A Hierarchical Description Language and Packing Algorithm for Heterogenous FPGAs

by

Jason Luu

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

A Hierarchical Description Language and Packing Algorithm for Heterogenous FPGAs

Jason Luu

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2010

The complexity of Field-Programmable Gate Array (FPGAs) logic blocks have undergone constant evolution to the point where both the basic *soft* logic blocks that implement combinational logic and the fixed-function *hard* blocks contain complex interconnects, hierarchy and modes. The goal of this thesis is to both support that complexity and enable future architecture exploration of even increased complexity and new kinds of hard functionality. To accomplish this, a Computer-Aided Design (CAD) flow that can map a user circuit to an FPGA with these complex blocks is needed. We propose a new language that can describe these complex blocks and a new area-driven tool for the *packing* stage of that CAD flow. The packing stage groups components of a user circuit into the complex blocks available on the FPGA. We conduct experiments to illustrate the quality of the packing tool and to demonstrate the newly-enabled architecture exploration capabilities.

Acknowledgements

I would like to thank my supervisors Jonathan Rose and Jason Anderson for their deep insights and advice on my thesis work and life in general.

I would like to also express gratitude to my parents Can Luu and Huong Do for their support throughout my Master's as well as to my friend Jun Ye for her humour and encouragement.

I would like to thank Kenneth Kent, Ian Kuon, Danny Paladino, and many others in the lab that I worked with. Their feedback and input were helpful and well appreciated.

Lastly, I would like to thank NSERC and the Rogers Scholarship for their financial support in my degree.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are programmable digital integrated circuits that are a widely used media for implementing digital circuits. This is reflected in the annual revenue of the two largest FPGA vendors, Xilinx and Altera, which exceeded $3 billion in 2008 [10] [18]. Programmability is one reason, among others, for the widespread use of FPGAs; however, it comes with a high cost and one aspect of that cost is silicon area. The general-purpose logic fabric that enables programmability on an FPGA uses an average of 35 times more silicon area to implement a digital function compared to implementing the same function directly in silicon using standard cells [24]. Area is a major factor in the cost of a chip and so there is a strong incentive to narrow this area gap. Furthermore, reducing area has the added benefit of improving both power and delay because power is directly correlated with area, and delay can be improved by trading-off area.

FPGA area can be improved by selectively removing some of its programmability, hence reducing some of its cost. One approach on this front is to embed fixed-functionality blocks (also known as hard blocks) such as memories and multipliers into the FPGA logic

fabric. Kuon showed that the area ratio between the specific circuits he implemented on an FPGA and those same circuits implemented directly in silicon using standard cells reduced from 35 to 18 in designs with hard blocks [24]. This is under the condition that the number of hard blocks supplied on the FPGA exactly matches the demand of the user circuit. When the number of supplied hard blocks exceeds demand then the unused hard blocks waste area. But, if too few hard blocks are supplied, then surplus demand will be implemented in the area-inefficient general-purpose logic fabric. To account for these issues, hard blocks on an FPGA are often made more flexible to accomodate differing demands from the circuit. Two common methods to make hard blocks flexible are: 1) make them *fracturable* (divisible into smaller components), or 2) make them reconfigurable into different modes of operation.

Apart from embedding fixed-functionality blocks, another approach to reduce area is to remove some of the flexibility of the programmable logic on an FPGA by forcing constraints on what programmable logic may be put together. For example, a programmable logic block that can implement one large logic function may be fractured to implement two smaller logic functions with some shared inputs instead of two completely independent logic functions. As long as the constraints are satisfied often enough in a user circuit, it can provide significant area savings. There is a current trend for modern commercial FPGAs to have fracturable programmable logic blocks [12] [17] because they use large programmable blocks for delay benefits [1] and it is hard to fully utilize these blocks unless they are fracturable [22].

Various clever architectures [16] [13] have been developed that aim to improve FPGA area by selectively removing programmability. Modern commercial FPGAs, with their fracturable hard blocks and programmable logic blocks, are a few example points in this broad space of complex blocks in FPGAs. Architects require a method to compare the quality of these architectures in order to gain a better understanding of different design decisions.

FPGA architecture research is typically done empirically by mapping a set of benchmark circuit designs into each candidate architecture using a set of Computer-Aided Design (CAD) tools. Then, the results are compared on the basis of quality metrics such as area, speed and power. The CAD tools must be able to understand the architecture being explored and utilize the unique features of each architecture in question in order to fairly compare the different architectures and rigourously navigate this space. To date, a generic and flexible CAD flow that is aware of architectural features such as hard blocks, fracturabilility, and multiple modes of operation does not exist in the public domain. The creation of that capability is the focus of this research, with specific attention paid to the packing stage in the CAD flow.

The packing stage of a typical CAD flow begins at the point where the user circuit is in the form of a technology-mapped netlist, then groups the netlist components into the complex blocks available on the FPGA. In this thesis, we investigate architecture-aware packing by focusing on two problems: the first is a way to describe the complex blocks of an FPGA including fracturability, hierarchy, and multiple modes of operation, the second is a way to pack to those complex blocks.

We envision that the knowledge gained in this research will be a key step towards enabling exploration of new complex block architectures. Furthermore, it will aid in developing a future completely architecture-aware academic CAD flow that may be used to further do architecture exploration and algorithms research.

1.2 Research Goals

The goal of this research is to advance architecture-aware packing by:

1. Creating a new generic language that can describe a large number of different complex blocks for FPGAs.

2. Creating a packing tool and algorithm that can pack to a useful subset of the

language with a focus on area-driven packing.

3. Measure and confirm the quality of the packer for a subset of the language.

To satisfy the first goal, a new language to describe the complex block was developed. This language allows the FPGA architect to specify the routing, modes of operation, and hierarchy of blocks that form the complex blocks of an FPGA in an intuitive fashion. Secondly, a new area-driven packing tool was developed that packs to architectures specified by the new architecture description language and has been tested for fracturable memory, multiplier, and general-purpose logic blocks. Lastly, the quality of results of the packing tool was compared to a commonly-used packer for standard FPGA architectures and against a lower-bound for new architectures.

1.3 Thesis Overview

Chapter 2 provides background information and discusses previous work in modeling FPGA logic blocks and packing to those logic blocks. Chapter 3 introduces the new architecture modeling language for FPGA logic blocks. Chapter 4 describes the packing algorithm used to pack a user circuit into a subset of architectures described in Chapter 3. Chapter 5 describes the experimental methodology used to measure the quality of the packing algorithm and gives experimental results. The final chapter offers a summary of the thesis contributions and suggests future work.

Chapter 2

Background

This chapter covers the background material related to the key contributions in this thesis, which are a new language for describing FPGA complex block architecture, and a tool and algorithm that can *pack* smaller pieces of user logic into larger, widely varying, complex blocks. We begin by describing the relevant basics of FPGA architecture, and the prior work on languages that have been used to describe FPGA architecture. We then give an overview of the complete Computer-Aided Design flow needed to take a user circuit and implement it on an FPGA. As the focus of this work is on the packing step, we describe the related prior work in that field.

2.1 FPGA Architecture

The architecture of an FPGA consists of the set of blocks that perform internal computing, the Input/Output blocks that communicate with the extra-chip world, and the programmable routing structure that connects them. As FPGAs have evolved, they have employed increasingly more rich and complex blocks that consist of a larger number of small components, which we will call *primitives*, grouped together.

One purpose of this grouping is to leverage the locality typically found in circuits. This concept of locality leads naturally to the creation of a hierarchy of clusters. For the

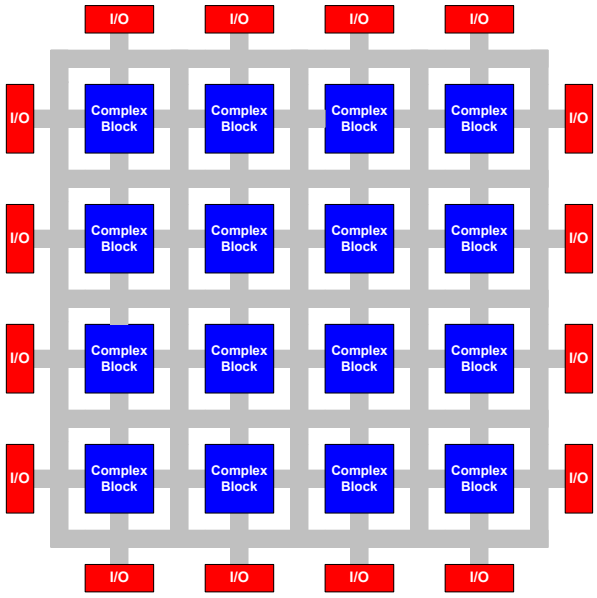

Figure 2.1: Island Style FPGA

purposes of the discussions below we will refer to clusters that only contain primitives as *level 1* clusters. We refer to the the top-level cluster as a *complex block*.

This thesis focuses on a class of FPGA architectures known as island-style FPGAs, which consist of complex blocks surrounded by interconnect with input and output pins at its periphery, as shown in Figure 2.1. There are two major categories of complex blocks for island-based FPGAs: general-purpose and special-purpose complex blocks. General-purpose complex blocks (GPCBs) are flexible enough to be able to implement any logic function. Special-purpose complex blocks (SPCBs) contain fixed-functionality blocks (and are also known as hard blocks) that perform a more limited set of computation, and does those computations more efficiently than the general-purpose blocks. SPCBs are not intended for general-purpose use, though they can be used for general-purposes under certain conditions [46] [20]. In the following sections we describe these each in turn.

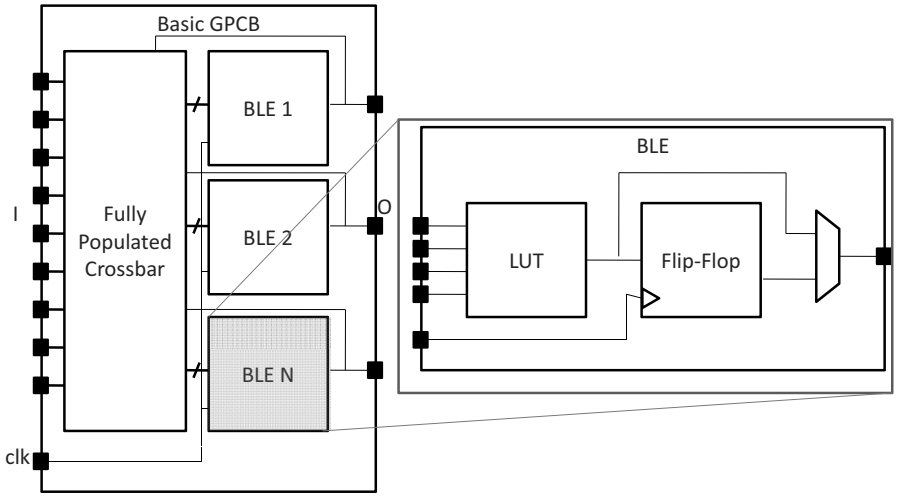

Figure 2.2: Basic general-purpose complex block and the BLE inside of it

2.1.1 General-Purpose Complex Blocks

GPCBs provide the core logic flexibility of an FPGA, and with a sufficient number of them, can implement any logic function. A typical GPCB consists of programmable logic primitives and flip-flops. One of the most common logic primitive is a k-input 1-output look-up table (LUT), which can be programmed to implement any k-input Boolean logic function.

Figure 2.2 illustrates the structure of a basic GPCB. This complex block consists of Basic Logic Elements (BLEs), and a full crossbar for interconnect. The full crossbar connects the inputs of the complex block to any of the inputs of the BLEs inside it, and provides a feedback path from the outputs of the BLEs back to any of the inputs of the BLEs. The outputs of the BLEs connect directly to the complex block outputs.

The expanded view of Figure 2.2 shows the internals of a typical BLE; the multiplexer allows the BLE to implement a simple LUT or a LUT feeding directly into a flip-flop. If the BLE is to implement a flip-flop only, then the mux would be set to have the LUT feed the flip-flop and the LUT is programmed to have the same logic function as a wire.

Modern FPGAs [12] [17] employ several techniques to improve the architecture of this basic GPCB. The first attempts to improve area efficiency by adding functionality to the

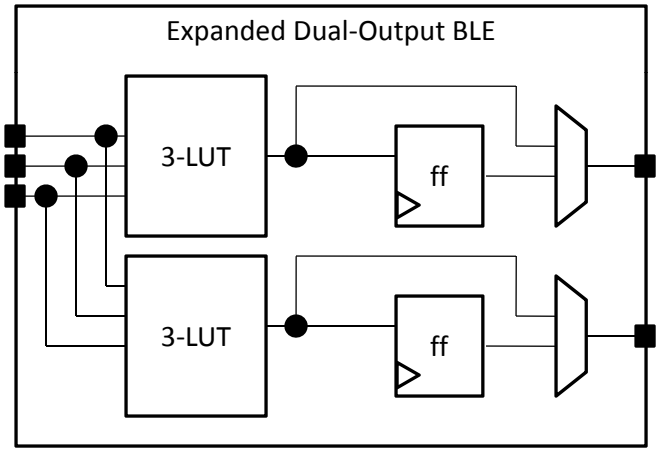

Figure 2.3: An expanded BLE with two LUTs that share common inputs

BLE. The BLE is expanded to have two outputs and contain two LUTs that share some input pins. An example of an expanded BLE with two 3-LUTs that share inputs is shown in Figure 2.4. This example has optional flip-flops for each output of the BLE. Since the two LUTs share input pins, they can reuse the interconnect resources required to bring connectivity to the pins of the BLE. It is thus possible to achieve an area saving as long as both LUTs in the expanded BLE are used often enough. Some FPGA architects take this technique further by allowing those two LUTs to optionally implement a larger LUT via a 2-to-1 mux. This technique relies on Shannon decomposition, which states that a function can be represented by an aggregate of two subfunctions of the original. Given a function that has one too many inputs for one of the two LUTs in the expanded BLE, this technique implements the function as follows: One LUT implements the function assuming the last input of the BLE is logic '0', the other LUT implements the function assuming that the last input is logic '1', and the last input is used to select which subfunction to use with the 2-to-1 mux. Figure 2.4 shows an example of the structure where two 3-LUTs can optionally implement one 4-LUT. A structure that can implement one large LUT or multiple smaller LUTs is called a *fracturable* LUT. Each option (ie. one large LUT or many small LUTs) is a *mode* for that fracturable LUT. Figure 2.5 shows an example of a fracturable LUT with two modes. It can operate as a 6-LUT or it can

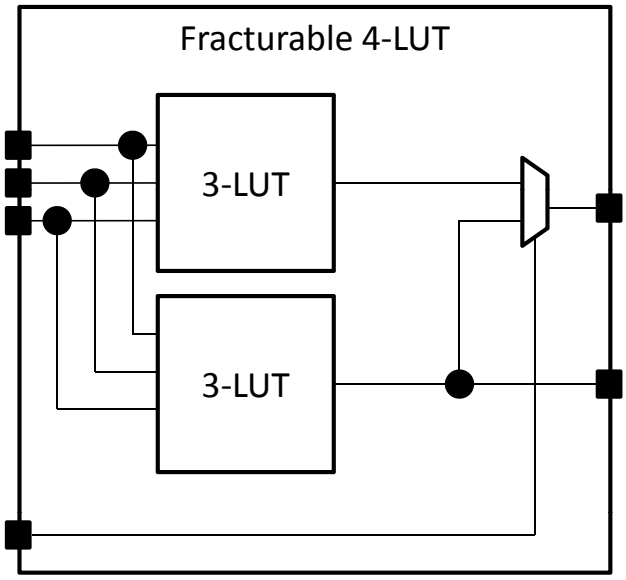

Figure 2.4: Merging two small LUTs to create a larger LUT using a 2-to-1 mux

operate as two 5-LUTs with 3 shared inputs. The number of inputs used in the dual LUT mode is called *FI*. For fracturable 6-LUTs, if FI is 5, then all the BLE input pins to the dual 5-LUTs are shared. If FI is 10, then none of those input pins are shared. In this example, FI is 7.

