94	16.80	0.80	1.00
s298	0.87	1.00	5.50	0.81	1.00
s38417	0.89	0.94	12.24	0.96	1.00
s38584.1	1.00	0.99	5.79	1.00	1.00
seq	0.97	1.00	5.14	0.78	1.00
spla	0.95	1.03	18.78	0.81	1.00
tseng	0.65	1.02	3.99	1.03	1.00
geomean	0.94	0.99	6.86	0.86	0.999
stdev	0.11	0.04	4.94	0.10	0.00

4.4.4 Runtime Analysis

As digital designers create circuits of ever larger size, CAD runtime scaling has become increasingly important. This section analyzes the runtime behaviour of the packer. We show theoretical worst-case runtime scaling first, followed by empirical runtime scaling, then a breakdown of pack time with respect to the rest of the CAD flow.

In a pathological scenario, every netlist atom has connections to every other netlist atom through very high fanout nets. In this situation, the packer is not able to restrict the search space to locally connected groups, so every atom that gets packed will cause the packer to visit, possibly, every other atom. Let the number of atoms be N, then in this worst-case scenario we have N^2 behaviour. Let the maximum number of primitives of a given logic block be P. Every atom may, in the worst case, potentially be considered for up to P placement locations. Furthermore, it may be necessary to do detailed intra-logic block routing for each placement location. Let the maximum number of pins within a logic block be V and the maximum number of interconnect edges within a logic block be E. The time to route one source pin to another sink pin would then be $O((E+V)*logV)$ [22] but the number of source or sink pins may be up to V. Therefore, asymptotic runtime for packing, in the worst case, is as follows:

$$O(N^2 * P * (E+V) * log(V) * V) \tag{4.8}$$

If we assume that the circuit is very large then the logic block can be viewed as a constant during packing. The worst-case asymptotic runtime then simplifies to the following:

$$O(N^2) \tag{4.9}$$

Table 4.10: Packer runtime with respect to number of netlist atoms.

Circuit	Pack Time	Num Netlist Atoms	Atoms per Sec
stereovision3	0.32	345	1085
diffeq2	0.49	597	1223
diffeq1	0.62	882	1412
ch_intrinsics	0.82	895	1094
mkPktMerge	0.98	1232	1254
sha	2.95	3288	1116
mkSMAdapter4B	3.72	3786	1018
raygentop	6.02	4364	725
or1200	4.87	4882	1002
boundtop	4.63	5325	1149
mkDelayWorker32B	11.53	11112	964
blob_merge	15.69	12468	795
arm_core	31.18	20876	670
stereovision1	19.20	22452	1169
stereovision0	20.32	24994	1230
LU8PEEng	101.39	33182	327
bgm	114.19	43437	380
stereovision2	44.11	44160	1001
LU32PEEng	390.80	108474	278
mcml	296.00	172521	583

However, this worst-case scenario rarely happens. Table 4.10 shows the rate of packing for the VTR benchmarks on the EArch architecture. The table is sorted based on number of atoms of the netlist in ascending order. The last column shows the packing rate in atoms per second. Overall, the rate of packing does slows with larger circuits but slows to approximately half the rate for 100-fold increase in netlist size. Thus, we conclude that packer runtime scales worse than linear in practice, but only slightly so.

Figure 4.23 and Figure 4.24 reproduces the runtime breakdown charts from Section 3.4 when mapping the VTR benchmarks to the EArch architecture. We conclude that pack time is 3.4% of total CAD time for an architecture experiment that employs minimum channel width search, and pack time is 17.5% of total CAD time for a production-style run with a fixed channel widths and faster placement parameters.

4.5 Conclusions

In this chapter, we have formally defined the architecture-aware packing problem where a netlist-like description of the logic block architectures is given as input to packing. We have described a greedy algorithm to solve this problem. Due to the flexibility of the architecture description, this algorithm must solve a placement-and-routing subproblem within the logic block itself. We employed a greedy placement algorithm and a variation of the Pathfinder [65] negotiated congestion routing algorithm for this subproblem [57]. Since the routing subproblem is particularly time consuming, we proposed three techniques (speculative packing, interconnect-aware pin counting, and pre-packing) to help speed it up [62]. In addition to runtime benefits, pre-packing improves quality of results by making use of an architect's insight regarding what to keep together during packing.

Our experiments and results demonstrate that the packer can target logic blocks with modern features such as soft logic blocks with fracturable LUTs, carry chains, and depopulated crossbars. We showed that although speculation and molecules each provide runtime speedups of approximately 2-fold to 3-fold

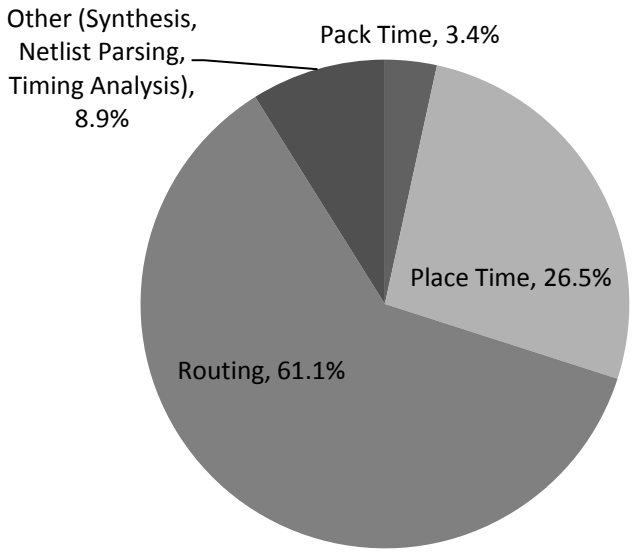

Figure 4.23: Breakdown of percent of total runtime for the different CAD stages for a minimum channel width search experiment. Values shown are the arithmetic mean of the relative runtime for each stage across the VTR benchmarks.

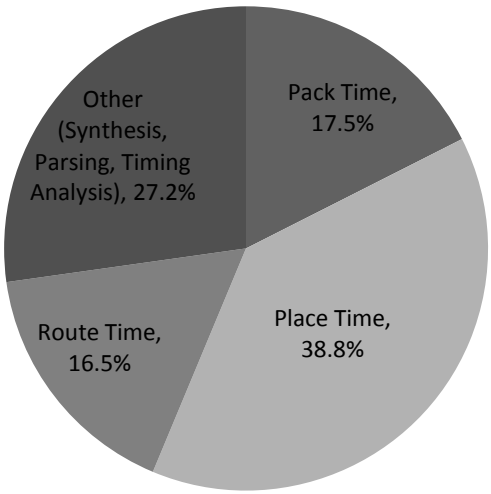

Figure 4.24: Breakdown of percent of total runtime for the different CAD stages for a production environment. Values shown are the arithmetic mean of the relative runtime for each stage across the VTR benchmarks.

in isolation, they together provide 5-fold speedup. Comparisons using a legacy architecture reveal a few percent improvement to minimum channel width and marginal improvement to critical path delay versus T-VPack for about a 10-fold runtime overhead from architecture flexibility.

For a minimum channel width search experiment, pack time is just 3.4% of total CAD time, so we conclude that the runtime overhead from architecture-aware packing is acceptable for architecture exploration. Outside of architecture exploration, when the channel width is known, AAPack runtime takes a much larger 17.5% of total CAD time, hence it may make sense to use architecture-specific packers after the architecture is determined.

Chapter 5

Architecture Study: Hard Adders and Carry Chains

This chapter describes an exploration of the architecture of hard adders and carry chains in an FPGA that uses the infrastructure described in Chapter 3 and the packing algorithm described in Chapter 4. It serves as both an illustration of the capability of the work of those chapters, and an interesting FPGA architectural result in its own right.

5.1 Introduction

One of the central questions in FPGA architecture is that of deciding which functions to harden and which to leave for implementation in the soft logic [79]. A function should be hardened if it appears often in the set of used applications, and if there is a large advantage when it is implemented in hard logic rather than soft. This argument has held sway in the case of adder-type arithmetic functions – they appear often and hardened adders are much faster than soft adders. Consequently, commercial devices commonly have hardened adder and/or carry logic and routing [51] [26] [45] [98]. Indeed, hardened arithmetic structures have been a longstanding feature of commercial FPGAs, yet there has been no comprehensive published study of the performance benefits they offer on complete designs or their cost in terms of area. This architecture study aims to fill that gap.

There are many degrees of freedom in the electrical and architectural design of hard adder logic, and in the software used to map a complete application to such structures. There has been little published work that sheds light on the set of such choices, nor the impact they have on the resulting implementations of complete designs in FPGAs. We study a number of these choices and determine their impact on performance, area and CAD complexity. We focus on architectures where the hard adders are integrated into the soft logic block. Some examples include: First, the determination of how an adder interacts with nearby LUTs and flip-flops. Second, the trade-off of performance and area between larger, faster, multi-bit adders and more flexible, slower, smaller single-bit adders. Third, the design of the connection between the carry bits of adjacent hard adder units; for example, should there be dedicated links for the carry signal across soft logic block boundaries so that wide additions may be done at high speed but with a more constrained CAD placement problem? Or should those connections cross soft logic boundaries using the general-purpose interconnect of the FPGA? These are important implementation details that

an architect must decide on when embedding hard adders in with soft logic. We present quantitative measurements of the impact of these decisions.

Prior published work on hardened arithmetic, described in Section 2.1.3, focused on the implementation of arithmetic structures, and evaluated results on microbenchmarks such as adders and adder trees or very small designs. A full design, on the other hand, imposes many other demands on the FPGA and its CAD flow. We seek to measure the impact of different hard adder choices not only on microbenchmarks, but also on complete designs with a full CAD flow.

This chapter is organized as follows: We first describe the FPGA architecture and circuit design that serve as the basis for the exploration. Afterwards, we describe the variations of the hard arithmetic structures and their interaction with the soft logic. Then, we present results of the various architectures on both pure-adder *microbenchmarks* and full application circuits. Finally, we give some concluding remarks.

5.2 Base Architecture Model

The base FPGA architecture used in this study is designed in a 22nm CMOS process, and is a heterogeneous architecture with soft logic blocks, simple I/Os, configurable memories and fracturable multipliers. We are *not* using the EArch architecture for this study because we require more precise area and delay values for the internals of the architecture than what is available in EArch. Figure 5.1 illustrates the base soft logic block used in this study, which contains eight Basic Logic Elements (BLEs), 40 general inputs, eight general outputs, one carry in (cin) pin and one carry out (cout) pin. The BLE consists of a 6-input LUT, a hard adder, and a flip-flop that can optionally register the BLE output. The BLE has a cin and a cout pin that connects the hard adder to adjacent BLEs (and possibly to BLEs in adjacent soft logic blocks). A fast path connects the flip-flop output to the LUT input. The specific details regarding the hard adder are described later in Section 5.3. We also consider one architecture that does not contain hardened arithmetic, and hence has neither cin nor cout pins.