The two major FPGA vendors today use fracturable BLEs in their GPCBs. Xilinx released its Virtex-6 FPGA that contain fracturable 6-LUTs. One of these fracturable 6-LUTs can implement one 6-LUT or two 5-LUTs that share common inputs (FI = 5) [17]. Altera's Stratix IV FPGA has GPCBs that contain 8-input fracturable BLEs that can implement two 5-LUTs with 2 shared inputs (FI = 8), one 6-LUT, a subset of 7-input logic functions, and some other combinations [12]. Both vendors chose to have optional registers at both outputs of the fracturable LUT. Despite the prevalence of fracturable BLEs in commercial FPGAs, publicly available tools for FPGA architecture exploration cannot yet model them. Furthermore, the vendors have chosen different FI values for their fracturable BLEs and it is unclear which one is "best" because of the lack of exploration tools for fracturable LUTs.

The second technique reduces the area of the basic GPCB by depoulating the full

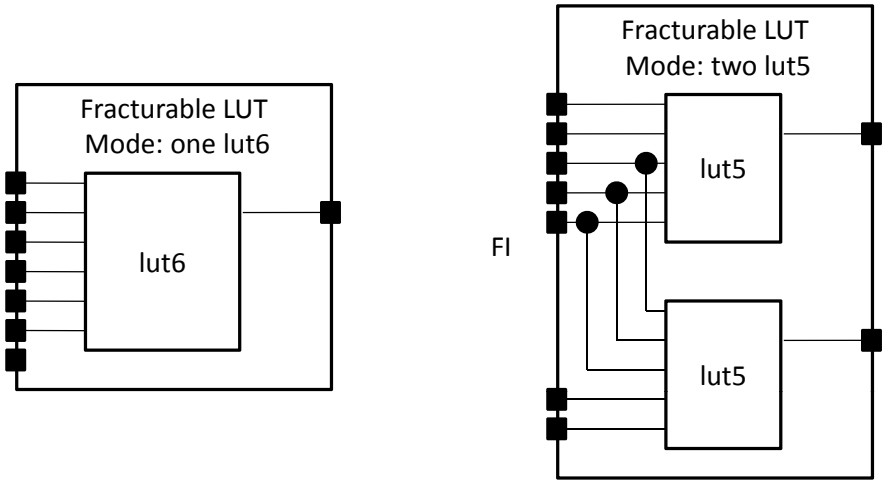

Figure 2.5: A fracturable 6-LUT with 7 inputs.

crossbar inside it. Lemieux [26] discusses different methods of depopulating that crossbar. Lewis [27] also discusses the use of depopulated crossbars in Stratix FPGAs. It is likely that academic research ignores the depopulation of crossbars inside the GPCB because of the lack of public-domain tools to model it.

A third technique attempts to improve the area and delay of GPCBs by adding special hardware and interconnect for arithmetic. A dedicated arithmetic structure is added to the BLEs and fast carry-chains [15] [8] are added to join together those arithmetic components across the BLEs. Without carry-chains, two BLEs are needed to implement a full adder because a full adder has two outputs while a BLE has just one. The carry-chain serves as a second output for a BLE so including arithmetic hardware and carry-chains allow one BLE, with some small area overhead, to implement a full adder instead of two BLEs. If arithemetic operations are common enough in user designs, then this results in an area savings. Furthermore, a carry-chain is a common timing critical path, if left to be implement in simple BLEs, and so these structures improve speed. Both the Virtex-6 and Stratix IV FPGAs contain adders and carry-chains in their GPCBs. Although there exist academic tools [40] that model carry-chains, they often place architectural assumptions in their source code making it difficult to model carry-chains that deviate

from those assumptions.

These advances on the GPCB and the lack of versatile public-domain tools to model them motivates the two key contributions of this thesis. The first is a public-domain language capable of describing these advances and that can be extended to cover future advances. The second is a public-domain tool that can map a user's circuit to the architectures described by the language.

2.1.2 Special-Purpose Complex Blocks

An FPGA architect may choose to embed specialized hard blocks in the FPGA. These hard blocks implement common functions found in digital designs typically using much less area, delay, and power than implementing those same functions using GPCBs [24]. An SPCB is a complex block that contains those hard blocks. Figure 2.6 shows an example of a heterogeneous island-style FPGA with GPCBs and different sized memory and multiplier SPCBs.

Hard blocks provide area savings if used, but waste area if unused, and so it is important to determine the right number and type of them to include in an FPGA. This remains an open problem and one that clearly depends on the applications being targeted. Furthermore, different designs may require different sizes or variations of hard blocks. As a result, an FPGA architect may try to improve the utilization of an SPCB by making it configurable into different, related functions, or fractured into multiple smaller hard blocks with the same function. Examples of hard blocks in the SPCBs of commercial FPGAs that make use of configuration/fracturing are memories and multiplier-oriented blocks. Due to the prevalence of these two hard blocks in modern FPGAs, we describe them in more detail.

Both Altera and Xilinx commercial FPGAs have SPCBs with configurable memories [17] [11]. For example, the Virtex-6 36 Kb dual-port memory SPCB can implement a dual-port memory with many aspect ratios a few of which include 32Kx1, 4Kx9, and

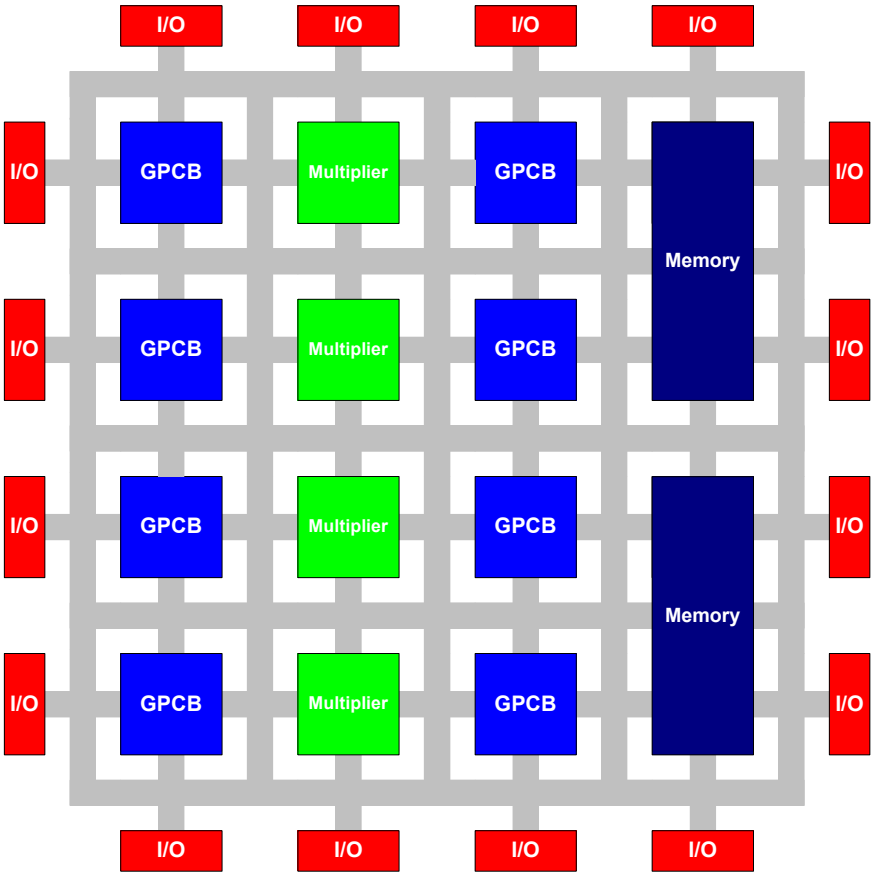

Figure 2.6: Heterogeneous FPGA with GPCB and different SPCBs

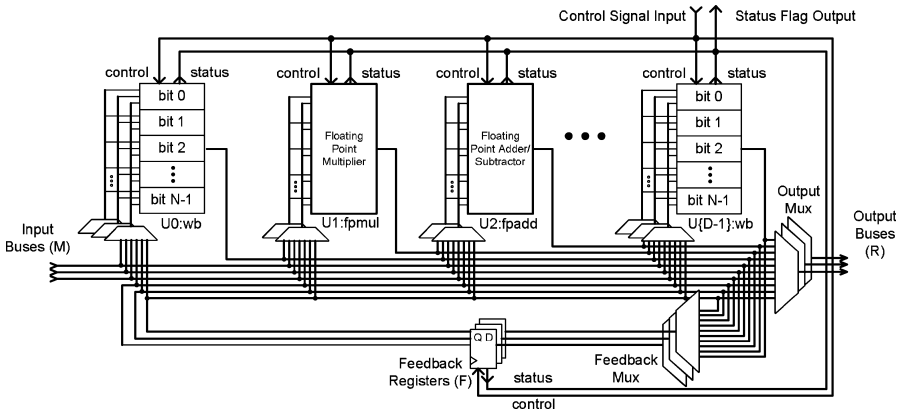

Figure 2.7: Floating-point core on an FPGA proposed by Ho [16]

1Kx36 [17]. Implementing such memories are non-trivial. Ngai describes a memory with configurable aspect ratios in [37].

Similar to memories, many commercial FPGAs contain hard multipliers that are configurable. The Stratix IV hard multiplier can be fractured from one 36x36 multiplier into multiple smaller 18x18, 12x12, or 9x9 multipliers [11]. These SPCBs can also operate in modes that perform different, but related functions such as addition, or multiply-accumulate [17] [11].

Apart from memories and multipliers, there has been prior work describing other types of SPCBs that can be embedded into an FPGA. Ho [16] proposed adding a family of SPCBs for floating-point computation onto an FPGA as shown in Figure 2.7. This family consists of combinations of floating-point multipliers, floating-point adders, *word-based LUTs*, and registers. Word-based LUTs are groups of LUTs that must share the same logic function. A word-based LUT perform the same operation on different bits of a bus of wires. The different primitives of Ho's SPCBs are linearly placed and connected together using bus-based interconnect and feedback paths. Instead of being fracturable, these SPCBs implement different floating-point operations by programming the bus interconnect and word-based LUTs.

Some FPGA architectures are composed entirely of SPCBs. These architectures are known as coarse-grained FPGAs. This is a deep area of study and several authors have

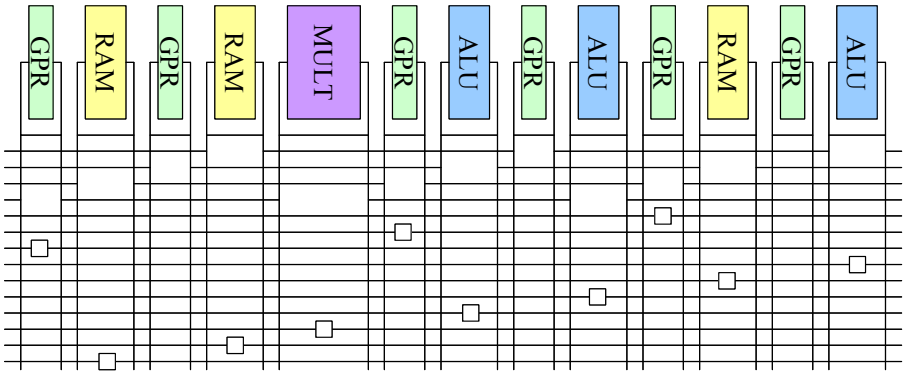

Figure 2.8: RaPiD FPGA architecture datapath SPCB [14]

proposed different SPCBs for coarse-grained FPGAs [36]. We will discuss one such FPGA called RaPiD [13].

RaPiD is an FPGA architecture that consists of one large datapath SPCB and some separate control logic. The RaPiD datapath follows a linear structure as shown in Figure 2.8. The blocks are attached using segmented, bidirectional busses. Although RaPiD is not an island-style FPGA, one can imagine that its datapath can be embedded in an island-style FPGA in a way similar to Ho's work.

The wide variety of SPCB designs motivates the need for a public-domain language that can describe them, as well as a public-domain tool that can map a user's circuit to these blocks.

2.2 Complex Block Architecture Description Languages

The previous section described a few points in the large space of complex blocks that FPGA architects may wish to investigate. In order to explore that space, a language that can precisely specify those complex blocks is needed. This section describes a taxonomy of different languages that were developed to describe complex blocks. Then, we introduce a mark-up language called XML [45] which is used in the new language that we will

present in the next chapter.

There are three main types of complex block description languages: 1) Languages that describe GPCBs only, 2) Languages that describe SPCBs only, 3) Languages that describe both GPCBs and SPCBs.

2.2.1 GPCB Description Languages

GPCB description languages focus on describing the basic GPCB shown in Figure 2.2 and some variants of that GPCB.

VPR 4.30 is an FPGA architecture exploration tool released in 1999 [5]. It is a widely cited tool used to explore FPGAs composed of the basic GPCB as well as a wide variety of FPGA routing architectures. It is also the ancestor of several different description languages that focus on the basic GPCBs.

In VPR 4.30, the architect describes the basic GPCB that he/she wants to explore using data-value pairs. Comments are preceded by the character '#' For example:

```
subblocks_per_clb 4          # This means 4 BLEs in a cluster
```

The FPGAs that VPR 4.30 could explore were very broad and extensive for its time. It was capable of varying the number of BLEs (N), number of input pins to the GPCB (I), and size of the BLE (K) for basic GPCBs. It also provided some support for alternative types of complex blocks through *logical equivalence*. Pins that are logically equivalent can have their locations swapped with each other without changing logic functionality.

The four drawbacks of this description language are:

1. There is only one kind of intra-block interconnect - a full crossbar,
2. BLEs can only have one output,
3. There can be only one type of BLE within a complex block, and
4. There can only be one type of complex block.

In our own prior work [30], we attempted to resolve some of the drawbacks of VPR 4.30 in the upgraded VPR 5.0. We modified the description language to make use of XML [45]. This stylistic change made it easier for users to make modifications to the language because XML parsers are readily available compared to the VPR 4.30 custom format. The two primary language contributions of the language developed in the VPR 5.0 release are that it enabled architects to specify different types of complex blocks, and that it supported multi-output level 1 clusters (recall that level 1 clusters are clusters that only contain primitives). That language assumed that the columns of an FPGA contain complex blocks of the same type so the ratio of different types of complex blocks, known as the supply ratio, is specified by the number of columns that a type of complex block occupies. However, this language also assumed that level 1 clusters were uniform, and did not let the user specify the arbitrary hierarchy and interconnect between and among those hierarchies.

Pervez [41] proposed two new additions to the VPR 5.0 language: the first is to individually specify the timing of each interconnect edge in a level 1 cluster. The second is to specify each potential programmable connection individually. This results in a high degree of flexibility but also a significant amount of verbosity in the architecture description language. In our work in Chapter 3, we describe a new language that attempts to strike a good balance between verbosity and flexibility.

These restrictions on GPCB description languages motivated other researchers to create languages to describe SPCBs.

2.2.2 Special Purpose Complex Block Description Languages

A SPCB description language describes complex blocks containing hard blocks. These languages range from simple parameter descriptions to a netlist representation.

Ho used parameters to describe different SPCBs that performed floating-point operations on an FPGA [16]. The tool assumed a particular pipelined architecture. The

user would describe the SPCB architecture by setting various parameters that specify the number of different types of blocks and busses available. The tool would then automatically generate the SPCB. This language is concise but is highly restrictive on the types of SPCBs that can be explored.

There are several architecture description languages for SPCBs that employ a netlist style representation. In RaPiD [13], users can specify the number of different blocks, their placements, and their routing. It assumes a linear architecture in which the coarse blocks are simply placed in a single line, with interconnect busses running alongside.

Filho proposed an architecture description language called CGADL that is designed for coarse-grained FPGAs [39]. Filho's language assumes that the SPCB is a level 1 cluster. The SPCB can contain five types of parameterizable primitives: memories, registers, calculation units, bus-based multiplexers, and finite state machines. The interconnect within a SPCB is specified as port-to-port connections where one output port can connect to multiple input ports. However, CGADL can be very verbose for some architectural constructs that are common on FPGAs. For example, the full crossbar connection found in the basic GPCB requires that the architect individually describe each mux of the crossbar. There are several other architecture description languages for FPGAs with SPCBs only and they are classified in [36]. These langauges tend to take the form of a netlist specification of primitives and clusters connected within the block. They are highly expressive and flexible but, similar to CGADL, that same expressiveness can make it cumbersome to express GPCBs.

The expressiveness of the netlist-style description found in several of these SPCB languages motivated the use of a netlist-style description for the new language that we propose in the next chapter. Also, the problem with verbosity encountered with a netlist-style language motivated us to reduce verbosity in our new language.

2.2.3 Combined GPCB and SPCB Description Languages

Some architecture description languages aim to accomodate both GPCBs and SPCBs. These include the recently proposed CARCH language [40] and standard HDLs.

In [40], Paladino proposed a new hierarchical language called CARCH with the intention of using this langauge to describe commercial FPGAs from Altera. This language is the most relevant to the goals of the present research, as its intent is to describe both GPCBs and SPCBs in FPGAs so it will be covered in more depth than the other languages presented thus far.

Similar to the VPR 5.0 language, CARCH describes complex blocks assuming a two-level hierarchy. Unlike the VPR 5.0 language, the architect can specify the primitives that comprise the level 1 cluster. Level 1 clusters contain a fixed number of primitives and these primitives can have different modes of operation. Similarly, complex blocks contain a fixed number of level 1 clusters and these clusters can have different modes of operation. Both complex blocks and level 1 clusters can be heterogeneous.

Another key concept in the language is the ability to have blocks that operate in different *modes*. A mode is a configuration for a block that is mutually exclusive with other configurations.

The overall structure that the CARCH language uses to describe complex blocks is follows:

```
define all types of primitives and their modes
define all types of level 1 clusters and their modes
define all types of complex blocks and their modes
```

The complex block, level 1 clusters, and primitives generally follows the definition pattern:

```
name
port definitions
modes of operation
rules on packing subblocks of cluster if applicable
```

CARCH has a few special case, very specific, constructs for blocks in each level of hierarchy. For example, it has a unique carry-chain construct for level 1 clusters.

CARCH is much more flexible than the GPCB-only language family and can be more concise than the SPCB languages. CARCH introduces ports for clusters and primitives, complex matching rules, modes of operation, and heterogeneity within a cluster. These are significant advancements in the description of GPCBs and a large step towards generically describing SPCBs.

There are some weaknesses with this language. It can be quite difficult to determine the design rules of the CARCH language for a new complex block architecture. For example, if a GPCB uses a depopulated crossbar, as found in the devices that it was tested on, the architect must use a rule to restrict the number of input pins used by the GPCB, and then empirically tune that number for the CARCH language to closely approximate the depopulated crossbar. Ideally, an architect should directly describe architectural structures, such as a depopulated crossbar, of a complex block instead of manually inferring rules from those architectural structures.

The CARCH language was built with the goal of representing the GPCBs of commercial FPGA architectures instead of architecture exploration and this bias makes the tool more difficult to use for architecture exploration. For example, the CARCH language has a large number of different parameters, conventions, and keywords, making the language difficult to learn. Also, additional features require a modification to the CARCH language. For example, it does not have an option to register input ports (a common trait in FPGA memories) because this trait did not appear in the GPCBs of the commercial devices. This feature can easily be added to the language as another property but such a system can quickly lead to a bloated language with many special case constructs.

Another method to specify GPCBs and SPCBs is to use an existing Hardware Description Language (HDL), such as Verilog. Such a specification gives very high architecture flexibility and precision as the user can describe any digital hardware device. The diffi-

culty with such an approach is that it becomes hard to parameterize and explore different complex blocks quickly because an HDL is too detailed for such purposes.

An ideal complex block architecture description language should be able to clearly and directly describe a wide variety of complex block architectures yet also be able to maintain conciseness through the use of abstraction.

2.2.4 XML

This section provides a brief introduction to XML [45] because XML is used by the new language described in this thesis. XML is a language that is widely used to describe content that contains intrinsic hierarchy, such as documents and software configurations. XML can also conveniently describe an FPGA architecture due to the intrinsic hierarchy found in FPGAs.

XML describes content using a hierarchy of elements. Elements are described using tags.

There are three types of tags. Start-tags mark the beginning of an element. They start with a <, followed by the name of the tag, then some optional attributes (that will be described later), and end with a >. An end-tag marks the end of an element and corresponds with a start-tag. It begins with a </, followed by the name of the matching start-tag, and ends with a >. For example: <*foo*> is a start-tag that marks the beginning of an element called *foo* and </*foo*> is an end-tag that marks the end of that element. All content between an element's start tag and corresponding end tag is contained inside that element. The third type of tag is called an empty-element tag. Empty-element tags describe elements that do not contain other elements or text. They follow the same structure as start-tags but end with a /> instead of a >. This type of tag does not have a corresponding end-tag.

Attributes are name-value pairs that optionally appear inside start-tags and empty-element tags. The name is separated from the value by a = and the value is enclosed

in quotes. For example <*foo bar="purple"*/> describes an empty-element tag called *foo* that has an attribute *bar* set to the value *purple*.

An short example of a hierarchy of XML elements is as follows:

```
<chapter>
  <foo bar="purple"/>
  <text>Hello World!</text>
</chapter>
```

Describes an element called *chapter* that contains two elements. The first element is an empty-element tag *foo* with an attribute *bar* set to *purple*. The second element is an element *text* that encloses the content *Hello World!* with its start and end tags.

Comments in XML start with a <*!–* and end with a *–*>. For example:

```
<!--  This is a comment
      that spans multiple lines
-->
```

These are the basic rules of the XML language that will be used in the next chapter.

2.3 Computer-Aided Design Tools for FPGAs

2.3.1 Overview

The design of modern FPGAs must almost always use Computer-Aided Design (CAD) tools to map a circuit to the FPGA, because of the size and complexity of the devices. The inputs to the CAD flow are a user circuit specified in a Hardware Description Language (HDL), and some method of describing the devices being targeted. Commercial FPGA CAD software contains databases that describe the specific features of the architecture of the devices being targetted, as well as software code written that is specific to the devices. Academic CAD flows, which are often aimed at architecture exploration, will more typically have a separate input architecture description language, such as those described in section 2.2 earlier, to specify the device being targeted. The output of the

CAD flow are the configurations that implements the user circuit on the specified FPGA architecture.

A typical academic FPGA CAD flow is shown in Figure 2.9 (a). The first stage of this flow is called front-end synthesis. Front-end synthesis first elaborates the HDL of the user circuit, it then performs logic optimizations on the circuit, and finally, it does technology-mapping which maps the logic of the circuit into the types of primitives found in the FPGA. The output of front-end synthesis is a technology-mapped netlist. The netlist then passes through the back-end flow. The first stage of the back-end flow is called packing and this stage maps the technology-mapped blocks of the netlist into the complex blocks available on the FPGA. Then the placement stage determines the location of those complex blocks and the final routing stage resolves the connections between all blocks. The end result is the configurations necessary to realize the user's circuit on the FPGA.

A large body of work has been published on different stages of this CAD flow. Despite this work, a complete flow to explore the more advanced complex blocks described earlier is lacking. For example, when Ho explored the architecture of floating-point cores on FPGAs, he did it using manual technology-mapping and packing[16].

The lack of a complete CAD flow for more advanced complex blocks is partly caused by architectural inflexibility in the front-end synthesis and packing stages.

Publicly available tools that implement these stages often hard-code architectural assumptions thus limiting the architectures that can be explored. For example, information such as detailed-routing inside complex blocks is often ignored in public-domain packers. Similarly, fracturable multipliers and configurable memories are often also ignored in front-end synthesis. If detailed architectural information was used in all stages of the CAD flow, as shown in Figure 2.9 (b), then the different parts of the flow could make more intelligent algorithmic choices to map a circuit into a given architecture. All tools need to be architecture-aware. There is currently new and on-going work to en-

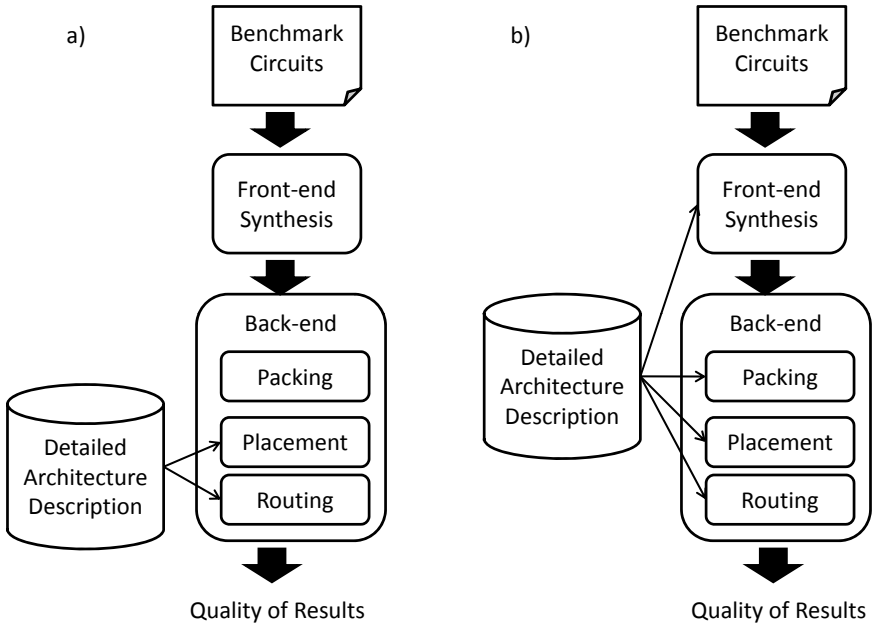

Figure 2.9: FPGA CAD Flow a) Typical CAD flow b) Fully architecture-aware CAD flow

able architecture-awareness in the up-stream tools and this thesis is a part of that effort. In the following paragraphs, we will describe this related work, as some of the results presented in this thesis rely on it.

ODIN II [21] is a front-end synthesis tool that performs elaboration and some small amount of synthesis on a Verilog netlist; it is a new, clean slate implementation of the first version of ODIN [19]. It incorporates architecture-awareness into the front-end flow. ODIN can infer multipliers, and provide black-box module support, and support for *splitting* memories and multipliers. Splitting is the process of dividing a large function to multiple smaller ones. Splitting is necessary when a user has a logical function that is too big to fit into the physical hard blocks on the FPGA. The function gets split into multiple smaller ones so that it can fit into the hard blocks of the FPGA. ODIN II uses the same detailed architecture description of the FPGA as the back-end so that it can split mutlipliers and memories based on the sizes available in the architecture. Publicly available tools such as ABC [44] can be then used to optimize and technology map the netlist circuit generated by ODIN II. The result is a user netlist that is ready for the

back-end flow.

There has also been work that attempted to expand the architectures supported in the back-end flow. If the packing stage can pack the technology-mapped netlist into the SPCBs and GPCBs that VPR 5.0 [30] supports (described earlier), then the recently released VPR 5.0 can be used to place and route that circuit. This is a significant step forward from the older, and more widely-used, VPR 4.30 tool [5] that can only model FPGAs with one type of basic GPCB.

The key missing link in enabling a complete architecture-aware CAD flow is in the packing stage. There is a large body of work in the field of packing for FPGAs but very little work in a generic packing algorithm that can understand the detailed architecture of an FPGA complex block and pack to it. A generic packer that can leverage that information will help enable a complete CAD flow to explore new FPGA architectures.

2.3.2 Packing Algorithms

Overview

The packing problem takes as input a technology-mapped netlist and maps it into available complex blocks on the FPGA. A technology-mapped netlist is composed of primitives (such as LUTs, flip-flops or multipliers) connected by nets.

For the output of the packer to be correct (legal), the primitives that are packed inside a complex block must be connectable (routable) as dictated by the netlist and the block's internal routing architecture. The packing problem is defined as finding a legal packing that minimizes a cost function, where the cost function is usually correlated with area, delay, and/or power consumption of the resulting operating circuit.

There is large body of prior work on the packing problem for FPGAs. Most of it focuses on optimizing packing for the basic GPCB described in Figure 2.2, which will be covered first. Subsequently, we will describe packing algorithms for specific types of

SPCBs and more complex GPCBs.

Basic GPCB Packing

Basic GPCB packers deal with the fairly homogeous problem of packing LUTs and flip-flops into GPCBs (as illustrated in Figure 2.2). A LUT and a flip-flop are grouped into one BLE. BLEs are in turn grouped to form the complex block. Basic GPCB packers assume that the technology-mapped user netlist is already packed into BLEs in a simple stage called pre-packing. The input is a netlist of pre-packed BLEs and the parameters that describe the GPCB:

1. Number of BLEs in a GPCB, N
2. Number of routed inputs to the GPCB, I
3. Number of inputs on each LUT in the GPCB, K

The output is a netlist of GPCBs that implement the user circuit. Most of these packers follow a greedy heuristic that packs one BLE at a time; these packers use an *affinity* function to choose the next block to pack into the GPCB. The higher the affinity a block has to a GPCB, the more "desirable" it is to pack that block into the GPCB.

T-VPack [33] is a timing-driven packer that packs a basic GPCB one BLE at a time until there is no space left, or until there are no suitable BLEs. It then 'opens' a new GPCB and repeats the process. There are cases where a GPCB cannot be filled because there aren't enough input pins available to pack more BLEs into it. T-VPack attempts to resolve this problem with a hill-climbing heuristic: it will try to pack more BLEs into the complex block in an attempt to reduce the number of input pins and acheive a legal packing. Figure 2.10 shows an example of a case where hill climbing is needed to pack more BLEs into a GPCB. The figure assumes a basic GPCB with N = 3, I = 4, and K = 4. BLE 1 is packed into the complex block first and this uses up all the input pins. Packing BLE 2 into the cluster results in an illegal packing because there

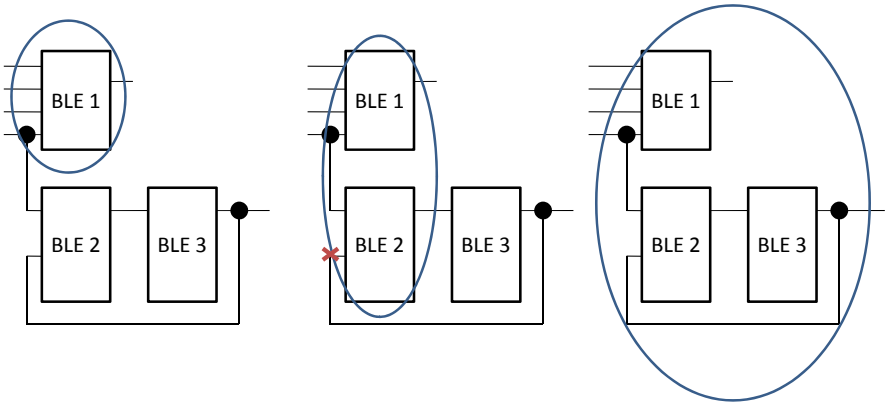

Figure 2.10: T-VPack hill-climbing example for a N = 3, I = 4, K = 4 basic GPCB

are not enough input pins. Packing BLE 3 results in a legal packing because its inputs consume some of the inputs that were previously required to come from the external routing, making the packing legal again. T-VPack uses an affinity function to determine which pre-packed BLEs to pack into a GPCB. The affinity function is composed of two parts, an area-driven shared net count and a timing-driven critical path gain. T-VPack is widely-used in conjunction with VPR and its area and delay results are both less than its predecessor, the area-driven VPack. The version of T-VPack that comes with VPR 5.0 does not pack other types of complex blocks and passes those structures through as monolithic black-boxes.