The internal connectivity of the blocks is provided by a 50% depopulated crossbar that connects block inputs and BLE outputs to the BLE inputs. We have chosen a depopulated crossbar as this is common in commercial devices [98, 51]. The depopulated crossbar itself is composed of four, smaller, fully populated crossbars designed by Chiasson in [18]; this depopulation results in the soft logic block inputs being divided into four groups of ten logically equivalent pins. The input pins are evenly distributed on the bottom and the right sides of the logic block, as this simplifies the layout of the FPGA.

There are some important implications that carry chains create. Consider the logic block of Figure 5.1 but without hard carry links. Which BLE performs which function can be changed by the routing stage of the CAD flow to allow different functions to access different output pins – the outputs are thus *logically equivalent*. When the carry links of the BLEs are used however, the order of those BLEs are fixed and cannot be exchanged, so the outputs of BLEs using their carry function are not logically equivalent. The VTR CAD flow does not have the capability to selectively switch off output pin logical equivalence in cases when the carry links are used by the BLEs. Hence, for correctness, we do not allow any BLE swaps at all, thus removing all output logical equivalence. To compensate for this restriction, each output pin can directly access two sides of the logic block, and hence both a vertical and a horizontal channel. Turning off logical equivalence for all outputs will lead to a slight pessimism on the routability of the soft logic only architecture vs. that of the hard adder architectures, but we believe the impact is small.

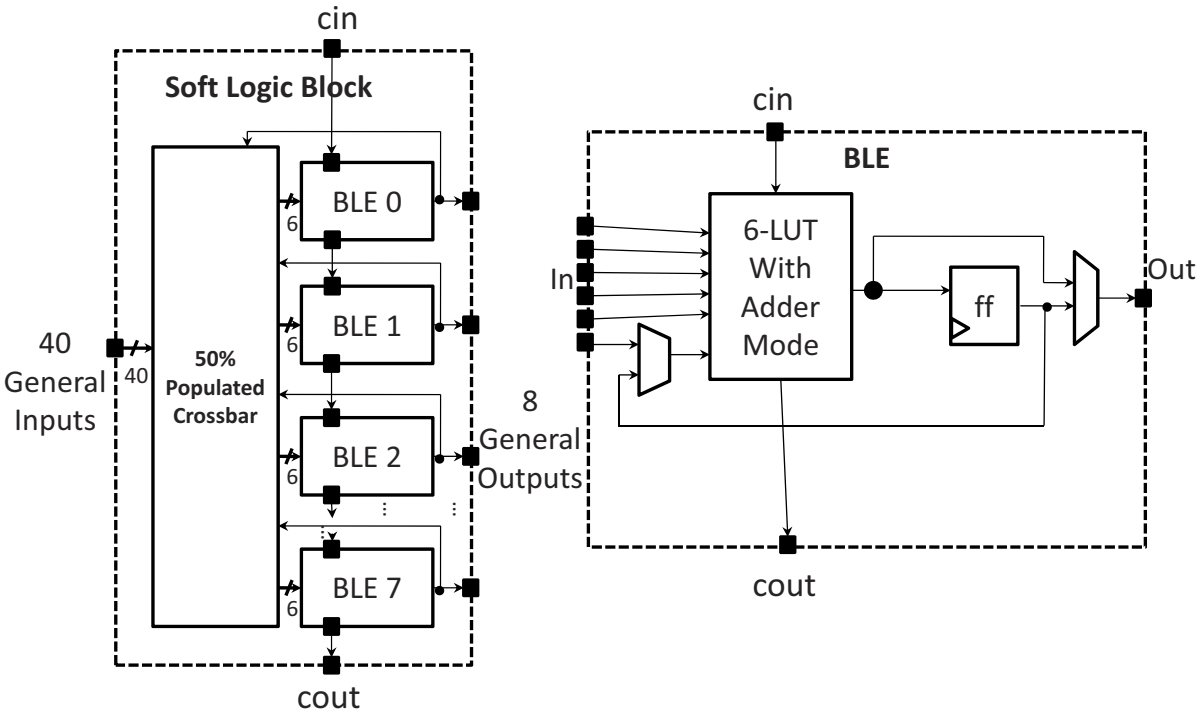

Figure 5.1: The base soft logic block.

Table 5.1: Routing architecture parameters.

Parameter	Value
Cluster input flexibility (Fc_{in})	0.2
Cluster output flexibility (Fc_{out})	0.1
Switch block flexibility (Fs)	3
Wire segment length (L)	4
Switch Block Type	Wilton
Interconnect Style	Single-driver

Table 5.2: Properties of the 1-bit hard adder used in this study.

Property	Value
Area	47.7 MWTAs
Delay cin to cout	11 ps
Delay sumin to cout	56 ps
Delay cin to sumout	30 ps
Delay sumin to sumout	83 ps

Table 5.3: Properties of the 4-bit carry-lookahead adder used in this study.

Property	Value
Area	257 MWTAs
Delay cin to cout	20 ps
Delay sumin to cout	80 ps
Delay cin to sumout LSB	25 ps
Delay cin to sumout MSB	30 ps
Delay sumin to sumout LSB	65 ps
Delay sumin to sumout MSB	82 ps

Table 5.1 gives the routing architecture parameters of the base architecture, which are chosen to be in line with the recommendations of prior research [42]. The hard memory logic block can implement memories of different aspect ratios ranging from 32Kx1 down to 1Kx32 for both dual-port and single-port modes. The multiplier logic block can implement a 36x36 multiplier that can optionally fracture to two 18x18 multipliers. Each 18x18 multiplier can further fracture down to two 9x9 multipliers.

The transistor-level design of the base soft logic blocks and routing architecture was done by Chiasson [61] using the COFFE tool [18] and a 22nm CMOS technology. The architecture uses pass gates. The statically controlled pass gates in the interconnect switches are gate-boosted by 0.2V. The architecture, area, and delay models for the memories and multipliers are scaled to 22nm from the 40nm EArch architecture.

5.3 Hard Adder and Carry Chain Design

The goal of this chapter is to explore various hard adder and carry chain architectures, and to do so in the context of careful electrical design of the key circuits. The two hard adder primitives in this study are hand-optimized at the transistor level by Huda [61]. The first adder primitive is a basic 1-bit full adder. In a soft logic block, eight of these full adders are linearly chained together to form a ripple carry chain. Table 5.2 shows the properties of the 1-bit hard full adder used in this study. Area is measured as minimum width transistor areas (MWTAs), using the transistor drive-to-area conversion equations of [18]. The adder circuitry, LUTs and routing are all designed with a similar goal of minimizing the area-delay product of the FPGA, and the cin-to-cout path of the adder is particularly optimized for delay as it occurs n-1 times within an n-bit adder.

The second adder primitive is a 4-bit carry-lookahead adder (CLA). Each logic block contains two of these 4-bit adders chained in a ripple carry fashion. Table 5.3 shows the properties of the 4-bit carry-lookahead adder used in this study. The carry-lookahead optimization allows for a faster carry path (20 ps) compared to a ripple of four 1-bit adders (44 ps) when performing a 4-bit addition. The CLA design trades off flexibility (as some bits are wasted if the desired adder length is not divisible by 4) and area in exchange for speed.

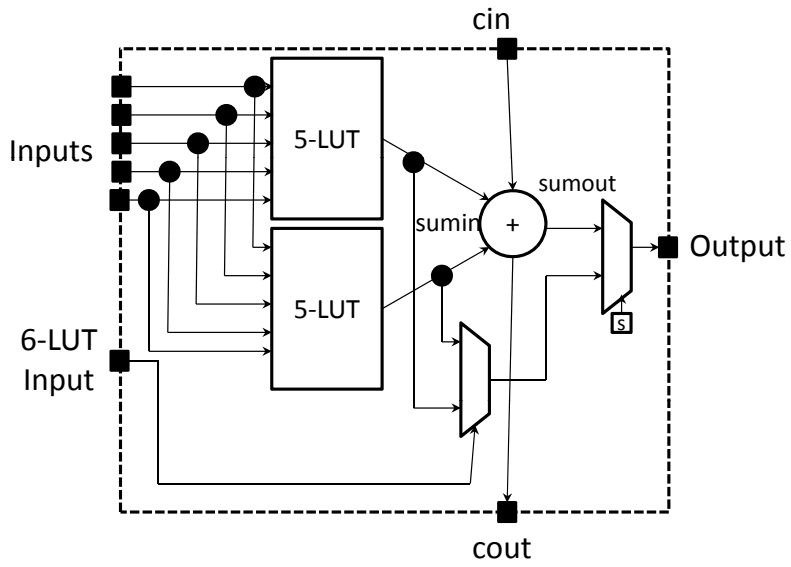

Figure 5.2: A *balanced* 6-LUT and adder interaction where both adder inputs are driven by 5-LUTs.

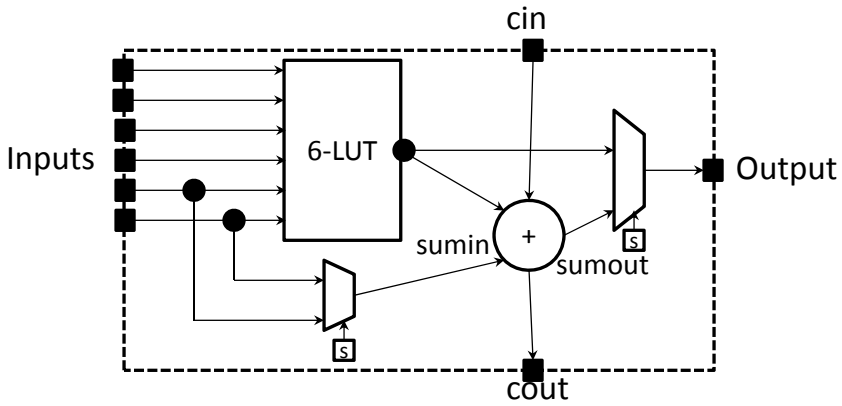

Figure 5.3: An *unbalanced* 6-LUT and adder interaction where the 6-LUT drives only one adder input.

Figure 5.2 shows one of the ways that we explore interaction between the adder and LUT. Here, we make use of the property that a 6-LUT is constructed with two 5-LUTs and a mux. If that mux is bypassed, then the adder can be driven by two 5-LUTs, where the LUTs share inputs. If the adder is not used, then another mux can be used to produce the 6-LUT output. We call this the *balanced* LUT interaction, and its underlying hypothesis is that a symmetric amount of prior logic is the most appropriate architecture. Example circuits that may benefit from this architecture would be applications where multiplexers select the inputs to an adder.

Figure 5.3 shows another LUT-adder interaction architecture that we will explore. Here, the 6-LUT output drives one of the adder inputs and the other adder input is driven by one of the 6-LUT *inputs*. As with the previous case, if the adder is not used, then another mux can be used to select the 6-LUT output. We call this the *unbalanced* LUT interaction. We model each additional SRAM-controlled 2-to-1 mux (one per BLE for the balanced LUT interaction, two per BLE for the unbalanced LUT interaction) as having 22 ps of delay and occupying 15 minimum width transistor areas (including the

Table 5.4: Architecture acronyms.

Acronym	Architecture
Soft	Soft logic only
Ripple	1-bit ripple carry, balanced LUT
U-Ripple	1-bit ripple carry, unbalanced LUT
CLA	4-bit CLA, balanced LUT
U-CLA	4-bit CLA, unbalanced LUT

SRAM configuration bit). The underlying hypothesis for this architecture is that there might be an advantage to allowing a faster input into one side of the adder, which may improve circuit speed.