Subsequent work on basic GPCB packers use an empirical method to compare their particular algorithm with the T-VPack algorithm. This empirical method consists of running a set of benchmark circuits through a CAD flow and then measuring some quality metrics. To isolate the effects of the packing algorithm in their experiments, these experiments typically keep all parts of the CAD flow constant except for the packing algorithm. The benchmark circuits used in these experiments are often from the MCNC [47] benchmarks. These circuits contain only logic and flip-flops (i.e. no hard blocks structures).

Bozorgzadeh [6] explored adding routability heuristics to T-VPack and developed the T-RPack algorithm. T-RPack uses the T-VPack algorithm with an altered area-driven affinity function. Instead of just counting the number of nets that a candidate pre-packed BLE has in common with nets in the current GPCB, the algorithm weights each pre-packed BLE pin based on whether it is an input or output pin, how many pins it is shares with the current GPCB, and whether the GPCB contains a driver for that pin. As well, T-RPack penalizes candidate pre-packed BLEs for each addition GPCB input pin that the candidate pre-packed BLE requires. On the 20 largest MCNC benchmarks, the T-RPack algorithm can reduce the average minimum channel width by 3% and reduce the average critical path delay by 5% compared to the T-VPack algorithm.

Singh [43] proposed a packer, IRAC, that is similar to T-VPack but with an additional routability heuristic to rank the candidate pre-packed BLEs. Singh's packer packs less tightly than T-VPack because it tries to create a uniform Rent's exponent across all the clusters. A key characteristic of IRAC is that it priortizes the BLEs before packing based on the number of incident nets and the total number of pins on incident nets. This allows the algorithm to more intelligently select the seed pre-packed BLE and candidate pre-packed BLEs such that more nets are absorbed into the GPCB, thus reducing minimum channel width. On the 20 largest MCNC benchmarks, the average area-delay product of circuits packed use this algorithm is 25% less than the ones packed using the T-VPack algorithm.

There has also been prior work on combined technology-mapping and packing. Cong [28] investigated simultaneous technology-mapping and packing, while Ling [29] investigated the application of techniques commonly found in technology-mapping to the packing problem (ie. finding a logic depth optimal packing solution using a dynamic programming algorithm). These methods tend to produce better delay, worse area, and similar area-delay results compared to the T-VPack algorithm.

Chen [9] proposed a timing-driven, greedy, packing algorithm called HDPack. This algorithm first employs an extremely fast, min-cut placement using hMetis [23] to place the pre-packed BLEs on a grid. It uses the weighted sum of these grid distances together with the number of shared nets and timing criticality as the affinity function. The key difference between the shared net count in HDPack and T-VPack is that the affinity of a net is scaled by $1/(num\_connections - 1)$ instead of just 1, thus weighting nets with less connections more heavily than nets with many connections. HDPack then employs a two-stage packing process: in the first stage, it packs BLEs with high affinity together into clusters with the aim of absorbing nets. In the second stage, it uses the small clusters as seeds for GPCBs. It then packs the GPCB by adding one BLE at a time until the cluster is full or until no more BLEs will fit. Over the 20 largest MCNC 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. The HDPack and T-VPack algorithms have approximately the same runtime. This packer is the current state-of-the-art in basic GPCB packing.

With the exception of [28], basic GPCB packers are very fast and most MCNC circuits pack within several seconds. This is in sharp contrast to the much longer placement and routing runtimes in the CAD flow.

2.3.3 SPCB and Extended GPCB Packing

An FPGA may contain complex blocks that differ from the basic GPCB and there has been prior work that focus on packing for these architectures. These packers either target GPCBs that add additional features to the basic GPCB or they target specific SPCBs.

The basic GPCB packers assumed that BLEs were packed together in a pre-packing stage before starting the packing algorithm for the complex block. More complex GPCBs may have level 1 clusters (recall that level 1 clusters are clusters that only contain primitives) that are different from the basic BLE. Ni proposed an algorithm that pre-packs

level 1 clusters only [38]. The inputs to the algorithm are the user netlist and a list of all level 1 cluster configurations. A level 1 cluster configuration is defined as a group of primitives with fixed routing. For example, a BLE has three configurations, a LUT, a flip-flop, and a LUT followed by a flip-flop. The output of the algorithm is a pre-packed netlist for the next stage in packing. Ni's algorithm first selects a level 1 cluster configuration from a set of all possible configurations then tries to match the netlist to that configuraton. If it fails, then algorithm selects another configuration and repeats the process. There is no cost function, as the purpose of this algorithm is to find legal matches only and assumes that the next packing stage will properly group those level 1 clusters into a legal packing.

Paladino proposed a design rule check (DRC) based packer called DC that can pack to the GPCBs of two different Altera FPGA families [40]. DC is based on T-VPack but with many additional design rules, a special packing case for carry-chains, and commercial timing models. It models a subset of the CARCH language discussed earlier. DC is timing-driven and models carry-chains, different modes of operation, and ports. It assumes a two-level hierarchy. It assumes that a pre-packer has already grouped level 1 clusters together. If there are multiple types of complex blocks on the FPGA, then DC selects the type to pack based on priorities specified by the architect. It interfaces into a commercial FPGA CAD tool called Quartus II [42] from Altera which performs the other parts of the FPGA CAD flow.

DC implements its design rule checks during the candidate block selection phase in the T-VPack algorithm. It supports all of the packing rules defined in the CARCH language but does not support heterogeniety of complex blocks. DC supports carry-chains using the following algorithm: when a carry-chain head is packed into a complex block, DC will *force pack* the rest of the carry-chain in sequential order into the cluster. If the cluster fills, it starts a new cluster and continues packing the carry-chain.

Paladino speculates that three levels of hierarchy are needed to pack to architectures

with fracturable logic elements (recall these were discussed in subsection 2.1.1) and he also speculates that adding two additional rules to the CARCH language will be sufficient for his tool to pack memories. DC is the only academic tool capable of packing to commercial FPGA architectures. Its major drawbacks are that it assumes homogeneous complex blocks in the FPGA and has limitations in its modeling language. These restrictions prevent the tool from exploring circuits which contain non-trivial SPCBs.

There has also been work to perform packing, or parts of it, during the later stages of the CAD flow. Lemieux proposed using a basic GPCB packer to pack to GPCBs with depopulated crossbars. This is done by leaving extra input pins unused on GPCBs with depopulated crossbars and then use the post-placement router to legalize routing both inside and outside the GPCBs [26]. Xilinx suggests doing packing during placement and using the router to resolve the internal interconnect inside a GPCB [2].

There has also been work on packing for specific types of SPCBs. Xilinx investigated packing their DSP blocks for common operations in such a way that the packed blocks can be aligned during placement and routed with high regularity thus improving delay [2].

The RaPiD portion of the Totem project [14] packs the datapath of a partitioned application into one SPCB. It must determine the specific primitives to use, as well as their placements and routing. Unlike typical FPGA packing, the optimization goals are very different. In Totem, the SPCB interfaces to other large systems, such as a processor, so the packer does not consider interconnect outside of the SPCB and instead focuses on optimizing the internals of the SPCB. In FPGA packing, several complex blocks must interface together so packing algorithms are concerned with the impact on external routing, such as minimum channel width.

This discussion of architecture-specific packing shows that a generic, architecture-aware packer, should have certain properties. It should be able to select and balance the number and types of different complex blocks, clusters, and primitives in the FPGA. It

should automatically infer design rules based on the architecture of the complex block. It should also be capable of performing placement and routing of clusters and primitives within a complex block. Such a packer would greatly expand the ability of FPGA architects to do architecture exploration. A packer that has some of these properties is describe in Chapter 4.

Chapter 3

Complex Block Architecture Description Language

3.1 Introduction

In this chaper, we introduce one of the primary contributions of this thesis – a modeling language for describing FPGA complex block architectures. We first describe the high-level goals that guided the language design. We then introduce the language itself using examples. In particular, we begin by showing how trivial complex blocks can be modeled using the language, and gradually illustrate more advanced features of the language using more complex examples, eventually showing how complex blocks (both GPCBs and SPCBs) with rich connectivity can be modeled. We refer to the new language as the *University of Toronto FPGA Architecture Language* (UTFAL) [1].

The features incorporated into any language should be guided by how the language is intended to be used. In this case, the modeling language will be used by an FPGA architect to describe the FPGA complex block that he/she wishes to investigate. A high-

[1] UTFAL is an FPGA architecture description language, so it must also specify architectural constructs outside of a complex block (such as the interconnect that joins complex blocks together). This is done using the same language as that used in VPR 5.0 [30] and is defined in [31].

level of abstraction is therefore desirable, permitting a broad range of architectures to be explored without bogging down the architect in unnecessary detail. In addition to being used by a human FPGA architect, the language must also be read and used by the FPGA CAD software tools that automatically map a circuit into the target architecture under investigation. The language must therefore provide enough detail to enable such an automatic mapping. With this context in mind, we identified three top-level objectives for the language design:

- Expressiveness: The language should be capable of describing a wide range of complex blocks.
- Simplicity: The language constructs should match closely with an FPGA architect's existing knowledge and intuition.
- Conciseness: The language should permit complex blocks to be described as concisely as possible.

To meet these goals, UTFAL incorporates constructs that directly correspond to the hardware structures that most commonly occur in FPGA complex blocks – constructs that we believe also align closely with an FPGA architect's intuition. In essence, the language provides a toolbox of easy-to-understand constructs that an FPGA architect can use to build descriptions of complex blocks, such as those appearing in today's state-of-the-art commercial FPGAs.

3.2 UTFAL Specification

We now describe how one uses UTFAL to model a complex block. The language constructs in UTFAL use XML, so readers unfamiliar with XML should review subsection 2.2.4.

At a high-level, UTFAL contains two categories of construct: 1) physical blocks, and 2) interconnect. Physical block constructs in UTFAL are used to represent the core

logic, computational, and memory elements within the FPGA; for example, LUTs, flip-flops and memories. Interconnect constructs represent connectivitiy within and between physical blocks, including wiring, programmable switches, and multiplexers. We begin by describing the physical block construct in UTFAL.

3.2.1 Physical Blocks

FPGAs comprise blocks of various types that are repeated multiple times throughout the array, for example, LUT-based complex blocks, memories and multipliers. All of the blocks of a given type have the same design (i.e. same logic and layout). For example, the classical FPGA architecture shown in Figure 2.1 is composed of a tiled array of LUT-based complex blocks. Each LUT-based complex block contains sub-blocks that, in turn, are instances of another type of physical block: a BLE (ie. a LUT/flip-flop pair). UTFAL contains a construct called *physical block type* to represent a type of physical block within the FPGA.

One specifies a physical block type in UTFAL using the XML element *pb_type*. The *pb_type* element has a name attribute to identify it. We begin with a simple example. Consider the empty complex block shown in Figure 3.1 a). The following code snippet shows how the block is modeled in UTFAL:

```
<pb_type name="exampleCB">
</pb_type>
```

Any given complex block will typically contain multiple internal physical blocks. These physical blocks may in turn contain other internal physical blocks and so forth. In fact, the hierarchy formed by physical blocks can be arbitrary and one can think of there being a parent/child relationship between a block and its internal blocks. UTFAL can model an arbitrary hierarchy of physical blocks with a corresponding hierarchy of *pb_type* elements. Often, there is more than a single instance of one type of child block inside a parent block. One uses the *num_pb* attribute to specify the number of instances of a

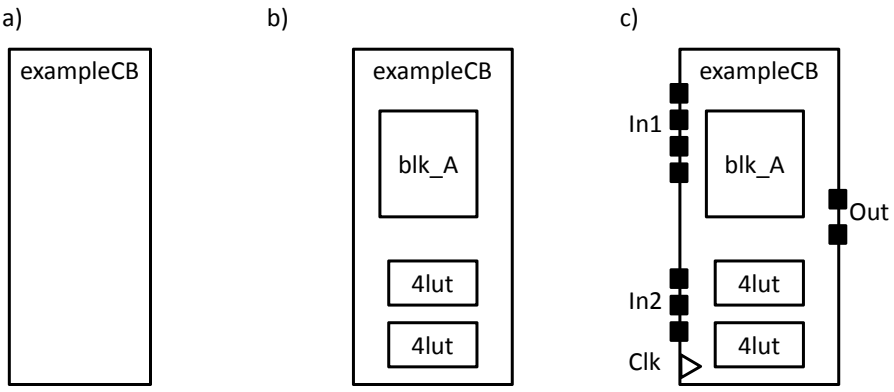

Figure 3.1: Example step-by-step process of describing a physical block

child physical block that are contained in its parent physical block. Figure 3.1 b) extends the example in Figure 3.1 a) to illustrate how block hierarchy is represented in UTFAL. The example adds three child physical blocks to the parent block. The parent block now contains one instance of a block whose type is *blk_A* and it contains two instances of a type of block called *4lut*. The UTFAL specification of the example complex block in Figure 3.1 b) is as follows:

```
<pb_type name="exampleCB">
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2">
  </pb_type>
</pb_type>
```

In this example, *num_pb* is an XML tag that represents the number of instances of a type of block, which is 1 and 2 for *blk_A* and *4lut*, respectively.

Physical blocks must communicate with one another, and also with the external world. A physical block will have a combination of input, output, and clock ports, each comprising one or more pins. In UTFAL, one describes input, output, and clock ports using XML tags *input*, *output*, and *clock*, respectively. Each tag is declared as a child element of the *pb_type* on which the ports reside. A port tag must be given an identifier with the *name* attribute. And, the number of pins in a port is specified with the *num_pins* attribute. The *exampleCB* block shown in Figure 3.1c) adds four ports to the complex

block of part b), and its UTFAL specification is given below. Notice that the In1 port has four pins; the In2 port has three pins; the Out port has 2 pins; and, the Clk port has a single pin.

```
<pb_type name="exampleCB">
  <input name="In1" num_pins="4"/>
  <input name="In2" num_pins="3"/>
  <output name="Out" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2">
  </pb_type>
</pb_type>
```

3.2.2 Modeling Primitives

Recall that *primitives* are physical blocks at the bottom level of hierarchy – they do not contain other physical blocks. A primitive is used to implement a block in the technology-mapped user netlist. UTFAL has an attribute, *blif_model* that must be included in the primitive *pb_type* element. The blif_model attribute specifies the type of netlist block that the primitive implements. UTFAL needs to identify components found in a technology-mapped netlist and so we chose BLIF format[2] [3] as the netlist format but this can easily be modified to support other formats. The value of the *blif_model* attribute for a primitive pb_type is a string the should exactly match the string in BLIF used for the netlist block that can reside in the primitive.

UTFAL incorporates special handling for three of the most common types of primitives found in FPGAs: flip-flops, LUTs, and memory. The rationale for this is that such primitives require special handling by the FPGA CAD tools. UTFAL has a *class* attribute that is used to identify these common primitives. The class attribute is only used for these three common primitive classes; it should be left unspecified for other

[2]BLIF (Berkeley Logic Interchange Format) is a popular public-domain netlist format used for representing digital circuits.

types of physical block. A more detailed description of the three special primitive classes is explained below in subsection 3.2.2. Revisiting the example in Figure 3.1 b), we make the *4lut* a LUT primitive type by adding the *blif_model* and *class* attributes as follows:

```
<pb_type name="exampleCB">
  <input name="In1" num_pins="4"/>
  <input name="In2" num_pins="3"/>
  <output name="Out" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2" blif_model=".names" class="lut">
  </pb_type>
</pb_type>
```

In BLIF [3], LUTs belong to the type *.names* hence the *blif_model* attribute is assigned *.names* for the *4lut* primitive type[3].

In addition to the using *class* attribute, the ports on such primitives must be declared with a special attribute called *port_class*. The port_class attribute is needed to differentiate between the pins on the primitive. For example, the port_class attribute allows an architect to specify which pins on a memory are address pins versus data pins. It turns out that such differentiation is needed by the FPGA CAD tools to map circuits into the architecture. We now delve into the port_class requirements for the common primitives and explain their rationale.

1. *lut*: The LUT primitive has two port classes: one for its inputs and one for its output. The input port class is called *lut_in*; the output port class is called *lut_out*. Requiring that the architect specify such port classes for a LUT permits the FPGA CAD tools to differentiate between the LUT's inputs and outputs. With such knowledge, the CAD tools can perform certain checks and optimizations that they could not otherwise. For example, the tools can execute a design-rule-check to ensure that each LUT primitive has a single output pin. On the input side, the

[3]In BLIF, LUT instances begin with *.names* followed by the names of the signals attached to the LUT inputs and output.

tools can take advantage of input pin *swapability*: signals on LUT inputs can be permuted and the LUT's truth table re-programmed accordingly. Note that more complex LUTs, such as the fracturable LUTs described in subsection 2.1.1, are described as clusters; the basic LUTs within the more complex LUT can then be described using this LUT primitive.

2. *flipflop*: A flip-flop has three port classes: *D*, *Q*, and *clock* for the input, output, and clock ports, respectively. These port classes must have exactly one pin each. The library can be extended to support more ports for flip-flops (such as asynchronous clear).

3. *memory*: Single-port memories have three input port classes: *address*, *data_in*, and *write_en* and one output port class: *data_out*. These port classes represent the address bus, input data bus, write enable, and output data bus, respectively. Dual-port memories have six input port classes: *address1*, *data_in1*, *write_en1*, *address2*, *data_in2*, and *write_en2* and two output port classes: *data_out1* and *data_out2*. Both single and dual-port memories have one optional clock port class: clock (for synchronous memories). The library can be extended to support more ports for memories.

The following example describes a single-port memory primitive type to illustrate the usage of the *class* and *port_class* attributes:

```
<pb_type name="mem_1024x2" blif_model=".subckt single_port_ram"
   class="memory" num_pb="1">
  <input name="addr" num_pins="10" port_class="address"/>
  <input name="data" num_pins="2" port_class="data_in"/>
  <input name="we" num_pins="1" port_class="write_en"/>
  <output name="out" num_pins="2" port_class="data_out"/>
  <clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>
```

It may occur to the reader that an alternative to introducing the *class* and *port_class* attributes would be to require that the architect give specific pre-defined names to

pb_types and ports. We considered that approach, however, we deemed it overly restrictive. With the proposed class and port class scheme, the architect is free to name pb_types and ports any way he/she likes, which enhances readability and may ease integration with other tools that use different naming conventions.

3.2.3 Intra-Block Interconnect

Having introduced how physical blocks are modeled in UTFAL, we move on to describe how the ports/pins on physical blocks can be connected to one another. Interconnect is specified using an *interconnect* element that is declared *within* a parent physical block type. Within the interconnect element in UTFAL, there are three main ways to create connectivity between ports/pins, inspired by the most common interconnect structures present in commercial FPGAs:

1. *direct*: This is a direct connection from one set of pins to another set of pins. This is used to model single metal wires or buses that have no programmability or switching..

2. *mux*: This is a multiplexed connection of multi-bit (bus) signals. It is assumed that signals internal to the FPGA control the select inputs of the multiplexer. That is, this construct represents a bus-based multiplexer whose input-to-output path is set during FPGA configuration.

3. *complete*: This represents a complete crossbar switch from a set of inputs pins to a set of output pins. As in the case of mux, it is assumed that the particular input pin that is matched with a particular output pin is controlled by signals internal to the FPGA whose values are set during device configuration.

The input and output pins of interconnect elements are specified by one *input* attribute and one *output* attribute declared within the interconnect element. The *direct*

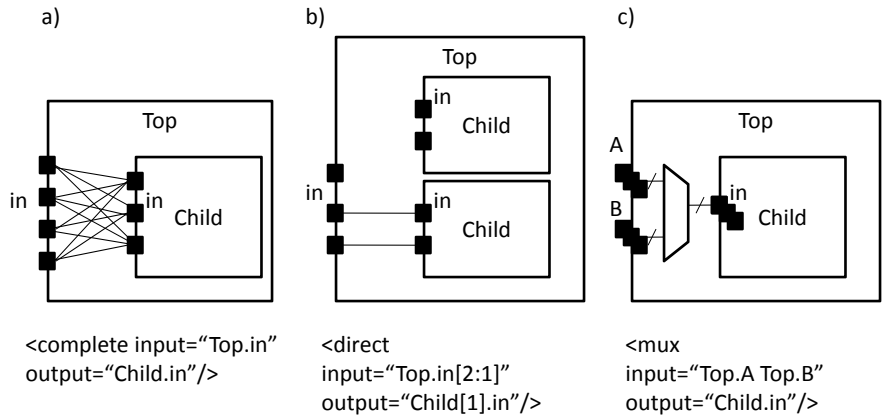

Figure 3.2: Examples of the three interconnect types: a) complete b) direct c) mux

element has one set of pins for its inputs and another set of pins for its outputs. The *mux* element has multiple sets of pins for its input and one set of pins at its output. Each set of input pins is delimited by a space. The *complete* element has one set of pins for its input and one set of pins for its output.

A set of pins to be connected is specified by first selecting physical blocks that are to be connected, and then specifying the desired pins on those blocks: In the case of there being multiple instances of a physical block, the following syntax is used:
<name of *pb_type*>[<starting index of physical block>:<ending index of physical block>]
Physical blocks are indexed from 0 to *num_pb - 1*. If only one physical block is selected, then the *:* and ending index may be skipped. If there is only one physical block, then the entire *[* to *]* may be skipped.

Having specified one or more blocks, pins are specified in the following way:
<name of port of physical block>[<starting index of pins>:<ending index of pins>]
Pin indices start from 0 and end at *num_pins - 1*. There is one shortcut for pin selection: if the architect wishes to select all the pins of a port, then he can skip the section from *[* to *]*.

Figure 3.2 gives examples of the three interconnect models in UTFAL: complete, direct and mux. Underneath each figure is the UTFAL code that produces the corre-

sponding interconnect. The examples assume that interconnect connectivity is *from* pins on a physical block called *Top to* pins on one of Top's child physical block. For the *complete* interconnect case (in Figure 3.2 a)), there is one physical block for each *pb_type* so only the *pb_type* is specified when selecting the blocks. All pins of the ports are used, so only the names of the ports are specified.

For the *direct* interconnect example (in Figure 3.2 b)), only the last two of the three *Top.in* pins are used so the corresponding UTFAL specifies the range of pins using *[2:1]*. There are two physical blocks of type *Child* and only the one with index 1 is used, so the UTFAL code includes a *[1]* in *Child[1]* to identify that block. This specification creates a one-to-one mapping between two input pins of *Top* and two input pins of *Child[1]*.

The *mux* interconnect example specifies a 3-bit 2-to-1 mux (in Figure 3.2 a)). The input attribute to the mux has two 3-bit pin sets. The first pin set is *Top.A* and the second pin set is *Top.B*. The two pin sets are separated by a space. The output of the mux is one 3-bit pin set of *Child.in*.

For ease-of-use, UTFAL provides a mechanism to concatenate sets of pins together. It follows a similar syntax to the concatenate construct in Verilog [7]. Sets of pins that are to be concatenated together are delimited by spaces and enclosed in { } as follows:

```
{set1 set2 ... setN}
```

An example of concatenation is shown in Figure 3.3. Two ports from *Top* connect to one port in *Child*. *Top.A* has one pin, *Top.B* has two pins. These two ports are then concatenated together {*Top.A Top.B*} and connected to the *Child.in* port.

A "scope" question naturally arises with the use of the interconnect element: in an arbitrary multi-level hierarchy of physical blocks, which ports/pins can be used within an interconnect element that is declared within a physical block at some specific level of the hierarchy? We take a straightforward approach to interconnect scope – the interconnect element can use pins of its parent physical block, or can use pins of physical block declared in the same level of the hierarchy.

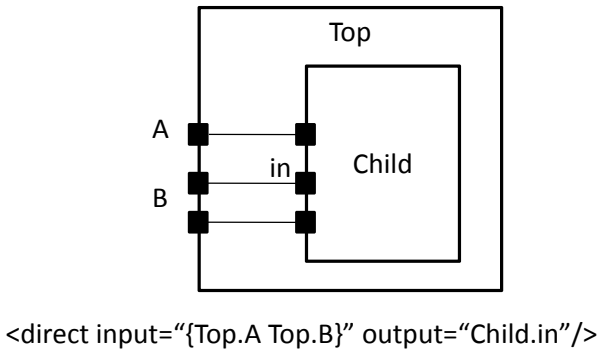

Figure 3.3: Concatenation example

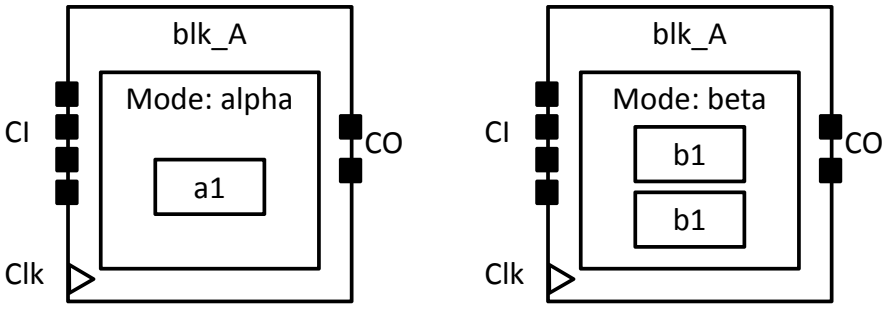

Figure 3.4: Example of a physical block with multiple modes of operation

3.2.4 Modes

A physical block in an FPGA may have multiple modes of operation and such modes are normally mutually exclusive. For example, consider an FPGA memory block that can be configured with different aspect ratios [37]. Each aspect ratio is one unique mode of operation for the memory. To represent the mode concept, UTFAL allows the definition of one or more *mode* elements within the *pb_type*. Multiple modes of operation are represented by multiple sibling *mode* elements declared within a parent *pb_type*. If a mode is declared, child physical blocks and interconnect can be declared *inside* the mode element, representing blocks (and connectivity) that is specific to the particular mode. In general, modes represent different ways of using a given piece of underlying FPGA hardware. A mode has one attribute *name* that serves as an identifier for the mode. Figure 3.4 shows a physical block with multiple modes of operation. The first mode is

called *alpha* and it contains one physical block *a1* and the second mode is called *beta* and contains two physical blocks of type *b1*. The corresponding UTFAL code is:

```
<pb_type name="blk_A">
  <input name="CI" num_pins="4"/>
  <output name="CO" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <mode name="alpha">
    <pb_type name="a1" num_pb="1">
    </pb_type>
  </mode>
  <mode name="beta">
    <pb_type name="b1" num_pb="2">
    </pb_type>
  </mode>
</pb_type>
```

Different modes can each have their own unique interconnect by declaring one or more *interconnect* elements as children of a *mode* element. For example, going back to the *blk_A* physical block in Figure 3.4, the *interconnect* element is declared inside each mode as follows:

```
<pb_type name="blk_A">
  <input name="CI" num_pins="4"/>
  <output name="CO" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <mode name="alpha">
    <pb_type name="a1" num_pb="1">
    </pb_type>
    <interconnect>
      <!-- declare mode alpha connections here -->
    </interconnect>
  </mode>
  <mode name="beta">
    <pb_type name="b1" num_pb="2">
    </pb_type>
    <interconnect>
      <!-- declare mode beta connections here -->
    </interconnect>
  </mode>
</pb_type>
```

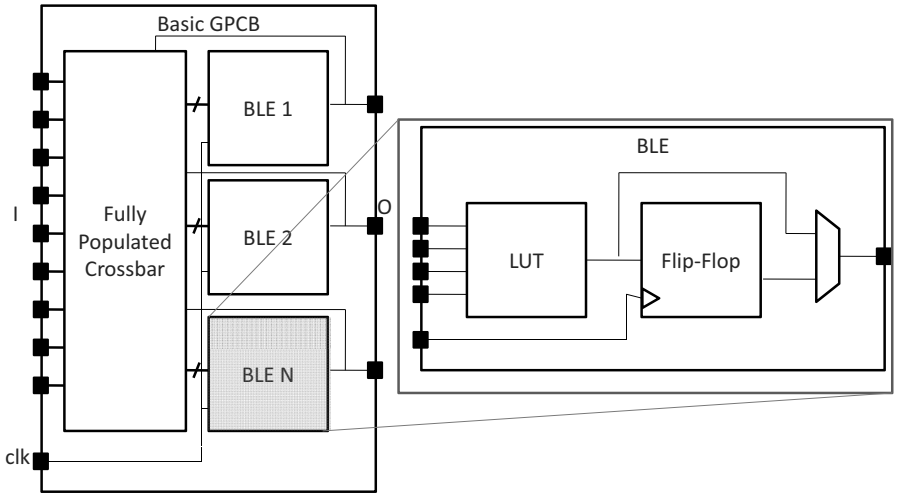

Figure 3.5: Basic general-purpose complex block and the BLE inside of it

3.3 More Complex Examples

This section presents two complex examples that illustrate how the UTFAL language can be used to model realistic complex blocks. The first use case is the basic GPCB; the second example is a configurable memory SPCB. Appendix A presents a use case for a fracturable mulitplier SPCB.

3.3.1 Basic GPCB

The first complex example is the basic GPCB shown in Figure 3.5. The parameters of this GPCB are N = 10, K = 4, and I = 22. It contains a full crossbar that connects to the inputs of the 10 BLEs. Each BLE consists of a 4-input lookup table (LUT) and a flip-flop. A BLE can implement three configurations: a LUT, a flip-flop, or a LUT and flip-flop pair (where the LUT output drives the flip-flop input).

First, a complex block *pb_type* called CLB is declared with appropriate input, output and clock ports:

```
<pb_type name="clb">
  <input name="I" num_pins="22"/>
  <output name="O" num_pins="10"/>
  <clock name="clk"/>
```

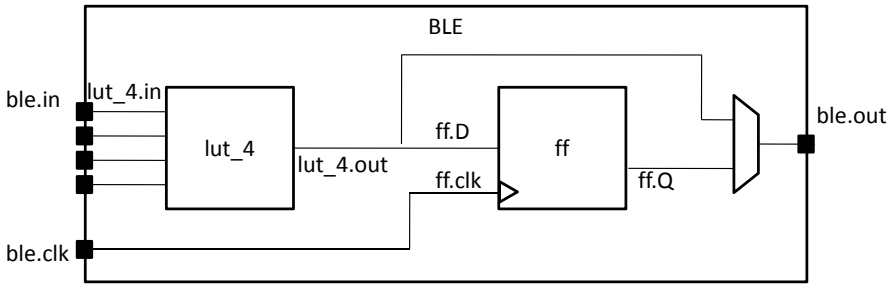

Figure 3.6: Port names and connections.