A third type of architecture we are interested in are those with hardened adders but no dedicated carry link between logic blocks. Here, both the cin and cout pin are treated as though they are regular input and output pins with respect to the inter-block routing architecture. Within the logic block, the carry signals maintain the same restricted connections. We create two physically equivalent pins at the right and bottom sides of the logic block for both carry-in and carry-out (i.e. 4 pins in total). For architectures that have a dedicated carry link, the carry link has a delay of 20 ps.

Finally, there are a few different ways to implement the starting location of a multi-bit addition. One can place a mux at every carry link that can select from logic-0, logic-1, or a carry signal of a previous stage but this can incur a significant delay penalty because every carry link must now go through a mux. Alternatively, one can place these muxes only on selected carry links, thus minimizing the overhead of excessive muxing but at the cost of having fewer locations where an addition may begin. This latter approach is typical in commercial devices. Alternatively, the responsibility for starting an addition can be implemented in a front-end CAD tool – the tool can pad the addition with a dummy LSB that generates a 0 or a 1 carry-in for addition and subtraction, respectively. We employ this approach in our research.

5.4 Experiments and Results

This section describes the experiments and results of the study. We begin with an experiment on pure adder benchmarks to characterize the architectures then describe a detailed study on complete user designs. Questions that we explore include: what is the right adder granularity (one-bit ripple vs. four-bit CLA)? Should the LUT structure feeding the adder be symmetrical? And, how useful are high-speed inter-CLB carry links?

Table 5.4 lists the 5 different ways of supporting arithmetic in an FPGA soft logic block that we investigate. For the four architectures with hard adders, we also investigate both flexible and dedicated inter-logic-block links. Thus, this results in a total of 9 architectures in these experiments.

Table 5.5 shows the area, in MWTAs, for each soft logic block architecture. Hard adders and carry chains increase an individual logic block area by 4% to 6%.

The application circuit benchmarks we use are the VTR benchmarks, described in Section 3.1, excluding any benchmark under 1,000 6-LUTs in size. In addition, the mkDelayWorker32B benchmark is excluded as it caused ABC to crash. We will refer to these as the VTR+ benchmarks. The geometric average *atom* count across all 14 circuits is 11,700.

The VTR CAD flow employed is described in Section 3.3. The software used is revision 4310 of the publicly available trunk, which has advanced substantially beyond the latest, VTR 7.0, release. The

Table 5.5: Area of each soft logic block for each architecture in MWTAs.

Arch	Area
Soft	12039
Ripple	12541
U-Ripple	12660
CLA	12674
U-CLA	12794

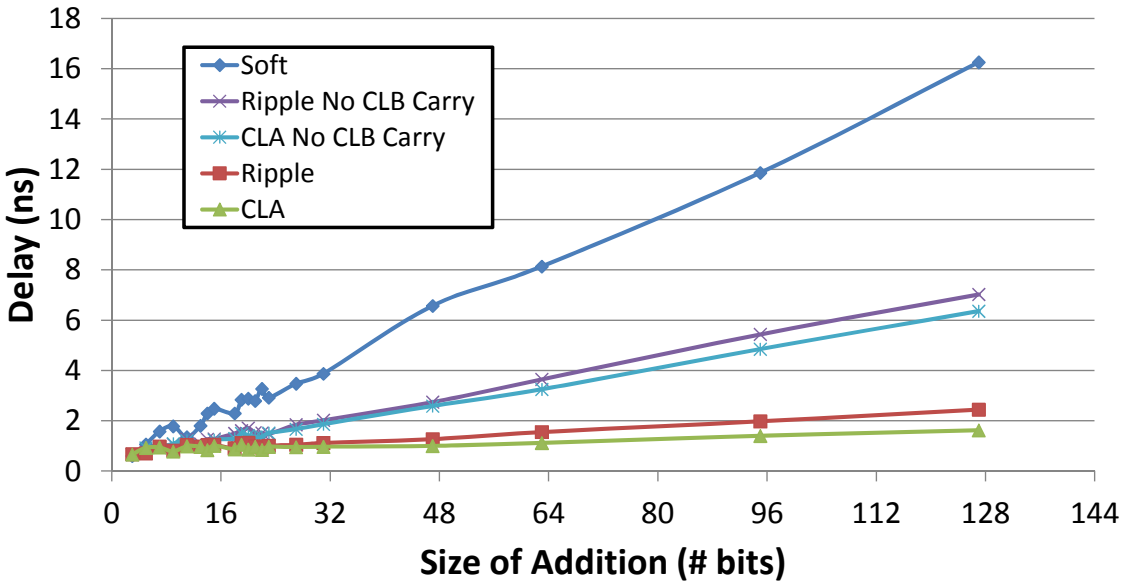

Figure 5.4: Delay vs adder length for various architectures.

placement option *inner_num* was set from a default of 1 back to the historical value of 10 because this produced both better quality of results and faster overall experiment runtimes. The machine used for these experiments is a 64-bit Intel Xeon 5160 at 3 GHz running in single core mode with access to 8GB of physical memory.

5.4.1 Pure Adder Experiment

Before exploring the effect of each architecture on full designs, it is instructive to measure their effect on various sizes of simple, pure adders. Here, each circuit is an adder of N bits, where N ranges from 2 to 127. Both the inputs and outputs of the adder are registered, so that critical path delay measurement is a direct function of the adder combinational logic delay.

Figure 5.4 shows the impact on critical path delay vs. width of addition, for the Soft, Ripple and CLA architectures, where the critical path delay is averaged over three placement seeds. In addition, two variants of the Ripple and CLA architectures are included, labelled *no CLB carry*, in which the general-purpose interconnect is used to implement carry links across soft logic blocks, rather than using dedicated carry links. The unbalanced architectures are not included here as their performance difference vs. balanced is similar on the microbenchmarks.

These results show trends that we generally expect. Delay grows linearly with adder size and more "hardened" architectures are faster. In the extreme case, for 127 bit addition, it is interesting to note

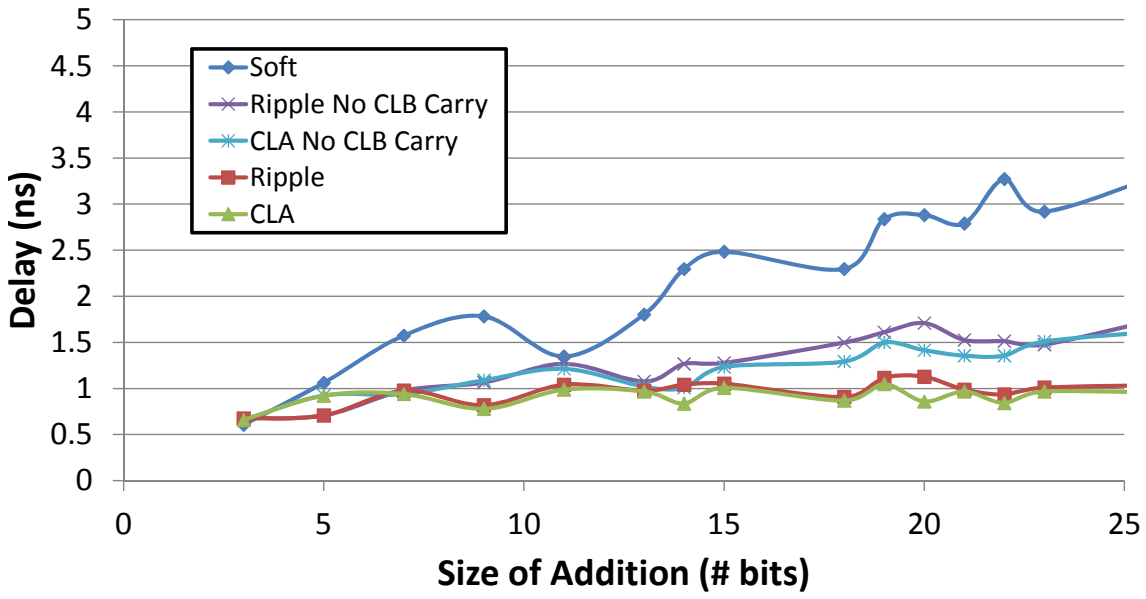

Figure 5.5: Delay vs adder length for various architectures for additions under 25 bits.

that a pure soft adder is ten times slower than the fastest (CLA) adder. The *no CLB carry* circuits have delay values in between fully hard and fully soft adder architectures. While the CLA architecture is the fastest of all, ripple carry is only 14% slower for 32 bit adders, and 43% slower for 127 bit addition. A ripple architecture can sustain 500 MHz operation even at 96-bit addition.

Figure 5.5 shows the delays for additions under 25 bits for the different architectures. This figure shows that for smaller additions, that swings in delay for the soft logic architecture is much greater than that of the hard adder architectures. For hard adders, the lack of CAD flexibility forces a predictable physical design, thus reducing CAD noise for these microbenchmarks when compared against soft adders. The combination of higher and more predictable performance provided by hard adders, especially those with hard inter-CLB links, is very desirable.

The data from this experiment also shows that a 3-bit addition implemented in soft logic is actually slightly faster than any of the hard-logic adders. This suggests that there is a threshold below which addition/subtraction should be implemented in soft logic and above which they should be implemented using hard adders.

5.4.2 Application Circuit Statistics

Table 5.6 provides statistics on the VTR+ benchmarks, and includes the number of addition/subtraction functions found in the benchmarks. The table columns list the number of 6-LUTs, the number of adder bits after elaboration, the length of the longest adder chain in bits, the average adder chain length, and the ratio of adder bits to LUTs. The benchmarks exhibit a wide range in the number and length of addition/subtraction functions. On average, the ratio of adder bits divided by the number of 6-LUTs is 0.21, indicating arithmetic is plentiful and hence it is reasonable to include hard adder circuitry in *every* CLB. The widest addition/subtraction generated in these benchmarks is 65 bits, which corresponds to a 64-bit operation (as the first bit must always be used to generate the carry-in signal). For *blob_merge*,

Table 5.6: Benchmark Statistics when mapped to Ripple architecture.

Circuit	Num 6LUTs	Num Add Bits	Max Add Len	Avg Add Len	Add/LUT Ratio
arm_core	13812	537	35	9.2	0.04
bgm	32337	5438	25	9.3	0.17
blob_merge	7843	3754	13	10.0	0.48
boundtop	2846	309	19	7.2	0.11
LU8PEEng	21668	3251	47	11.0	0.15
LU32PEEng	73828	8249	47	11.9	0.11
mcml	94048	24302	65	47.5	0.26
mkSMAdapter4B	1819	431	33	6.9	0.24
or1200	2813	534	65	23.9	0.19
raygentop	1778	580	32	11.8	0.33
sha	1994	309	33	24.0	0.15
stereovision0	8282	2920	18	11.2	0.35
stereovision1	7845	2388	19	6.4	0.30
stereovision2	11006	13843	32	23.9	1.26
geomean	8606	1807	31.2	12.5	0.21

the longest chain has just 13 bits. The geometric mean of the longest addition/subtraction lengths is 31.2 bits. The most adder-intensive circuit is *stereovision*2 with 1.26 adders per LUT, while the least adder-heavy circuit is *arm_core* at 0.04. These measurements correspond well with other modern benchmarks. For the TITAN benchmarks (with the SPARC cores excluded because these cores have almost no adders at all) [69], the geometric average of the fraction of LUTs in arithmetic mode and the maximum of length of addition/subtraction is 0.22 and 35.8, respectively.