A CLB contains 10 BLEs. Each BLE has 4 inputs, one output, and one clock. A BLE block and its inputs and outputs are specified as follows:

```
<pb_type name="ble" num_pb="10">
  <input name="in" num_pins="4"/>
  <output name="out" num_pins="1"/>
  <clock name="clk"/>
```

A BLE consists of one LUT and one flip-flop (FF). Both of these are primitives. Recall that primitive physical blocks must have a *blif_model* attribute that matches with the model name in the BLIF input netlist. For the LUT, the model is ".names" in BLIF. For the FF, the model is ".latch" in BLIF. The class construct denotes that these are special (common) primitives. The primitives contained in the BLE are specified in UTFAL as:

```
  <pb_type name="lut_4" blif_model=".names" num_pb="1" class="lut">
    <input name="in" num_pins="4" port_class="lut_in"/>
    <output name="out" num_pins="1" port_class="lut_out"/>
  </pb_type>
  <pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>
```

Figure 3.6 shows the ports of the BLE with the input and output pin sets. The inputs to the LUT and flip-flop are *direct* connections. The multiplexer allows the BLE output to be either the LUT output or the flip-flop output. The code to specify the interconnect is:

```
  <interconnect>
```

```
      <direct input="lut_4.out" output="ff.D"/>
      <direct input="ble.in" output="lut_4.in"/>
      <mux input="ff.Q lut_4.out" output="ble.out"/>
      <direct input="ble.clk" output="ff.clk"/>
    </interconnect>
  </pb_type>
```

Finally, the CLB interconnect is modeled (see Figure 3.5). The inputs to the 10 BLEs (*ble[9:0].in*) can be connected to any of the CLB inputs (*clb.I*) or any of the BLE outputs (*ble[9:0].out*) by using a full crossbar. The clock of the CLB is wired to multiple BLE clocks, and is modeled as a full crossbar. The outputs of the BLEs have direct wired connections to the outputs of the CLB and this is specified using one *direct* tag. The CLB interconnect specification is:

```
  <interconnect>
    <complete input="{clb.I ble[9:0].out}" output="ble[9:0].in"/>
    <complete input="clb.clk" output="ble[9:0].clk"/>
    <direct input="ble[9:0].out" output="clb.O"/>
  </interconnect>
</pb_type>
```

The complete specification of this complex block is given in Appendix A. Note that this example does not contain carry-chains. Carry-chains are common in GPCBs and we plan to support carry-chains and delay information in future work.

3.3.2 Fracturable Memory Cluster

The second complex example is the single-ported memory shown in Figure 3.7. It is reconfigurable with different width and depth configurations. The inputs can be either registered or combinational. Similarly, the outputs can be either registered or combinational. Also, each memory configuration has groups of pins called ports that share common properties. Examples of these ports include address ports, data ports, write enable, and clock.

In this example, the block memory has the following three configurations: 2048x1, 1024x2, and 512x4, which will be modeled in UTFAL using modes. We begin by declaring

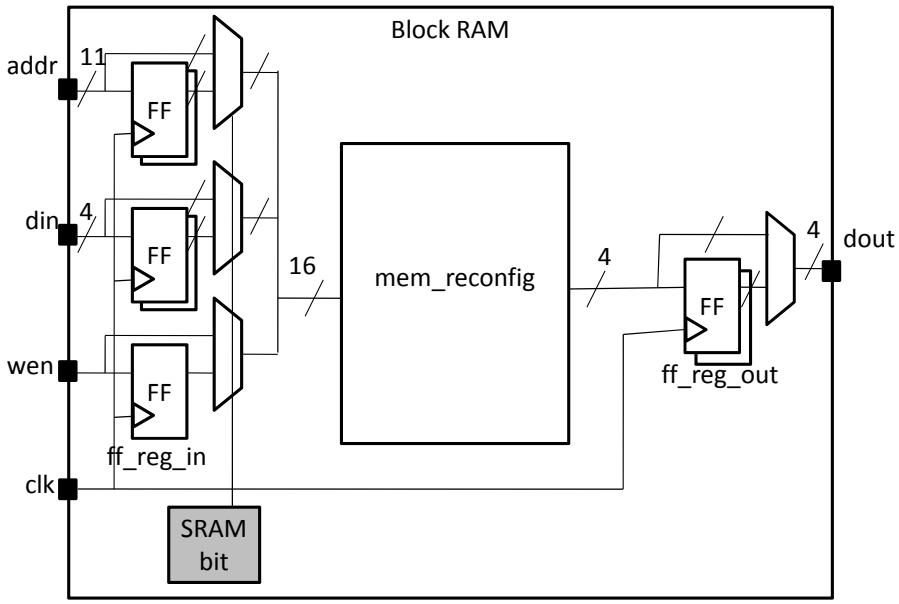

Figure 3.7: Example of an embedded block RAM.

the reconfigurable block RAM along with its I/O as follows:

```
<pb_type name="block_RAM">
  <input name="addr" num_pins="11"/>
  <input name="din" num_pins="4"/>
  <input name="wen" num_pins="1"/>
  <output name="dout" num_pins="4"/>
  <clock name="clk"/>
```

The input and output registers are defined as 2 sets of bypassable flip-flops at the I/Os of the block RAM. There are a total of 16 inputs that can be registered as a bus so 16 flip-flops (named *ff_reg_in*) must be declared. There are 4 output bits that can also be registered, so 4 flip-flops (named *ff_reg_out*) are declared:

```
  <pb_type name="ff_reg_in" blif_model=".latch" num_pb="16"
     class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>
  <pb_type name="ff_reg_out" blif_model=".latch" num_pb="4"
     class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>
```

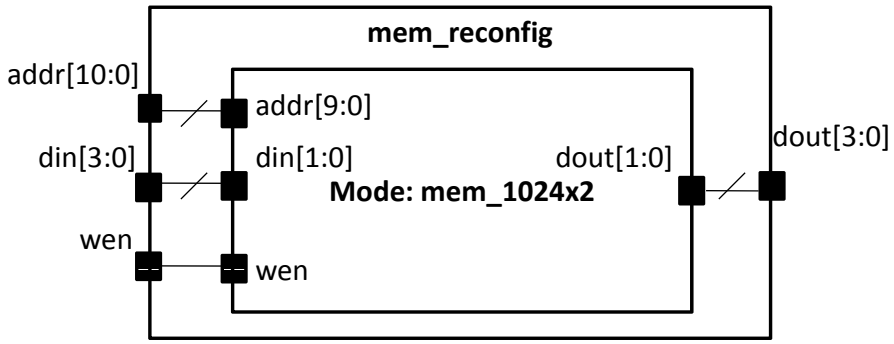

Figure 3.8: The mem_reconfig mode representing a 1024x2 RAM.

Each aspect ratio of the memory is declared as a mode within the memory physical block type as shown below. Also, observe that since memories are one of the special (common) primitives, they each have a *class* attribute:

```
<pb_type name="mem_reconfig" num_pb="1">
  <input name="addr" num_pins="11"/>
  <input name="din" num_pins="4"/>
  <input name="wen" num_pins="1"/>
  <output name="dout" num_pins="4"/>

  <!-- Declare a 1024x2 memory type -->
  <mode name="mem_1024x2_mode">
    <pb_type name="mem_1024x2" blif_model=".subckt sp_mem"
       class="memory">
      <input name="addr" num_pins="10" port_class="address"/>
      <input name="din" num_pins="2" port_class="data_in"/>
      <input name="wen" num_pins="1" port_class="write_en"/>
      <output name="dout" num_pins="2" port_class="data_out"/>
    </pb_type>
    <interconnect>
      <direct input="mem_reconfig.addr[9:0]"
         output="mem_1024x2.addr"/>
      <direct input="mem_reconfig.din[1:0]" output="mem_1024x2.din"/>
      <direct input="mem_reconfig.wen" output="mem_1024x2.wen"/>
      <direct input="mem_1024x2.dout"
         output="mem_reconfig.dout[1:0]"/>
    </interconnect>
  </mode>

  <!-- Declare a 2048x1 memory type -->
  <mode name="mem_2048x1_mode">
    <!-- Follows the same pattern as the 1024x2 memory type declared
       below -->
  </mode>

  <!-- Declare a 512x4 memory type -->
  <mode name="mem_512x4_mode">
```

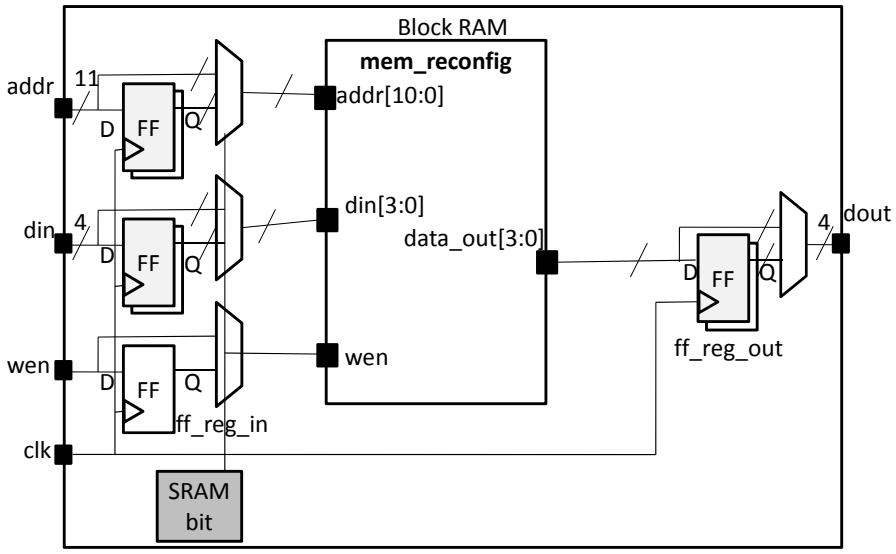

Figure 3.9: Routing connections for block RAM.

```
      <!-- Follows the same pattern as the 1024x2 memory type declared
          above -->
    </mode>

  </pb_type>
```

The top-level interconnect structure of the memory SPCB is shown in Figure 3.9. The inputs of the SPCB can connect to input registers or bypass the registers and connect to the combinational memory directly. Similarly, the outputs of the combinational memory can either be registered or connect directly to the outputs. The UTFAL description of the interconnect is as follows:

```
1   <interconnect>
2     <direct input="{block_RAM.wen block_RAM.din block_RAM.addr}"
          output="ff_reg_in[15:0].D"/>
3     <direct input="mem_reconfig.dout" output="ff_reg_out[3:0].D"/>
4     <mux input="mem_reconfig.dout ff_reg_out[3:0].Q"
          output="block_RAM.dout"/>
5     <mux input="{block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]}
           ff_reg_in[15:0].Q"
6           output="{mem_reconfig.wen mem_reconfig.din
              mem_reconfig.addr}"/>
7     <complete input="block_RAM.clk" output="ff_reg_in[15:0].clk"/>
8     <complete input="block_RAM.clk" output="ff_reg_out[3:0].clk"/>
9   </interconnect>
10 </pb_type>
```

The interconnect for the bypassable registers is complex and so we provide a more detailed explanation. First, consider the input registers. Line 2 shows that the SPCB inputs drive the input flip-flops using direct wired connections. Then, in line 5, the combinational configurable memory inputs {*mem_reconfig.wen mem_reconfig.din mem_reconfig.addr*} either come from the flip-flops *ff_reg_in[15:0].Q* or from the SPCB inputs {*block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]*} through a 16-bit 2-to-1 bus-based mux. Thus completing the bypassable input register interconnect. A similar scheme is used at the outputs to ensure that either all outputs are registered or none at all.

The complete memory specification is given in Appendix A.

3.4 Summary

This chapter described a new FPGA architecture description language, UTFAL, and how this language can be used to describe the complex blocks of an FPGA architecture. UTFAL can specify an arbitrary hierarchy of clusters and primitives inside a complex block. The language also provides a library for common primitives on an FPGA such as LUTs, flip-flops, and memories. Physical blocks need to connect to each other within a complex block. UTFAL uses three interconnect constructs to specify those connections. The language can express different modes of operation for physical blocks and different interconnect for each mode of operation. Lastly, we gave two complex examples that illustrate how to use UTFAL to describe a basic GPCB and a configurable memory SPCB.

Note that a complete FPGA architecture language needs to cover FPGA structures outside of a complex block, such as the interconnect that joins complex blocks together. UTFAL specifies these structures using the same description language as that found in VPR 5.0 [30] and is defined in [31].

The next section describes a new tool that supports architectures described with

UTFAL.

Chapter 4

Packing Algorithm for Heterogeneous FPGAs

In this chapter, we propose a novel generic packing tool for FPGAs called Architecture-Aware Packer (AAPack). Recall that the input to a packer are the components of a technology-mapped circuit (such as a circuit described in LUTs and flip-flops). The output is the grouping of those components into the complex blocks available on an FPGA. AAPack was developed to enable exploration into a much wider space of FPGA logic block architectures than is possible with prior packers.

The chapter starts by giving the scope and context for the new algorithm, and then presents the algorithm.

4.1 Introduction

Packing links the technology-mapping stage with the placement stage of an FPGA CAD flow (recall the CAD flow shown in Figure 2.9). Furthermore, packing requires information about the FPGA architecture and perhaps some additional user options. As a result, AAPack has three inputs and one output as shown in Figure 4.1. The first input is a technology-mapped circuit netlist; we use the BLIF input format [3]. The second input is

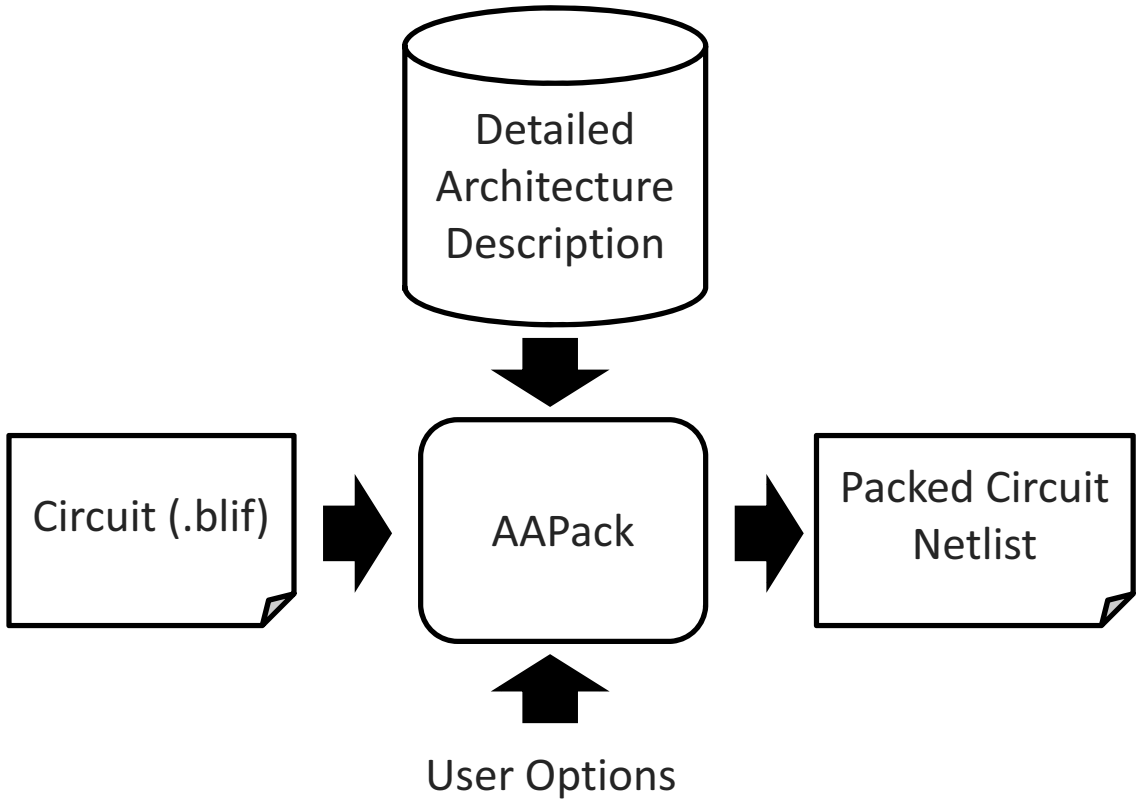

Figure 4.1: Inputs and outputs of AAPack

a description of the FPGA architecture. This description uses the UTFAL language presented in Chapter 3. The third type of input are the user options, which set parameters of the algorithm such as the weights of various metrics on internal cost functions within the algorithm. The output of AAPack is a netlist of complex blocks (and the contents of those blocks) that is used by the downstream placement stage.