5.4.3 Threshold of When to Use Hard Adders

While hardened adder and carry logic is clearly good to use for wide arithmetic structures, for small adders the flexibility provided by soft logic might actually prove superior as hard adders impose a boundary across which it is difficult for logic synthesis to optimize. We define the *hard adder threshold* as the size, in bits, of addition/subtraction above which the CAD flow will implement the addition/subtraction with hard adders and below or equal to which the function is implemented in soft logic.

Figure 5.6 shows the impact on delay of different hard adder thresholds when we target the ripple carry architecture. The x-axis shows the hard adder threshold in bits. The y-axis shows the geometric mean of the critical path delay over the 14 circuits of VTR+. There is a general trend towards achieving a minimum mean delay at a threshold of around 12 bits.

Figure 5.7 shows the area impact of different hard adder thresholds. The x-axis is again the hard adder threshold, while the y-axis shows geometric mean of the area for all benchmarks. Area is computed as the total number of soft logic blocks (CLBs) multiplied by the area of a soft logic tile, where a tile is one soft logic block with interconnect.

The area consumed using an architecture with hard adders is on average more than that of an equivalent architecture without carry chains. We see a gradual drop in area with an increasing hard adder threshold: area drops from 10% above the soft adder architecture with a hard adder threshold of 0, to 5% above with a threshold of 12. Note that, however, preliminary measurements we made on commercial FPGAs showed that using carry chains in the CAD flow reduced area. We therefore suspect that with further improvements in logic synthesis, the remaining 5% area penalty could be eliminated.

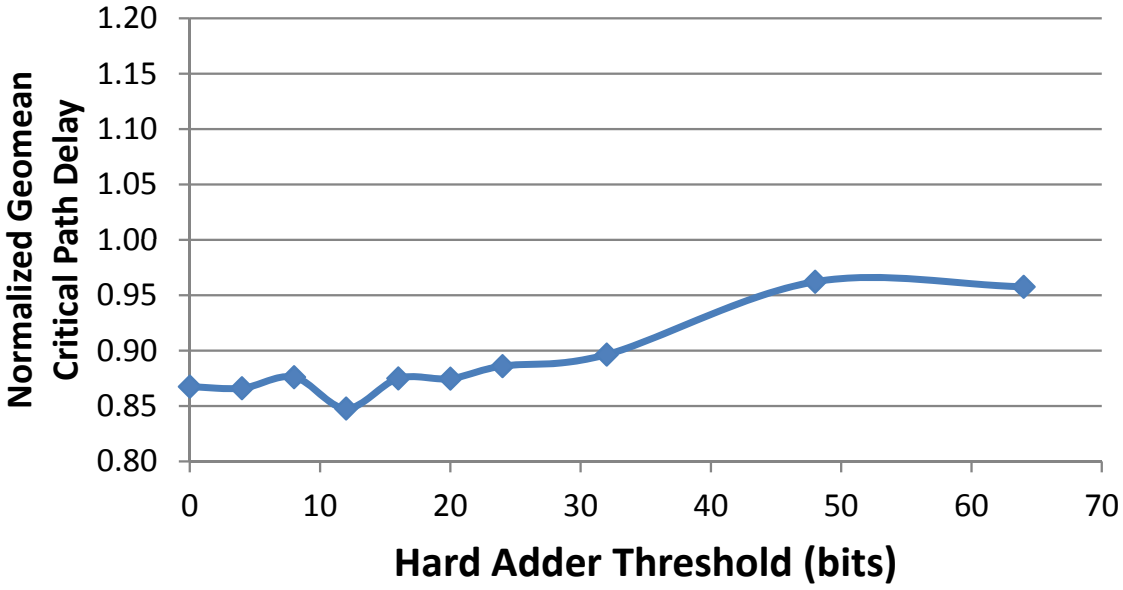

Figure 5.6: Circuit speed vs. hard adder threshold. Results are the average across 14 benchmarks and normalized to the soft implementation.

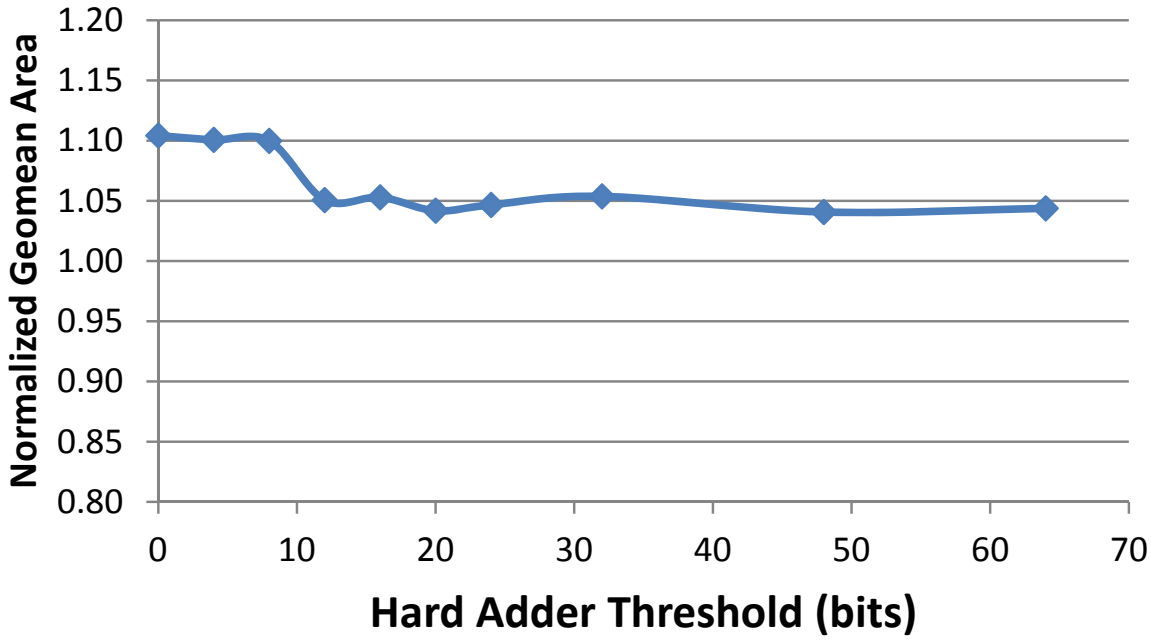

Figure 5.7: Average area of different hard adder thresholds normalized to the soft architecture.

Table 5.7: Geometric mean critical path delays across the VTR+ benchmarks for different hard adder architectures.

Arch	32-bit Add Delay (ns)	Application Circuits Geomean Delay (ns)
Soft	3.87	14.14
Ripple	1.12	11.90
U-Ripple	1.05	12.03
CLA	0.97	11.85
U-CLA	0.90	11.65

Table 5.8: Delay for different hard adder architectures, normalized to the soft logic architecture.

Arch	32-bit Add Delay	Application Circuits Geomean Delay
Ripple	0.290	0.842
U-Ripple	0.272	0.850
CLA	0.250	0.838
U-CLA	0.232	0.824

For these benchmarks and on this architecture, we conclude that the best hard adder threshold is approximately 12 bits. Therefore, all subsequent experiments use a hard adder threshold set to 12.

5.4.4 Microbenchmarks vs. Application Circuits

An interesting first comparison is to assess the impact of hard adders on application circuits as compared to those above with the pure adders microbenchmarks. We use a 32-bit adder as a representative microbenchmark, as this is close to the average size of the longest adders in the application circuits. Table 5.7 shows the geometric mean critical path delay across all 18 VTR+ circuits for each of the architectures. Table 5.8 shows these delays normalized to the soft logic architecture. A more detailed circuit-by-circuit breakdown of application circuit delays are provided in Appendix B. An isolated 32-bit adder sees a compelling delay reduction of 71% to 77% with hard carry architectures, while application circuits see much smaller (but still very significant) delay reductions of 15% to 18%, depending on the hard carry architecture. This is a common outcome in the hardening of any kind of circuit – the final impact on critical path delay is limited because other paths in the design quickly become more critical than the hardened circuit. On the application circuits, the best average delay reduction achieved by hardening adders is 18% (which corresponds to a speedup of 21%), for the U-CLA architecture. Observe, however, that the other hardened adder architectures benefit circuit speed almost as much.

5.4.5 Simple vs. High Performance Adder Logic

An FPGA architect must choose between smaller, more flexible, slower adders vs. larger, less flexible, faster adders. The delays in the second column of Table 5.7 shows that, on average, the two ripple architectures have 17% more delay than the two carry-lookahead architectures for a 32-bit addition. For the application circuits, however, the ripple architectures average only 1.8% more delay than the CLA architectures. We conclude that the benefit of a very fast adder for long word-length additions is greatly diluted by the presence of all the logic surrounding adders in complete designs – indeed, for some circuits, addition and subtraction are not on the critical path.

Table 5.9: QoR of the VTR+ benchmarks on different carry chain architectures. Values are the geometric mean of VTR+ circuits normalized to the soft adder architecture.

Arch	Area	Area-Delay Product	Min W	Num CLB	Per CLB Area
Ripple	1.042	0.877	0.957	1.029	1.042
U-Ripple	1.034	0.879	0.920	1.031	1.052
CLA	1.055	0.884	0.943	1.037	1.053
U-CLA	1.045	0.861	0.925	1.035	1.063

Table 5.9 shows the quality of results (QoR) for each of the architectures normalized to the soft logic architecture (a circuit-by-circuit breakdown of area results is provided in Appendix B). The columns from left to right are the architecture, the total soft logic area including routing, area-delay product, minimum channel width, number of used soft logic blocks, and area of each soft logic block. Values for area, area-delay product, minimum channel width, and number of used soft logic blocks are geometric averages across all application benchmarks, normalized to the soft adder architecture. Area of each soft logic block is measured directly from the architecture and normalized to the soft adder architecture. The CLA architecture increases area slightly (by between 1% and 2%) but cuts delay by roughly the same amount, leading to an area-delay product that is very close to that of the ripple architectures.

On these complete circuits, the results reaffirm the importance of hard adders but show that different hard adder granularities (1-bit ripple or 4-bit CLA) remain reasonable architectural choices. This is an unexpected result, as Table 5.2 and Table 5.3 show markedly different area and delay characteristics between 1-bit and 4-bit hard adders, respectively. One would normally expect that architectures with 1-bit adders would result in smaller circuits that are also slower, yet the area and delay results on complete circuits exhibit this trend only very weakly.