4.2 Scope

The FPGA architecture description language presented in Chapter 3 can describe an very large class of complex logic blocks, up to and including a block that itself looks like a complex FPGA. While the intent of this work is to ultimately work well for any such block, it is difficult to immediately build a packer that gives high-quality results given any FPGA architecture. So, in this initial work we decided to limit the scope of AAPack such

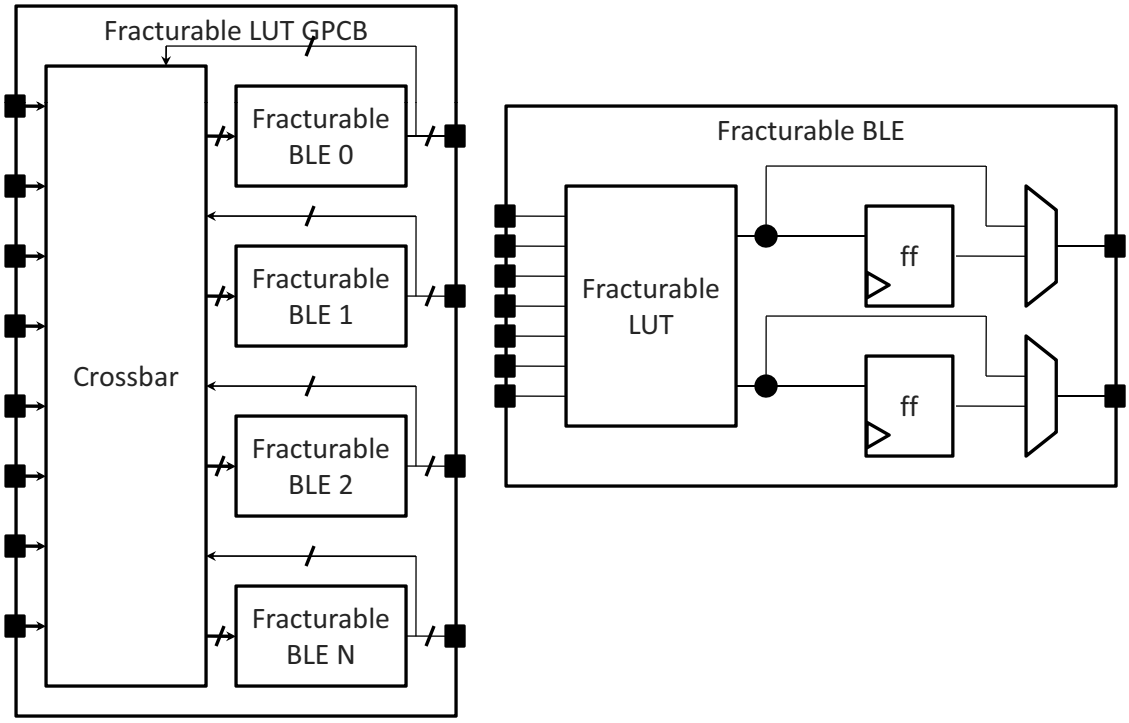

Figure 4.2: Fracturable LUT GPCB

that the packing problem becomes more tractable while still providing new and useful exploration capability. For the present research, the goals/limits are as follows: First, we will focus on area-driven packing only. Although timing-driven packing is important, it will be left for future work. Second, we limit the types of complex blocks that AAPack supports to the following:

1. The types of GPCB shown in Figure 4.2. This complex block is similar to the basic GPCB discussed in Chapter 2 but with two additional architectural features. The first feature is fracturable LUTs. These fracturable LUTs have two outputs and are contained in a fracturable BLE that has one or two flip-flops (of which the more common two flip-flop case is shown). The second feature is a crossbar that may be depopulated. These are two major features that may be found in modern FPGAs.

2. A memory block SPCB, in which the memory can be configured into different aspect ratios. We will assume that these are synchronous memories. There are no

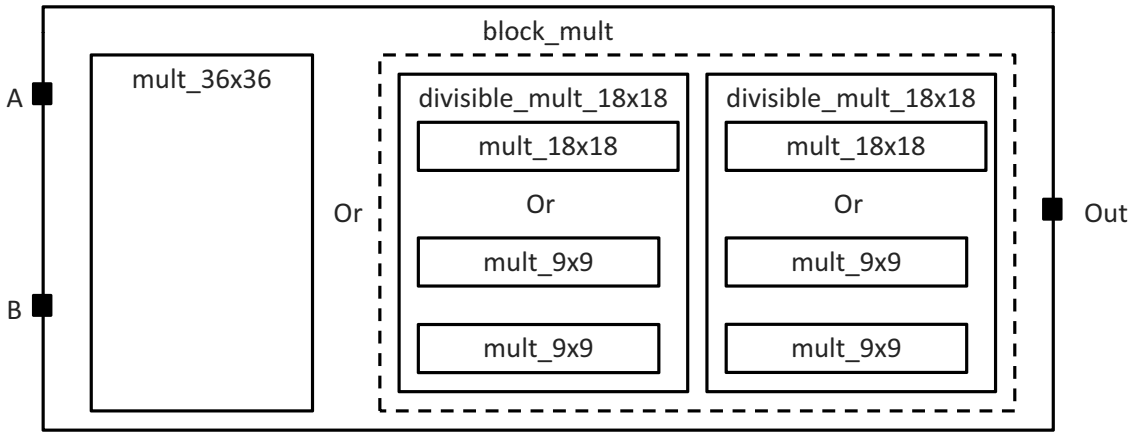

Figure 4.3: Fracturable multiplier CLB

intra-cluster interconnect between different memories. Configurable memories are a fundamental capability in modern FPGAs so a packer that can give high quality support for them is essential.

3. A fracturable multiplier SPCB that is composed of one multiplier that can be divided into two smaller multipliers. Each of these smaller multipliers can be further fractured into two smaller, indivisible mulitpliers. For example, a 36x36 multiplier example is shown in Figure 4.3. Here, a 36x36 multiplier can fractured down to two 18x18 multipliers which in turn can be fractured into two 9x9 multipliers. The input and output ports of the 36x36 are split such that the first half of the bits go to one 18x18 multiplier and the other half to its sibling. These ports are then again split in half for the 9x9 multipliers. There are no interconnect between sibling multipliers. Multipliers are common in modern FPGAs so support for them is important.

These three types of complex blocks are common in commercial FPGAs but so far have not been well supported in public-domain packers. Furthermore, the lack of tools that can support configurable memories and multipliers prevent modern, large, and useful designs, such as [32], from being used as benchmark circuits. As a result, we focused AAPack on these complex blocks to help make a contribution to the modernization of

FPGA architecture research.

There are many features that this first version of AAPack will not support, including complex blocks with bus-based routing or carry-chains. Due to time constraints, these important features will be left for future work. Lastly, AAPack is designed to be general enough to correctly pack architectures outside the specified scope, but the quality of those solutions are not examined as part of the present work.

4.3 The AAPack Packing Algorithm

The AAPack algorithm takes a greedy approach to selecting primitives from the user circuit to pack into complex blocks.

In addition to providing support for basic GPCBs, the AAPack algorithm must support three new kinds of architectural features that are described with the new UTFAL language presented in Chapter 3. The first complexity is support for arbitrary heterogeniety. The architectures modeled by the new language contain different types of complex blocks as well as different types of primitives and groups of primitives within a complex block. The AAPack algorithm needs to determine where in the complex block hierarchy to place a primitive. The other key, and new, capability of the algorithm is to comprehend the new more complex internal routing structures defined in the UTFAL language. For a primitive to be successfully packed into a block, there must be a way for its required netlist signals to route out to the pins of the block. Finally, the packer must select the correct mode of operation (also described in the new language) of each primitive.

4.3.1 Algorithm Overview

We begin with a high-level description of the AAPack algorithm. Initially, all primitives in the input circuit netlist are set to an *unpacked* state. AAPack first selects a seed primitive and an "open" complex block to place that seed into. It then continues looking

for companion primitives, based on a cost function, to pack into the complex block until no more can be found. It then closes that complex block and opens a new one and continues until there are no more unpacked netlist primitives. Note that there is no pre-packing stage in AAPack, rather it operates directly on the technology-mapped input circuit. The following pseudocode gives the high-level key stages of the AAPack routing algorithm:

```
1 while(exist_unpacked_netlist_blocks()) {
2   current_complex_block = start_new_complex_block();
3   while(exists_candidate_block_for_complex_block(current_complex_block)) {
4     try_pack_candidate_block_into_complex_block(current_complex_block);
5   }
6 }
```

AAPack begins by selecting a new complex block in its *start_new_complex_block* function. This function selects a seed primitive, the complex block to pack that primitive into, and determines if that primitive can be legally packed into the complex block.

The inner loop of the algorithm, beginning at line 3 of the pseudocode, looks for additional primitives to pack into the complex block. The primitive is selected from a candidate list of primitives based on a cost function which we will describe later. The selected primitive is tested for *legality* which determines if the primitive can be routed successfully within the complex block. If it cannot, then the primitive is rejected.

Once a complex block is full, or if there are no more candidate blocks, the inner loop exits, and a new complex block is opened. The algorithm finishes when there are no unpacked primitives left.

One of the key capabilities of the new algorithm is its ability to handle complex blocks with arbitrary levels of hierarchy. The next section provides an example of how multiple levels of hieararchy are packed, before proceeding on with a more detailed explanation of the algorithm.

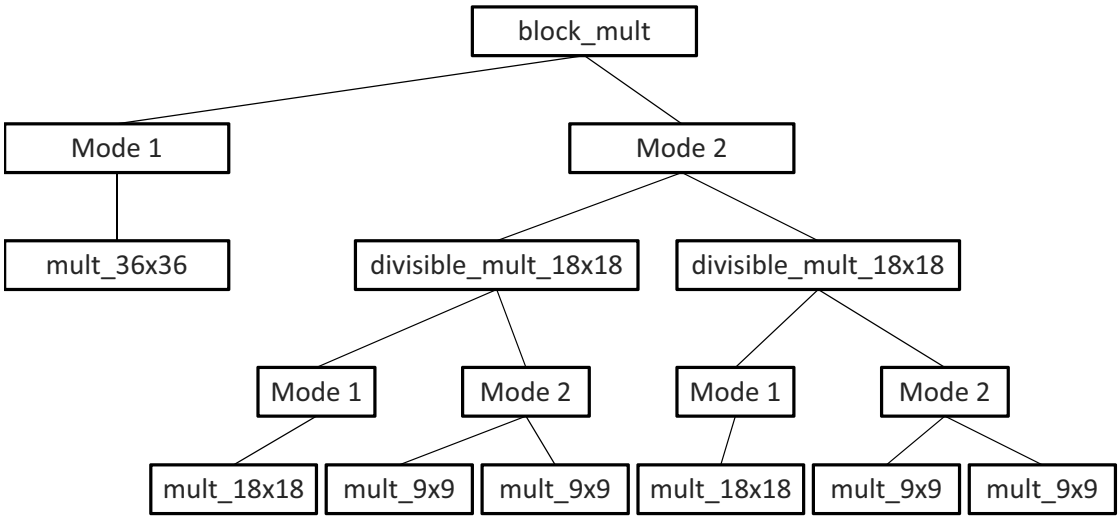

Figure 4.4: A tree representation of a complex block hierarchy

4.3.2 Dealing with Arbitrary Hierarchy

Recall that we think of the hierarchy of the complex block as a hierarchy of *clusters*. The highest-level cluster is the complex block itself, and then child clusters are contained within that block, and then clusters within those, recursively, until there is a leaf cluster that contains only architectural primitives. (These architectural primitives match directly to the primitives that are described in the netlist). A physical block refers to a complex block, a cluster, or a primitive.

We model the hierarchy of a complex block as a tree; the nodes of the tree are either physical blocks or a mode indicator. The edges represent parent-child relationships in the hierarchy. Modes at any given level are mutually-exclusive. Consider the multiplier SPCB illustrated in Figure 4.3. The tree representation of it is illustrated in shown in Figure 4.4. Note that multiplier tree is unbalanced because the 36x36 multiplier is closer to the top-level than the 9x9 or 18x18 multipliers resulting in one half of the tree being deeper than the other half. An algorithm that packs to this complex block must be able to manage unbalanced hierarchies.

As described in the general algorithm earlier, the algorithm "opens" a complex block, and then tests different candidates for inclusion into the block. A key question is to

determine which level in the hierarchy to attempt to place the candidate. This is done using a depth-first traversal of the tree, and is illustrated by the following example: Consider a user circuit that contains one 16x16 multiplier primitive and one 8x8 multiplier primitive, and that the FPGA contains the multiplier SPCB shown in Figure 4.3.

Assume that the first candidate primitive selected for potential inclusion in the block is the 16x16 multiplier. The algorithm will perform a depth-first traversal of the tree to find a suitable matching block. Beginning at the top, with block_mult, it traverses the tree in a depth first manner, as illustrated in by the arrows and numbers in Figure 4.5. The traversal eventually discovers, at number 6, the 18x18 physical multiplier first and so it packs the 16x16 multiplier into the 18x18 physical multiplier. Figure 4.5 indicates how the algorithm traverses this tree and the shaded boxes indicate the physical blocks that are used after packing is complete.

Subsequently, the algorithm walks back up the tree from the 18x18 multiplier until it reaches a level that contains other empty primitives, namely *block_mult*. It then selects the 8x8 circuit mulitplier candidate, and descends another part of the hierarchy. Ultimately it packs the 8x8 multiplier into a 9x9 physical multiplier. There are no more multipliers in the user circuit so the complex block is packed. Note that this example only describes how the algorithm selects which part of the hierarchy to use, and not the legality checking, which is described later in subsection 4.3.5.

A standard depth-first traversal occasionally leads to poor packing decisions and so we modify this traversal in AAPack to improve quality. The following example shows the problem with a standard depth-first traversal. Consider Figure 4.6, it shows a cluster with 2 LUTs and a flip-flop along with a user circuit where the shaded LUT in the circuit is packed into the shaded LUT in the cluster. Suppose the flip-flop is selected as the next candidate block to pack into the cluster. The flip-flop cannot pack into the cluster because of the constraints imposed by the interconnect so it gets rejected. In a standard depth-first traversal, the algorithm will not revisit this cluster again because it

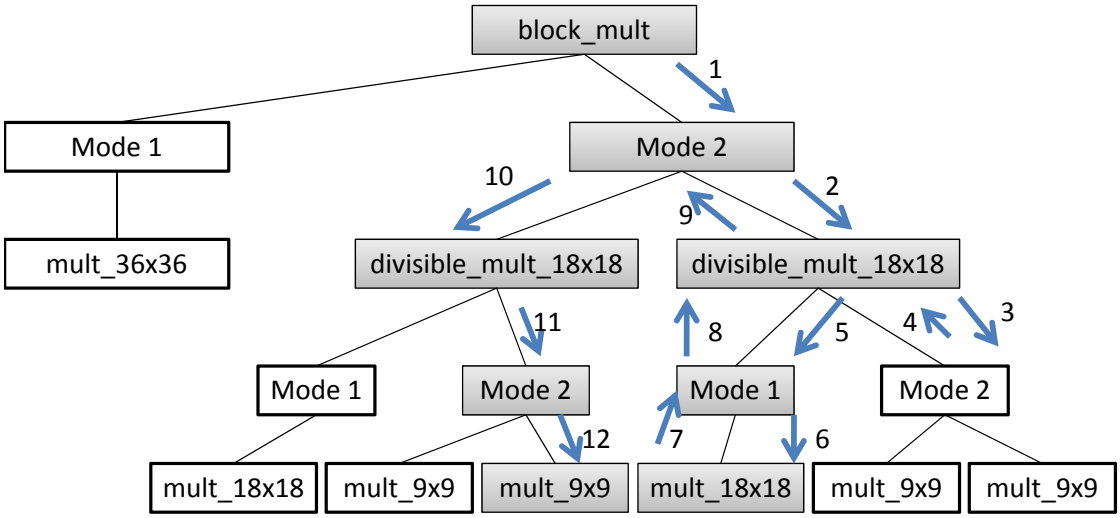

Figure 4.5: Mapping multiplier blocks from a circuit to a complex block hierarchy

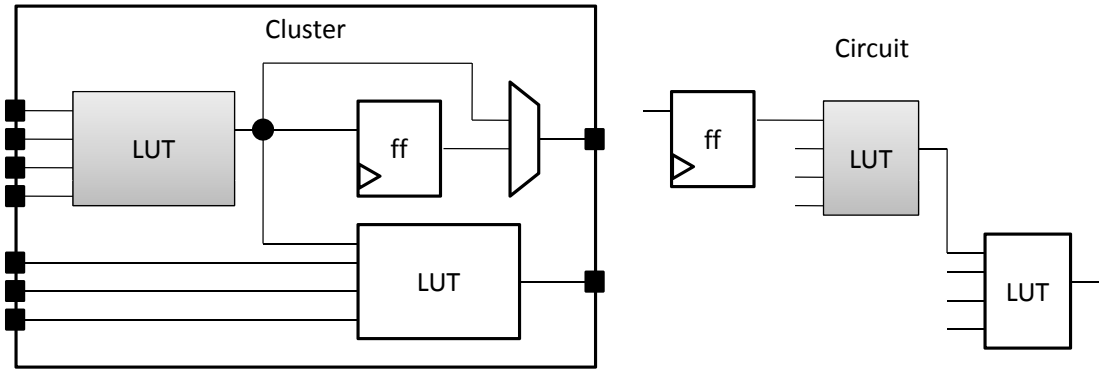

Figure 4.6: Example where interconnect limits what blocks can be packed into the cluster

has already been visited. Consequently, the LUT at the output of the packed LUT will not get packed into the cluster. We modified the depth-first traversal to try up to three different candidates for each cluster before traveling to a different part of the tree. Thus it will pack the second LUT into the cluster after failing to pack the flip-flop.