5.4.6 Balanced vs. Unbalanced

We now consider how best to integrate the LUT and arithmetic circuitry described in Section 5.3. The balanced approach (shown in Figure 5.2) of splitting the 6-LUT into two 5-LUTs, where each 5-LUT drives a different adder input has good symmetry. The unbalanced approach (shown in Figure 5.3) of using the 6-LUT to drive one adder input and a small mux to select BLE input pins for the other adder input offers richer LUT functionality feeding the adder input (six pins compared to five for the balanced case) but worse symmetry. It is thus unclear which of these two approaches is better. Note also that commercial FPGAs differ in their approach: Altera's Stratix V FPGAs [7] support a balanced style, while Xilinx's Virtex7 FPGAs [98] allow both unbalanced and balanced styles.

The third column of Table 5.8 shows the normalized delay values for each of the different architectures. The delay of the U-Ripple architecture is approximately the same as that of the Ripple architecture. The delay of the U-CLA architecture is 1.4% faster than the CLA architecture. From these results, we conclude that balanced and unbalanced architectures achieve approximately the same overall delay.

Table 5.9 shows the QoR for each of the architectures normalized to the soft logic architecture. The balanced and unbalanced architectures require virtually the same CLB count, indicating that the packer can fill both architectures with roughly the same amount of logic per CLB, despite the fact that the balanced architectures can use a LUT on each input of an adder instead of only one input. Interestingly, the unbalanced architectures require a channel width that is approximately 2-3% lower, on average. This is due to the fact that the unbalanced architecture can use all 6 inputs of a BLE when in adder mode,

Table 5.10: QoR for architectures with soft inter-CLB links. Values are the geometric mean of VTR+ circuits normalized to the equivalent architecture with dedicated inter-CLB links.

Arch	32-bit Add Delay	VTR+ Delay	VTR+ Area	VTR+ Area-Delay Product
Ripple	1.805	1.046	0.997	1.043
U-Ripple	1.844	1.041	1.000	1.040
CLA	1.923	1.033	1.001	1.034
U-CLA	1.950	1.021	0.995	1.016

while the balanced architectures can use only 5 – the packer has more freedom on what to pack with the adder in the unbalanced architecture and reduces the number of signals to route between clusters. The net impact is that while the unbalanced architectures require slightly more logic area due to their extra 2:1 mux per BLE, they reduce overall area by 1% by reducing the required amount of inter-cluster routing.

5.4.7 Utility of Inter-CLB Carry

Dedicated carry links between logic blocks improve the speed of long adders significantly, as shown in Figure 5.4, but their use constrains the placement engine to keep long adders in a fixed relative placement, which may lengthen the wiring between other blocks. Table 5.10 compares the QoR of architectures with soft inter-CLB carry links (i.e. routed using the general-purpose interconnect) normalized to their corresponding architectures with hard inter-CLB carry links. The first column gives the architecture name. The second column shows normalized delays for the 32-bit addition micro benchmark. The next three columns show the normalized geometric mean of delay, area, and area-delay product over the VTR+ benchmarks. Using soft inter-CLB links increases the delay of a 32-bit adder by 88%, on average, across the hard adder architectures, but increases the delay of the VTR+ designs by only 3.5%. The area cost of hard inter-CLB carry is negligible, as little hardware needs to be added to support them, and their use does not significantly increase the required inter-CLB channel width, despite the constraint they create on the placement engine. A detailed circuit-by-circuit breakdown of application circuit area and delay values for the different soft inter-logic block carry architectures are provided in Appendix B.

We expect that the impact of hard inter-CLB carry links is a strong function of the number of adder bits per logic block. Fewer adder bits per block means more inter-CLB links are required for an addition of a given size, which in turn may have a bigger impact on delay. Therefore, we believe that architectures with 4 adder bits per logic block (e.g. Virtex 7 [98]) will benefit more from hard inter-CLB links than architectures with 20 bits per block (e.g. Stratix V [7]).

5.4.8 Circuit-by-Circuit Breakdown

Table B.9 provides a circuit-by-circuit breakdown comparing the U-CLA and Soft architectures. The columns from left to right are the benchmark name followed by the ratio of the U-CLA/Soft values for critical path delay, the total soft logic area including routing, and the number of LUTs on the critical path. The last column is the number of (4-bit) hard adders on the critical path for the U-CLA architecture. All values are obtained after routing at 1.3x min W. On average, the delay of the circuits is reduced by 18% and the critical path LUT depth is cut by 46%, but there are 3 distinct classes of circuits that show markedly different behaviour.

Table 5.11: Circuit-by-circuit breakdown comparing the U-CLA architecture to the Soft architecture.

Circuit	Delay	Area	LUTs on Crit Path	CLA cout on Crit Path
arm_core	0.925	1.067	0.875	1
mcml	0.567	1.104	0.379	43
or1200	0.670	1.083	0.148	3
sha	0.549	1.023	0.250	6
stereovision2	0.605	0.784	0.091	5
boundtop	0.998	1.030	0.889	0
LU8PEEng	0.781	1.057	0.642	0
LU32PEEng	0.744	0.958	0.654	0
mkSMAdapter4B	0.959	1.129	0.857	0
bgm	1.068	1.098	0.969	0
blob_merge	0.939	1.060	0.944	0
raygentop	0.971	1.112	N/A	0
stereovision0	1.000	1.044	3.000	0
stereovision1	1.049	1.145	N/A	0
geomean	0.824	1.045	0.561	–
stdev	0.186	1.044	0.760	–

For the top 5 circuits, hard adders are on the critical path, and we obtain a large delay reduction of 35% (a 54% speed-up). The next 4 circuits (boundtop, LU8PEEng, LU32PEEng, and mkSMAdapter4B) have reductions in the LUT depth on the critical path of more than 10% when targeting the U-CLA architecture, even though no hard adders occur on their critical path. This indicates that adder logic was likely timing critical in the Soft architecture[1], but has sped up enough to move off the critical path in the U-CLA architecture. While these 4 circuits have an average LUT depth that is 25% lower when targeting U-CLA vs. Soft, the average delay reduction across the 3 designs is only 14%. We believe this illustrates a trade-off when hard carry chains are added to an FPGA: by limiting the flexibility of the packer and placer, the carry chains have increased the average routing delay per LUT level on non-adder paths, and this costs some of the speed gain one would expect from reducing the logic on the critical path with hard adders. Finally, there are five circuits where the LUT depth is not significantly reduced and where there is not a significant delay reduction, indicating adders were not very timing-critical in even the Soft architecture. Two of these circuits (raygentop and stereovision1) have hard multipliers as their critical paths so they show very little variation in speed vs. carry architecture, as one would expect. The critical path for stereovision0 contains more LUTs in the U-CLA architecture than in soft but differences in inter-logic block routing delay negated the delay penalty from increased LUT depth resulting in almost the same delay for both architectures.

5.5 Conclusions

The work in this chapter covered a broad range of different implementations of hard adders and carry chains within a soft logic block. We show, using the infrastructure described in Chapter 3 and packing algorithm described in Chapter 4, that different hard adder and carry chain architectures show very similar area and delay values on real applications despite significant differences on microbenchmarks. We conclude that hardened adders provide a speed up of approximately 20% for an area penalty of

[1] Ideally we would examine the Soft implementation of a design to directly determine if its critical path included addition, but as ABC does not preserve node names, we cannot trace LUTs back to specific HDL.

approximately 5% resulting in an overall area-delay product reduction of approximately 14%.

Chapter 6

Conclusions and Future Work

This chapter summarizes the main conclusions and contributions of this thesis, and suggests avenues for future research.

6.1 VTR: FPGA Architecture Exploration Infrastructure

The rising software development effort required to explore increasingly complex FPGA architectures drives the need for a flexible FPGA CAD flow that can target a broad space of architectures, with little or no code modifications. We presented an open source FPGA architecture and CAD exploration infrastructure, called Verilog-to-Routing (VTR), that minimizes that software development effort by accepting a flexible *description* of the architecture as input. The main contributions of the VTR project are as follows:

- A new comprehensive FPGA architecture exploration infrastructure with real benchmarks, tuned architecture files, a full CAD flow from Verilog elaboration to routing, regression tests, and documentation.
- Open-source CAD support for modern FPGA architectural features such as configurable memories, fracturable multipliers, hard adders, carry chains, fracturable LUTs, and depopulated crossbars.
- Regular tracking of software quality so that all components stay compatible with each other and so that quality of results are either preserved or improved.
- Enabling new research: VTR 6.0 has accumulated 700 unique downloads, received 89 citations, and served as the base infrastructure for many research papers in the past two and a half years.
- Enabling commercial projects: VTR was/is used as part of the commercial CAD infrastructure within Texas Instruments and also by a new FPGA startup, Efinix.

Several individuals from multiple research groups have worked on VTR. As the one responsible for the VTR system as a whole, I oversaw and/or contributed to many different parts of the project [80] [59] [69].

6.1.1 Future Work

We believe that FPGA architecture exploration remains a rich area for future research. The generic modelling of fixed-function logic in VTR opens up the exploration of specialized hard IP that would have been more difficult to study in the past. Architects interested in targeting high data bandwidth applications may want to model embedded ethernet transceivers, PCI Express, or other high speed data links in their architectures. Architects interested in exploring FPGA clock networks may want to add embedded PLLs then modify VTR to understand clock trees. Architects speculating on promising new hard blocks, such as hardened crossbars [38], now have access to measurements using a full CAD flow.

FPGA CAD and architecture are strongly coupled. As the space of architectures explored increases, so too will the need for better CAD to target those architectures. We anticipate that researchers will continue to use VTR as a baseline for future CAD studies.

As technology scaling increases the logic capacity of FPGAs, computation specified down to the bit level may become increasingly less scalable. At the other extreme, arrays of full processors offer a much higher level of abstraction, but incur a large delay/power overhead because of the need to process instructions (whereas an FPGA can encode computation directly in the datapath circuitry). Coarse-grained reconfigurable architectures (CGRAs) occupy a vast middle spectrum between fine-grained FPGAs on one end, and an array of full processors on the other. A well-designed CGRA might deliver the best of both of these worlds. Also, a good architecture may not necessarily be limited to just the level of coarsening. In much the same way as modern consumer computer systems pair a processor with a GPU, it is interesting to explore heterogeneous chips that include processors, fine-grained FPGA logic, and specialized compute units that best leverage the unique strengths of each. VTR may be the right vehicle to modify to explore important aspects of these diverse areas.

There remains much work to do in every major component of VTR and in every major stage of the VTR CAD flow. Gathering the latest benchmarks and making them compatible with the VTR flow will remain an on-going task. Creating new architecture description files is currently very challenging, in a large part because accurate area and delay values of various components requires transistor-level design of the full architecture. Thus, integrating automated transistor design for new, hypothetical FPGA architectures [18] is becoming increasingly important because the time required to hand-optimize designs at the transistor level has risen with the increasing complexity of FPGA architectures.

The different stages of the VTR CAD flow itself also have potential for improvement. The Odin II Verilog elaborator needs better language support. Kent's team has devoted much effort improving language support in Odin II [37] [59] but language features used in real designs are quite diverse so many used features are not yet covered. Hence, it remains labour-intensive to covert benchmarks to a form that can be accepted by Odin II, which in turn limits how many benchmarks we can provide in VTR. ABC logic synthesis needs to understand logic that crosses hard logic boundaries. Currently, ABC treats all hard logic as black boxes and hence misses important optimizations.

The AAPack packer needs to broaden the space of logic blocks that it can target well. For example, the packer takes a much longer time than it should when targeting complex bus-based arithmetic blocks. Placement needs to be improved to understand region constraints. Minimum channel width routing needs to be made faster, as it currently occupies more than half the total runtime of experiments. Since the later stages of the CAD flow have more information than the earlier stages, placement should be made capable of changing packing decisions and inter-logic block routing should be made capable of changing intra-logic block routing. The timing analyzer in VTR should do hold time analysis. Finally,

although VTR has power analysis, we currently do not do power optimization. Power optimization would be an interesting field to explore.

A complete, end-to-end, comparison between VTR and commercial tools would be very interesting. Currently, there have been studies [69] [32] comparing certain stages of the VTR CAD flow to corresponding stages of a commercial flow but several issues have prevented a full comparison. These issues include benchmark compatibility problems, differences in architecture features supported, and challenges in keeping logic models consistent.

6.2 Architecture-Aware Packing for FPGAs

The need to explore new complex logic block architectures, with few to no modifications to source code, requires that the packing stage of the CAD flow accept a flexible description of the architecture as input. We formally defined this expanded notion of packing and showed that flexibility in the description of an architecture results in a placement-and-routing subproblem, within each logic block, during packing. We then described an architecture-aware packing algorithm to solve this new packing problem.

The AAPack algorithm greedily fills logic blocks one netlist block at a time. It handles the intra-logic block placement subproblem using a greedy best-fit approach. It handles the intra-logic block routing subproblem using the Pathfinder [65] negotiated congestion algorithm.

Indiscriminate use of intra-logic block detailed routing to check packing feasibility can produce very long runtimes so three techniques were introduced to avoid detailed routing on well-understood interconnect. The first technique, interconnect-aware pin counting, is a filter that approximates the routing problem with a simpler counting problem. Interconnect-aware pin counting groups pins into pin classes based on intra-logic block interconnect then applies counting arguments to discover and reject impossible-to-route intermediate solutions without the use of full detailed routing. The second technique, speculative packing, assumes routing will be successful and attempts to only route once at the end of packing, using interconnect-aware pin couting as the only check for routability of intermediate solutions. Speculative packing reduces pack time by approximately 3-fold on a set of large, realistic, VTR benchmarks on an architecture with modern architectural features. The third technique, pre-packing, is meant to handle architectural primitives connected together by inflexible links (such as carry chains, LUT/FF pairs, multiply-add operations, etc.). Pre-packing groups parts of the netlist together before packing based on hints in the architecture description file. These groups then stay together as one unit during packing. Pre-packing results in approximately 2-fold speedup and improved logic density. Taken together, these techniques result in 5-fold speedup and better quality of results than the same algorithm without these techniques.

Our experiments and results demonstrate that the packer can target logic blocks with modern features such as soft logic blocks with fracturable LUTs, depopulated crossbars, and carry chains. We show, in an experiment with various levels of interconnect flexibility in a soft logic block, that AAPack increases computational effort with increasing interconnect difficulty. We also demonstrated that AAPack, a general purpose packer, can produce marginally better minimum channel width and critical path delay against a specialized packer, T-VPack [64], for an order of magnitude more runtime. This runtime difference remains a small fraction of total flow runtime. However, if used in a general CAD flow outside of minimum channel width search, AAPack runtime remains high as its runtime lies in between fixed channel width routing time and placement time.

The main contributions in packing are as follows:

- A new formal description of the generalized packing problem.
- Important enhancements to an existing packing algorithm, AAPack, that enable timing-driven optimizations and improve robustness, quality, and runtime of the baseline algorithm [57] [80] [62].
- New techniques to allow AAPack to adjust computational effort based on architectural complexity [62].

6.2.1 Future Work

There are several interesting new avenues for future research in architecture-aware packing. As mentioned earlier in Section 2.3.1 on traditional packing, packing for simple logic blocks has advanced substantially since T-VPack. It is future work to determine which of those techniques can be generalized to architecture-aware packing and which techniques are limited to only simple logic blocks. In the event that some of those techniques are not generalizable, it would be interesting to see if the packing algorithm can automatically detect when to employ a particular specialized technique, in much the same way as how AAPack can adapt computational effort based on interconnect complexity.

Currently, in AAPack, the architect must provide hints in the architecture description file on what molecules the pre-packer should form. This limitation increases the learning curve needed to use VTR efficiently. In the future, the pre-packer should automatically figure out these hints, which should simplify the specification of new architectures.

The resource balancing subproblem arises when different types of logic blocks can implement one particular atom because the packing algorithm must make a choice on what type of logic block to use. For example, if an architecture contains configurable block RAM of different sizes, then the packer must find an appropriate mix of block RAMs to assign the different user memories to [40]. The packer should choose in such a way that the final packed solution fits a given architecture (if the size of the architecture is fixed) or minimizes the size of the final architecture. The resource balancing problem is left to future work.

Ahmed et al. [4] showed that for a commercial FPGA with RAM blocks and arithmetic blocks, packing should assign atoms in such a way that connections between memories and arithmetic can be easily aligned during inter-logic block placement to obtain better routability and performance. It is interesting future work to generalize this approach, so that packing can automatically make use of information about the inter-logic block architecture.

When AAPack performs one-by-one packing, it re-routes all nets within the logic block for every intermediate solution. Runtime may be improved by only routing nets that change first, reserving the full detailed routing for cases where congestion is difficult to resolve.

6.3 Architecture Study on Hard Adders and Carry Chains

Addition and subtraction are common operations in digital circuit design. Wide implementations of these functions are very slow in soft logic, so it makes sense to provide hardened support for these functions. Despite being found in commercial FPGAs for decades, there has been little published work

demonstrating their effectiveness. We investigated the benefit of a broad range of different implementations of hard adders and carry chains on FPGAs, for both pure adder microbenchmarks, as well as large full applications. Furthermore, this study uses the new capabilities of packing and the VTR CAD flow, which demonstrates the usefulness of our software infrastructure for architecture exploration.

We show that different hard adder and carry chain architectures show very similar area and delay values on real applications, despite significant differences on microbenchmarks. We conclude that hardened adders provide a speed up of approximately 20% for an area penalty of approximately 5%, resulting in an overall area-delay product reduction of approximately 14%.

The main contributions of this architecture study are as follows [61]:

- Quantified effects on area and delay from hard adders and carry chains in FPGAs.
- Demonstration of major differences in conclusions between microbenchmarks and full applications.
- Showed that several details in the implementation of hard adders and carry chains matter little in real applications, despite larger differences on microbenchmarks.

6.3.1 Future Work

There remains much future work to explore on the topic of hard adders and carry chains. In terms of architecture, the interaction between fracturable LUTs and hard adders is interesting as it adds another dimension to the architecture space. In terms of benchmarks, we only explored pure adder benchmarks versus general applications, the space of arithmetic-heavy benchmarks (such as DSP applications) was not specifically targeted. We suspect that for these applications, the benefits of hard adders and carry chains would be in between pure adder results and the VTR+ benchmark results.

In terms of CAD, the most pressing issue is the lack of good logic synthesis when adders are used. Currently, VTR relies exclusively on Odin II to remove redundant logic during Verilog elaboration because ABC is unable to understand logic within hard adders. Thus, optimization opportunities that are revealed during logic synthesis are not exploited. Ideally, ABC would be upgraded to understand the logic within hard adders and optimize the soft logic accordingly.

Bibliography

[1] Ruth Ablett, Ehud Sharlin, Frank Maurer, Jörg Denzinger, and Craig Schock. Buildbot: Robotic Monitoring of Agile Software Development Teams. In *IEEE Robot and Human Interactive Communication*, pages 931–936, 2007.

[2] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE TVLSI*, 12(3):288–298, 2004.

[3] Norrudin Ahmed. Summer Undergraduate Intern at the University of Toronto. Personal communication.

[4] Taneem Ahmed, Paul Kundarewich, Jason Anderson, Brad Taylor, and Rajat Aggarwal. Architecture-Specific Packing for Virtex-5 FPGAs. In *ACM FPGA*, pages 5–13, 2008.

[5] Christoph Albrecht. IWLS 2005 Benchmarks. In *IWLS*, 2005.

[6] Altera Co. Stratix IV Device Family Overview. `http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf`, 2009.

[7] Altera Co. Logic Array Blocks and Adaptive Logic Modules in Stratix V Devices. `http://www.altera.com/literature/hb/stratix-v/stx5_51002.pdf`, 2013.

[8] Altera Co. Variable Precision DSP Blocks in Stratix V Devices. `http://www.altera.com/literature/hb/stratix-v/stx5_51004.pdf`, 2013.

[9] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, Massachusetts, 1999.

[10] Huimin Bian, Andrew C Ling, Alexander Choong, and Jianwen Zhu. Towards scalable placement for fpgas. In *ACM FPGA*, pages 147–156, 2010.

[11] E. Bozorgzadeh, S. Memik, X. Yang, and M. Sarrafzadeh. Routability-driven Packing: Metrics and Algorithms for Cluster-Based FPGAs. *Journal of Circuits Systems and Computers*, 13:77–100, 2004.

[12] Elaheh Bozorgzadeh, Seda Memik, and Majid Sarrafzadeh. RPack: Routability-driven Packing for Cluster-Based FPGAs. In *ACM ASP-DAC*, pages 629–634, 2001.

[13] Alexander Brant and Guy GF Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *IEEE FCCM*, pages 93–96, 2012.

[14] Yu Cao. Berkeley Predictive Technology Model, 2008.

[15] C.H. Ho, C.W. Yu, P.H.W. Leong, W. Luk and S.J.E. Wilton. Floating-Point FPGA: Architecture and Modeling. *IEEE TVLSI*, 17(2):1709–1718, 2009.

[16] Daniel Chen. Summer Undergraduate Intern at the University of Toronto. Personal communication.

[17] Doris Chen, Kristofer Vorwerk, and Andrew Kennings. Improving Timing-Driven FPGA Packing with Physical Information. *IEEE FPL*, pages 117–123, 2007.

[18] Charles Chiasson and Vaughn Betz. COFFE: Fully-Automated Transistor Sizing for FPGAs. In *IEEE FPT*, pages 34–41, 2013.

[19] Altera Co. Logic Array Blocks and Adaptive Logic Modules in Stratix IV Devices. `http://www.altera.com/literature/hb/stratix-iv/stx4_siv51002.pdf`, 2011.

[20] Jason Cong and Yuzheng Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE TCAD*, 13(1):1–12, 1994.

[21] Jason Cong, John Peck, and Yuzheng Ding. RASP: A General Logic Synthesis System for SRAM-based FPGAs. In *ACM FPGA*, pages 137–143, 1996.

[22] Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[23] Wenyi Feng. K-Way Partitioning Based Packing for FPGA Logic Blocks Without Input Bandwidth Constraint. In *IEEE FPT*, pages 8–15, 2012.

[24] Wenyi Feng, Jonathan Greene, Kris Vorwerk, Val Pevzner, and Arun Kundu. Rents Rule Based FPGA Packing for Routability Optimization. In *ACM FPGA*, pages 31–34, 2014.

[25] J.B. Goeders and S.J.E. Wilton. VersaPower: Power Estimation for Diverse FPGA Architectures. In *IEEE FPT*, pages 229–234, 2012.

[26] Jonathan Greene et al. A 65nm Flash-Based FPGA Fabric Optimized for Low Cost and Power. In *ACM FPGA*, pages 87–96, 2011.

[27] Julian Hartig, Guillermo Paya-Vaya, and Holger Blume. Design and Analysis of a Structured-ASIC Architecture for Implementing Generic VLIW-SIMD Processors. In *ICT.OPEN*, 2013.

[28] S. Hauck, M. Hosler, and T. Fry. High-Performance Carry Chains for FPGA's. *IEEE TVLSI*, 8(2):138–147, 2000.

[29] J. He and J. Rose. Technology Mapping for Heterogeneous FPGAs. *ACM FPGA*, 1994.

[30] William K. C. Ho and Steven J. E. Wilton. Logical-to-physical memory mapping for fpgas with dual-port embedded arrays. In *IEEE FPL*, pages 111–123, London, UK, 1999. Springer-Verlag.

[31] H-C Hsieh et al. Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays. In *IEEE CICC*, pages 31–2, 1990.

[32] Eddie Hung, Fatemeh Eslami, and Steven JE Wilton. Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices. In *IEEE FCCM*, pages 45–52, 2013.

[33] Xilinx Inc. The Programmable Gate Array Design Handbook. 1986.

[34] Xilinx Inc. Virtex-6 FPGA Configurable Logic Block User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`, 2012.

[35] Xilinx Inc. Virtex 2.5 V Field Programmable Gate Arrays. `http://www.xilinx.com/support/documentation/data_sheets/ds003.pdf`, 2013.

[36] Open Source Initiative et al. The MIT License, 2006.

[37] P. Jamieson, K Kent, F. Gharibian, and L. Shannon. Odin II-An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *IEEE FCCM*, pages 149–156, 2010.

[38] Peter Jamieson and Jonathan Rose. Architecting Hard Crossbars on FPGAs and Increasing Their Area Efficiency With Shadow Clusters. In *IEEE FPT*, 2007.

[39] S. Jang, B. Chan, K. Chung, and A. Mishchenko. WireMap: FPGA Technology Mapping for Improved Routability. In *ACM FPGA*, pages 47–55, 2008.

[40] Pradip K Jha and Nikil D Dutt. Library Mapping for Memories. In *IEEE EDTC*, pages 288–292, 1997.

[41] Kurt Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Papers on Twenty-Five Years of Electronic Design Automation*, pages 617–624. ACM, 1988.

[42] Ian Kuon and Jonathan Rose. Area and Delay Trade-Offs in the Circuit and Architecture Design of FPGAs. In *ACM FPGA*, pages 149–158, 2008.

[43] Ian Kuon and Jonathan Rose. Automated transistor sizing for fpga architecture exploration. In *ACM/IEEE DAC*, pages 792–795, 2008.

[44] D. Lampret. OpenRISC 1200 IP Core Specification. *OpenCores*, 2001.

[45] Lattice Semiconductor. LatticeECP3 Family Handbook. `http://d12lxohwf1zsq3.cloudfront.net/documents/HB1009.pdf`, 2013.

[46] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. RapidSmith: Do-It-Yourself CAD tools for Xilinx FPGAs. In *IEEE FPL*, pages 349–355, 2011.

[47] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *IEEE FPT*, pages 41–48, 2004.

[48] Guy Lemieux and David Lewis. *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, 2004.

[49] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, et al. The Stratix II Logic and Routing Architecture. In *ACM FPGA*, pages 14–20, 2005.

[50] D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, et al. The Stratixπ Routing and Logic Architecture. In *ACM FPGA*, pages 12–20, 2003.

[51] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural Enhancements in Stratix V. In *ACM FPGA*, pages 147–156, 2013.

[52] David Lewis et al. Architectural Enhancements in Stratix-III and Stratix-IV. In *ACM FPGA*, pages 33–42, 2009.

[53] J. Lin, D. Chen, and J. Cong. Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization. In *ACM/IEEE DAC*, pages 472–477, 2006.

[54] A.C. Ling, J. Zhu, and S.D. Brown. Scalable Synthesis and Clustering Techniques Using Decision Diagrams. *IEEE TCAD*, 27(3):423, 2008.

[55] Suya Liu. Summer Undergraduate Intern at the University of Toronto. Personal communication.

[56] Jason Luu. A Hierarchical Description Language and Packing Algorithm for Heterogeneous FPGAs. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 2010.

[57] Jason Luu, Jason Anderson, and Jonathan Rose. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *ACM FPGA*, pages 227–236, 2011.

[58] Jason Luu, Jeff Goeders, Tim Liu, Alexander Marquardt, Ian Kuon, Jason Anderson, Jonathan Rose, and Vaughn Betz. VPR User's Manual (Version 7.0). `http://code.google.com/p/vtr-verilog-to-routing/downloads/list`, 2013.

[59] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Miadi Nasr Nasartschuk, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. In *ACM TRETS*, May 2014.

[60] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. VPR 5.0: FPGA CAD and Architecture Exploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling. In *ACM FPGA*, pages 133–142, 2009.

[61] Jason Luu, Conor McCullough, Sen Wang, Safeen Huda, Yan Bo, Charles Chiasson, Kenneth Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. On Hard Adders and Carry Chains in FPGAs. In *IEEE FCCM*, 2014.

[62] Jason Luu, Jonathan Rose, and Jason Anderson. Towards Interconnect-Adaptive Packing for FPGAs. In *ACM FPGA*, pages 21–30, 2014.

[63] Shawn Malhotra, Terry P Borer, Deshanand P Singh, and Stephen Dean Brown. The Quartus University Interface Program: Enabling Advanced FPGA Research. In *IEEE FPT*, pages 225–230, 2004.

[64] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. *ACM FPGA*, pages 37–46, 1999.

[65] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *ACM FPGA*, pages 111–117, 1995.

[66] A. Mishchenko et al. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/alanmi/abc`, 2009.

[67] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *ACM DAC*, pages 532–535, 2006.

[68] Fernando Mujica. Manager in the Research and Development Business Group at Texas Instruments Inc. Personal communication.

[69] Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. TITAN: Enabling Large and Complex Benchmarks in Academic CAD. In *IEEE FPL*, 2013.

[70] Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. In *ACM ISPD*, pages 222–227, 2001.

[71] Tony Ngai. Founder and CTO of Efinix FPGA Startup. Personal communication.

[72] Gang Ni, Jiarong Tong, and Jinmei Lai. A New FPGA Packing Algorithm Based on the Modeling Method for Logic Block. In *IEEE ASICON*, volume 2, pages 877–880, 2005.

[73] Raphael Njuguna and Raj Jain. A Survey of FPGA Benchmarks. Technical report, CSE Department, Washington University in St. Louis, 2008.

[74] Daniele Paladino. Academic Clustering and Placement Tools for Modern Field-Programmable Gate Array Architectures. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 2008.

[75] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. A Novel FPGA Logic Block for Improved Arithmetic Performance. In *ACM FPGA*, pages 171–180, 2008.

[76] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. Efficient Synthesis of Compressor Trees on FPGAs. In *IEEE ASP-DAC*, pages 138–143, 2008.

[77] Kara K. W. Poon, Andy Yan, and Steven J. E. Wilton. A Flexible Power Model for FPGAs. In *IEEE FPL*, pages 312–321. Springer-Verlag, 2002.

[78] Madhura Purnaprajna and Paolo Ienne. A Case for Heterogeneous Technology-Mapping: Soft versus Hard Multiplexers. In *IEEE FCCM*, pages 53–56, 2013.

[79] Jonathan Rose. Hard vs. Soft: The Central Question of Pre-Fabricated Silicon. *IEEE ISMVL*, pages 2–5, 2004.

[80] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *ACM FPGA*, pages 77–86, 2012.

[81] Raphael Rubin. PhD student at University of Pennsylvania. Personal communication.

[82] Jeff Rudolph. Senior Software Engineer at Texas Instruments. Personal communication.

[83] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical report, EECS Department, University of California, Berkeley, 1992.

[84] A. Sharma, S. Hauck, and C. Ebeling. Architecture-Adaptive Routability-Driven Placement for FPGAs. In *IEEE FPL*, pages 427–432, 2005.

[85] Amit Singh, Ganapathy Parthasarathy, and Malgorzata Marek-Sadowksa. Efficient Circult Clustering for Area and Power Reduction in FPGAs. *ACM TODAES*, 7(4):643–663, 2002.

[86] Alastair M Smith, George A Constantinides, and Peter YK Cheung. FPGA Architecture Optimization Using Geometric Programming. *IEEE TCAD*, 29(8):1163–1176, 2010.

[87] Andrew Somerville and Kenneth B Kent. Improving Memory Support in the VTR Flow. In *IEEE FPL*, pages 197–202, 2012.

[88] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: Towards an Open-Source Tool Flow. In *ACM FPGA*, pages 41–44, 2011.

[89] Neil Joseph Steiner. *Autonomous Computing Systems*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, United States, 2008.

[90] Jordan S Swartz, Vaughn Betz, and Jonathan Rose. A Fast Routability-Driven Router for FPGAs. In *ACM FPGA*, pages 140–149, 1998.

[91] Benjamin Tseng, Jonathan Rose, and Stephen Brown. Improving FPGA Routing Architectures using Architecture and CAD Interactions. In *IEEE ICCD*, pages 99–104, 1992.

[92] K. Wang, M. Yang, L. Wang, X. Zhou, and J. Tong. A Novel Packing Algorithm for Sparse Crossbar FPGA Architectures. In *IEEE ICSICT*, pages 2345–2348, 2008.

[93] Lingli Wang. Professor at Fudan University in Shanghai. Personal communication.

[94] S.J.E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.

[95] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing FPGA vs. Custom CMOS and The Impact on Processor Microarchitecture. In *ACM FPGA*, pages 5–14, 2011.

[96] Nam-Sung Woo. Revisiting the Cascade Circuit in Logic Cells of Lookup Table Based FPGAs. In *ACM FPGA*, pages 90–96, 1995.

[97] Xilinx Inc. Xilinx Virtex-6 Family Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, 2009.

[98] Xilinx Inc. 7 Series FPGAs Configurable Logic Block User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`, 2013.

[99] Shanzhen Xing and William Wh Yu. FPGA Adders: Performance Evaluation and Optimal Design. *IEEE Design & Test of Computers*, 15(1):24–29, 1998.

[100] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. *MCNC, Jan*, 1991.

[101] Grace Zgheib, Liqun Yang, Zhihong Huang, David Novo, Hadi Parandeh-Afshar, Haigang Yang, and Paolo Ienne. Revisiting And-Inverter Cones. In *ACM FPGA*, pages 45–54, 2014.

Appendix A

VTR 7.0 Public Architectures

This section contains reference tables on the most recent (VTR 7.0) publicly released set of architecture files [59].

The VTR 7.0 release provides a series of sample architecture files in the *vtr_flow/arch* folder of the release package. The *timing* folder contains the key architecture files for the release. These architecture files have reasonable area-delay values. The other folders contain architecture files that illustrate certain architectural features and *do not* have reasonable area-delay values.

Table A.1 describes the key properties of an FPGA architecture. All subsequent tables characterize different architectures using these properties.

Table A.2 describes the timing-driven architectures of the VTR release. The *k6* series architecture files are variations of the EArch architecture, described in Section 3.2.2, which is based on a commercial Stratix IV architecture [6]. A new user of VTR should begin with these architecture description files. The *k4_N4_90nm* architecture file is a simple architecture file, containing only soft logic, sampled from the iFAR FPGA architecture repository [43]. The *fpu* series architectures are two proof-of-concept architectures for exploring FPGAs targeting floating-point applications. The *hard_fpu* architecture contains embedded, configurable, floating-point cores. The *soft_fpu* architecture contains soft logic only.

Table A.3 describes the other architecture description files that serve as proof-of-concepts for specific

Table A.1: Properties that describe an architecture.

Property	Value
Process	Process technology
K	Size of (largest) LUT.
N	Number of (fracturable) LUTs.
I	Number of inputs per soft logic cluster.
Frac LUTs	Whether or not a LUT can be fractured into smaller LUTs with shared inputs.
Carry Chain	Whether or not a soft logic block contains carry chains.
FI	Number of inputs used by fracturable LUT in dual-LUT mode.
RAM	Block RAM type.
Multiplier	Block multiplier type.
Fc_{in}	Fraction of tracks that may drive a logic block input pin.
Fc_{out}	Fraction of tracks a logic block output pin may drive.
L	Number of soft logic blocks a wire segment spans.
Power Models	Whether or not architecture contains power models.

Table A.2: Core timing-driven architectures of the VTR release.

Property	Value
k6 series	
Process	40nm
K	6
N	10
I	40
Frac LUTs	Optional
Carry Chain	Optional 1-bit ripple
FI	5 (if applicable)
RAM	Optional, configurable 32 Kb
Multiplier	Optional, fracturable 36x36
Fc_{in}	0.15
Fc_{out}	0.1
L	4
Power Models	Yes
k4_N4	
Process	90nm
K	4
N	4
I	10
Frac LUTs	None
Carry Chain	None
RAM	None
Multiplier	None
Fc_{in}	0.15
Fc_{out}	0.25
L	1
Power Models	None
fpu series	
Process	130nm
K	4
N	4
I	10
Frac LUTs	None
Carry Chain	Optional
RAM	None
Multiplier	None
Fc_{in}	1.0
Fc_{out}	0.25
L	4
Power Models	None

Table A.3: Description of the purpose of the other architectures files.

Architecture Series	Purpose
Titan	Allow the use of a commercial front-end, Quartus II, to feed into VPR via the Titan flow [69].
Power	Proof-of-concept on how to describe power for a series of fracturable LUT architectures.
no_timing	Proof-of-concept on how to describe different memory and fracturable LUT architectures.

features. They *do not* contain reasonable area-delay values.

Appendix B

Data Tables of the Adder Architecture Study

This section contains data tables that show the quality of results for the large VTR application circuits on the adder architectures in chapter 5. From left to right, the columns of these tables are: the circuit name, critical path delay in ns at 1.3 times minimum channel width, minimum channel width, number of soft logic blocks, and total soft logic area in minimum width transistor areas. Total soft logic area is measured as the total number of soft logic blocks times the area of one soft logic block and one tile of interconnect.

Table B.1: Quality of results of the soft adder architecture.

Circuit	Delay	Min W	Num CLB	Area
arm_core	16.01	112	1713	3.55E+07
bgm	23.92	120	3910	8.34E+07
blob_merge	8.40	82	753	1.42E+07
boundtop	5.39	58	381	6.56E+06
LU8PEEng	91.43	126	2742	5.92E+07
LU32PEEng	93.80	202	9166	2.42E+08
mcml	75.84	98	12700	2.50E+08
mkSMAdapter4B	5.39	66	248	4.43E+06
or1200	11.04	82	382	7.22E+06
raygentop	4.57	72	274	5.00E+06
sha	11.21	66	285	5.09E+06
stereovision0	3.56	62	1743	3.01E+07
stereovision1	5.35	84	1595	3.02E+07
stereovision2	19.59	150	3895	9.06E+07

Table B.2: Quality of results of the balanced ripple adder architecture.

Circuit	Delay	Min W	Num CLB	Area
arm_core	15.96	108	1783	3.75E+07
bgm	24.48	106	4519	9.42E+07
blob_merge	8.30	82	766	1.48E+07
boundtop	5.76	52	381	6.56E+06
LU8PEEng	72.47	124	2858	6.30E+07
LU32PEEng	72.06	184	9538	2.47E+08
mcml	40.07	104	13045	2.71E+08
mkSMAdapter4B	5.46	66	268	4.93E+06
or1200	8.07	86	397	7.85E+06
raygentop	4.44	70	298	5.58E+06
sha	6.35	62	290	5.21E+06
stereovision0	3.58	58	1747	3.09E+07
stereovision1	5.48	102	1610	3.32E+07
stereovision2	12.00	102	3407	7.00E+07

Table B.3: Quality of results of the unbalanced ripple adder architecture.

Circuit	Delay	Min W	Num CLB	Area
arm_core	16.28	112	1776	3.80E+07
bgm	25.18	88	4599	9.04E+07
blob_merge	8.08	82	764	1.49E+07
boundtop	5.45	54	380	6.66E+06
LU8PEEng	73.39	100	2996	6.20E+07
LU32PEEng	70.38	158	9817	2.39E+08
mcml	47.76	104	13090	2.73E+08
mkSMAdapter4B	5.37	64	267	4.86E+06
or1200	7.75	78	390	7.48E+06
raygentop	4.44	70	296	5.58E+06
sha	6.46	62	286	5.17E+06
stereovision0	3.67	58	1742	3.10E+07
stereovision1	5.61	106	1593	3.35E+07
stereovision2	12.01	104	3369	7.04E+07

Table B.4: Quality of results of the carry-lookahead adder architecture.

Circuit	Delay	Min W	Num CLB	Area
arm_core	14.90	106	1791	3.78E+07
bgm	23.61	102	4549	9.42E+07
blob_merge	7.75	86	765	1.51E+07
boundtop	5.57	54	386	6.77E+06
LU8PEEng	68.23	130	2839	6.43E+07
LU32PEEng	70.91	192	9475	2.53E+08
mcml	41.23	100	13167	2.71E+08
mkSMAdapter4B	5.39	66	272	5.04E+06
or1200	7.99	82	400	7.83E+06
raygentop	4.31	68	298	5.54E+06
sha	7.31	58	293	5.27E+06
stereovision0	3.59	54	1761	3.07E+07
stereovision1	5.84	98	1679	3.45E+07
stereovision2	12.51	100	3436	7.09E+07

Table B.5: Quality of results of the unbalanced carry-lookahead adder architecture.

Circuit	Delay	Min W	Num CLB	Area
arm_core	14.81	108	1783	3.79E+07
bgm	25.56	88	4628	9.16E+07
blob_merge	7.88	82	766	1.50E+07
boundtop	5.38	56	381	6.75E+06
LU8PEEng	71.41	102	2996	6.25E+07
LU32PEEng	69.78	142	9981	2.32E+08
mcml	42.98	106	13105	2.76E+08
mkSMAdapter4B	5.17	68	267	5.00E+06
or1200	7.39	84	392	7.82E+06
raygentop	4.44	70	293	5.56E+06
sha	6.15	62	286	5.21E+06
stereovision0	3.56	58	1751	3.14E+07
stereovision1	5.61	104	1637	3.46E+07
stereovision2	11.84	108	3359	7.10E+07

Table B.6: Quality of results of the balanced ripple adder architecture with soft inter-logic block carry links.

Circuit	Delay	Min W	Num CLB	Area
arm_core	16.97	110	1783	3.80E+07
bgm	24.83	106	4519	9.50E+07
blob_merge	8.08	82	766	1.49E+07
boundtop	5.44	54	381	6.66E+06
LU8PEEng	76.34	122	2858	6.32E+07
LU32PEEng	77.74	184	9538	2.49E+08
mcml	50.36	94	13045	2.60E+08
mkSMAdapter4B	5.42	64	268	4.87E+06
or1200	8.19	82	397	7.77E+06
raygentop	4.44	72	298	5.63E+06
sha	7.82	56	290	5.12E+06
stereovision0	3.54	56	1747	3.06E+07
stereovision1	5.48	102	1610	3.35E+07
stereovision2	12.98	94	3407	6.81E+07

Table B.7: Quality of results of the unbalanced ripple adder architecture with soft inter-logic block carry links.

Circuit	Delay	Min W	Num CLB	Area
arm_core	16.79	106	1776	3.77E+07
bgm	24.11	88	4599	9.10E+07
blob_merge	8.12	84	764	1.52E+07
boundtop	5.57	54	380	6.69E+06
LU8PEEng	74.96	102	2996	6.26E+07
LU32PEEng	74.28	152	9817	2.38E+08
mcml	43.24	92	13090	2.62E+08
mkSMAdapter4B	5.20	66	267	4.97E+06
or1200	8.51	84	390	7.79E+06
raygentop	4.57	70	296	5.61E+06
sha	7.69	52	286	5.01E+06
stereovision0	5.02	56	1742	3.07E+07
stereovision1	5.61	102	1593	3.33E+07
stereovision2	11.93	98	3369	6.93E+07

Table B.8: Quality of results of the carry-lookahead adder architecture with soft inter-logic block carry links.

Circuit	Delay	Min W	Num CLB	Area
arm_core	15.67	106	1791	3.81E+07
bgm	23.70	106	4549	9.62E+07
blob_merge	7.84	86	765	1.53E+07
boundtop	5.80	54	386	6.80E+06
LU8PEEng	70.77	126	2839	6.36E+07
LU32PEEng	75.01	192	9475	2.56E+08
mcml	50.82	92	13167	2.64E+08
mkSMAdapter4B	5.33	66	272	5.07E+06
or1200	8.22	82	400	7.88E+06
raygentop	4.44	68	298	5.57E+06
sha	7.94	54	293	5.19E+06
stereovision0	3.57	54	1761	3.09E+07
stereovision1	5.74	98	1679	3.48E+07
stereovision2	11.80	96	3436	6.93E+07

Table B.9: Quality of results of the unbalanced carry-lookahead adder architecture with soft inter-logic block carry links.

Circuit	Delay	Min W	Num CLB	Area
arm_core	14.78	106	1783	3.81E+07
bgm	26.23	86	4628	9.19E+07
blob_merge	7.96	82	766	1.51E+07
boundtop	5.47	56	381	6.79E+06
LU8PEEng	72.55	98	2996	6.22E+07
LU32PEEng	71.85	138	9981	2.33E+08
mcml	42.62	92	13105	2.64E+08
mkSMAdapter4B	5.13	66	267	5.01E+06
or1200	8.41	82	392	7.77E+06
raygentop	4.44	70	293	5.59E+06
sha	6.76	56	286	5.12E+06
stereovision0	3.56	54	1751	3.09E+07
stereovision1	5.48	102	1637	3.45E+07
stereovision2	12.08	98	3359	6.95E+07