Lastly, to save runtime, there are two pruning methods employed by this traversal. First, other than the target cluster, any other cluster that contains packed primitives are skipped. This means that a packed cluster in a complex block will only be visited once by the AAPack algorithm. Second, every cluster keeps track of the primitive type(s) that it contains. So, the traversal will not explore a physical block that does not have a

matching primitive type for the candidate.

4.3.3 Dealing with Heterogeneity: Matching Supply and Demand

One of the key features of the new algorithm is that it can handle the heterogeneous nature of the FPGA. The key here is that there is a fixed number (or a fixed ratio) of each type of block *supplied* by the FPGA. The goal of the packer is to use all of the blocks present on the FPGA, if possible; this depends on the *demand* presented in the netlist. This can only be possible if the input netlist either contains the primitives in the same ratio as present on the chip (the demand matches the supply), *or* if a primitive in the netlist can map to one or more of the supplied complex blocks. In the latter case, a packing algorithm must carefully select which block to pack into in order to match the demand to the supply.

In this initial packing algorithm development, we will focus only on the case that is typically used in architecture research, in which the size of the FPGA being packed to "floats" to match the size of the input netlist. We do this by growing the size of the FPGA as additional GPCBs or SPCBs are opened as described above. The architecture description language indicates how many of each type of complex block are present, by specifying which columns contains which blocks. So, when the FPGA "grows" as new blocks are added, this specification is respected. Thus, if a new kind of block is needed, but one is not available, the FPGA size is increased until a new becomes available.

Figure 4.7 illustrates this case where the FPGA is too small and a bigger one is needed. Assume that there are two types of block, A and B, but B is specified to be only in every third column. When the size of the FPGA contains only four blocks, there is no third column. So, when a primitive that can only pack into complex block type B, the FPGA is "grown", as shown, and then contains three blocks of type B.

Note that this method of growing the FPGA in a somewhat greedy response to the

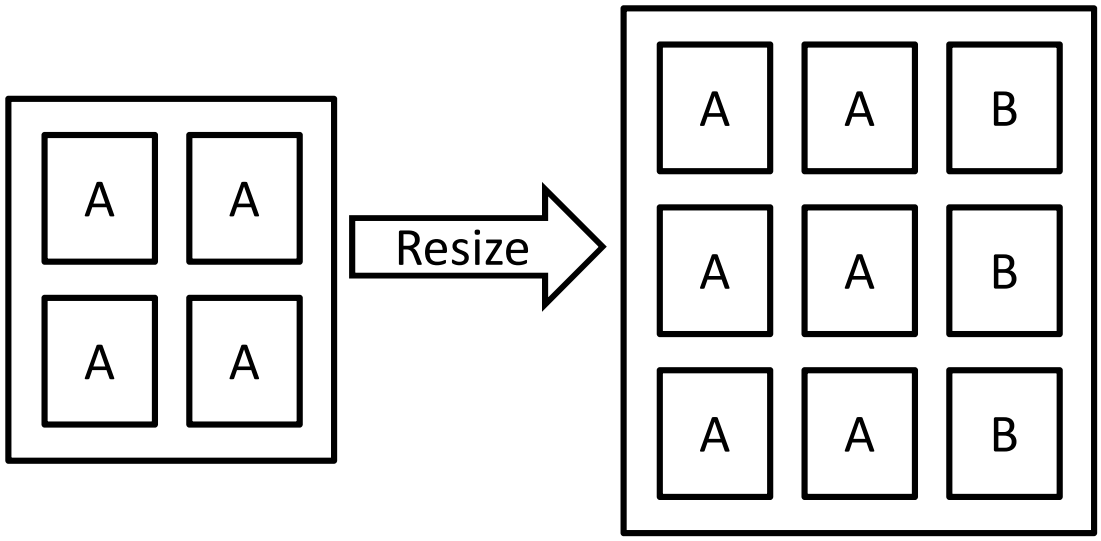

Figure 4.7: If an FPGA is too small, AAPack tries a larger FPGA

netlist's demand for a specific type *does not* attempt to balance the target complex block when there may be two possibilities. This is left as future work.

4.3.4 Algorithm Details

We now provide more detail of the algorithm. We need to select a new complex block. To do this, we first select a seed from the set of all unpacked primitives. Then, an empty complex block that can legally pack that seed is selected. That complex block, with the seed packed into it, becomes the "open" complex block and is returned by the function.

Select Seed

The primitive with the most number of unique input nets is selected to serve as a seed. This criteria for seed selection is the same as the one used by VPack [4]. The rationale behind this criteria is as follows: A primitive with a high number of input nets may have a higher demand for input pins. Input pins of a cluster are often limited and so it will likely be harder to pack this primitive into an existing complex block because there may not be enough input pins available. Hence, this primitive should serve as a seed to avoid

the difficulties of packing it into a pre-existing complex block.

Select New Complex Block

As described above, the Select New Complex Block function must deal with the heterogeneity of the FPGA, growing it to find blocks that can house the primitives in the netlist. The actual growing of the FPGA occurs when the seed cannot be packed into any of the currently available complex blocks. It should be noted, that, depending on the specified supply of the complex blocks, that the FPGA will have to grow until one of the right type appears.

Try Pack Candidate Block Into Complex Block

This function takes as input the open complex block. It selects one primitive from the netlist, based on a cost function, and tries to pack that primitive into the current open complex block. The first step of the Try Pack function is to select a cluster inside the complex block, and this is done using the depth-first travels described in subsection 4.3.2 above. It then selects a candidate block. Afterwards, the algorithm tries to pack the candidate into the target cluster with the *try_add_block* function.

Select Candidate Block

Recall that the algorithm performs a traversal of the hierarchy of clusters (and their modes) within a complex block. So this function returns a candidate block to pack into a target cluster within the complex block hierarchy. The pseudocode for the function is as follows:

```
1 get_next_netlist_block(target_cluster, complex_block) {
2   candidate =
        select_lowest_cost_candidate_block_that_passes_quick_legality_check();
3   if(not exists(block)) {
4      candidate =
           unpacked_block_with_most_unique_nets_and_passes_quick_legality_check();
5   }
```

```
6   return(candidate);
7 }
```

The function first tries to find an unpacked primitive that has the lowest cost function and has a good chance of legally packing into the target cluster (i.e. passes a quick legality check). During this initial stage, unpacked primitives that share no common nets with the open complex block are not considered. If the first stage cannot find a suitable primitive, then the unrelated primitive that passes a quick legality check and has the most number of unique nets is selected.

We describe the cost function used to select candidate blocks and the details of the quick legality check in the next sections.

Candidate Block Selection Cost Function

A list of candidate primitives for a complex block are ranked based on a cost function as follows:

```
1 affinity = (a * net_absorption_gain + (1 - a) *
    num_shared_nets)/num_input_pins_of_candidate
2 cost = 1/affinity
```

Affinity measures the "attraction" that an unpacked primitive has with a complex block. The cost of a candidate is the inverse of the affinity for that candidate; a small affinity means a high cost and a high affinity means a low cost. Affinity for a primitive is determined by the nets of that primitive and the relationship those nets have with the "open" complex block. Affinity is determined by 3 variables and they are as follows:

- *net_absorption_gain* increases the affinity of a primitive if it contains nets that are close to being absorbed into the complex block. *net_absorption_gain* is calculated as follows: For each net of a primitive, add *1 / number of connections outside complex block* to *net_absorption_gain*. Thus, a net with most of its connections in a complex block provide higher affinity than a net with connections outside the complex block.

- *num_shared_nets* increases the affinity of a primitive if it has nets that are shared with the complex block. Each net adds one to *num_shared_nets*. This is the same affinity metric as the one found in [4].

- *num_input_pins_of_candidate* normalizes affinity between different types of primitives based on their number of inputs because primitives with more input pins have a higher chance of getting a higher *net_absorption_gain* or *num_shared_nets* than primitives with fewer pins.

Lastly, 'a' is a constant that controls the weighting of the affinity variables. We default the value of 'a' to 0.9 as we found that this provides good quality of results.

Quick Legality Checking

A quick legality check is employed to remove infeasible candidates. This check employs two rules as follows:

1. There exists at least one empty physical primitive in the target cluster that the candidate block can map to and this primitive does not violate any previously set modes.

2. There exists sufficient input and output pins available on the target cluster, and all its ancestor clusters, to meet any increase in pins demanded by the candidate.

The quick legality check does not eliminate all infeasible candidates so a more complete legality check is performed later in the algorithm. A complete legality check consumes significantly more runtime than the quick check so filtering candidates with a quick check saves runtime.

4.3.5 Try Add Block

This function takes as input a candidate primitive, the current complex block, and a target cluster within the complex block. The output is a pass or fail flag indicating

whether or not the primitive legally packed into the candidate cluster of the complex block.

This function first finds a location within the complex block to place the candidate primitive. This procedure follows the tree traversal described earlier in subsection 4.3.2. For a given location, it also needs to route all the nets inside the complex block, including the nets of the newly place primitive. The pseudocode for this function is as follows:

```
1 try_add_block(target_cluster, complex_block, candidate) {
2   for each legal placement i {
3     if (routing(i, complex_block, candidate) == success) { return PASS }
4   }
5   return FAIL;
6 }
```

Routing

Routing determines the interconnect configurations of a complex block such that the nets within it are implemented. It returns *true* if such a configuration exists, *false* otherwise. First, a representation of the interconnect is described. Then, the routing algorithm that operates on this representation is presented.

The routing stage uses a graph representation of the interconnect of a complex block. The interconnect edges are modelled as directed edges and the pins of clusters are modelled as nodes. Inputs and outputs pins of primitives connect to sinks and sources respectively where one pin connects to one source/sink. Inputs and outputs of the complex block connect to sources and sinks respectively. There are an equal number of sources as there are input pins to the complex block and each source represent a potential external net that can connect into the complex block. Each of these sources has an edge to every input pin to represent the full connectivity of external routing. Similarly, there are an equal number of output sinks as there are output pins and each output pin has an edge to every output sink. Lastly, there is an edge from every output pin of a complex block to every input pin to represent cases where a complex block must use external interconnect

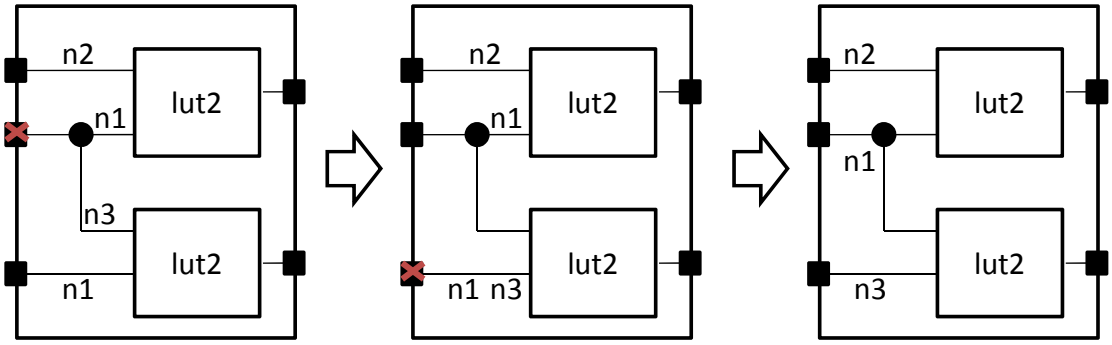

Figure 4.8: Negotiated congestion routing to resolve congestion conflicts

to connect between points within the complex block. This graph is known as the *routing graph* of the complex block.

The routing algorithm determines the interconnect configurations in order to implement all nets currently inside the complex block. The algorithm first maps the start and endpoints of each net to their corresponding sources and sinks in the routing graph of the complex block. It then determines the interconnect path for each net using the breadth-first negotiated congestion routing algorithm in VPR [5]. A negotiated congestion routing algorithm first routes all nets allowing some nets to share the same wire resulting in a conflict. It then repeatedly rips up a net and re-routes the net based on a cost function that tries to remove wire conflicts. Figure 4.8 gives an example of this type of routing algorithm for the input nets of a physical block containing two 2-LUTs with one shared input. The top LUT has nets *n1* and *n2* while the bottom LUT has nets *n1* and *n3*. The first route has a wire conflict with the middle input pin because that pin is used by *n1* and *n3*. Net *n3* is re-routed to avoid the middle pin but ends up conflicting with the bottom pin. Then *n1* is re-routed and takes the middle pin thus removing all wire conflicts resulting in a legal routing. At which point, the router returns *true*.

We set the breadth-first search routing in VPR with the following parameters:

- Maximum router iterations: 20
- Present congestion cost (cost of conflicting wires): 10

- Increase present congestion cost by 10x after the first iteration
- Increase present congestion cost by 2x for each subsequent interation

These parameters were selected to make the router converge faster than the default settings.

If a legal route exists, the function returns *true*; otherwise, the function returns *false*.

4.3.6 Dealing with Memories

Supporting packing for memories presents an unusual scenario for packing and requires some special handling in the algorithm. A memory instantiated in a user circuit may have an aspect ratio that does not fit onto any one physical memory of an FPGA architecture and is thus implemented using multiple physical memories.

The AAPack algorithm requires that memories in the input netlist be specified as one bit-wide memories that do not exceed the depth of the largest physical memory in the FPGA[1]. For example, if a design contained a 256 x 8 memory, the input netlist would contain eight 256 x 1 memories that, ultimately, will end up residing together in one or more RAM primitives. The total number of netlist primitives that can map into a memory primitive is equal to the size of its data width. This makes memory primitives different from other primitives because they can accommodate more than one netlist primitive.

For two memory blocks in the input netlist to be packed together into one memory primitive two requirements must be met: 1) the memory blocks in the netlist must have the same address bus width, and 2) the signals on corresponding bits of the address bus and control signals must be identical.

Other than these differences, memories are packed following the same algorithm as the other primitives.

[1]The upstream tool ODIN II[21] is designed to meet this requirement

4.4 Error Checking

The AAPack algorithm performs error checking after packing to test if the final solution it generated is correct. Some basic checks that are done include checking that all netlist blocks have been packed and checking that there are no primitives or clusters that are overused. One key feature of the error checking employed in AAPack is that it can output a BLIF netlist of the packed circuit. This allows us to use combinational and sequential equivalence checkers, such as those available in ABC [44], to do a formal logic comparison between the input BLIF netlist and the output complex block netlist.

The BLIF output feature outputs the used primitives of a complex block, the routing inside a complex block, and the routing outside a complex block in BLIF format. This feature supports outputting LUTs, flip-flops, and circuit I/Os. It outputs all interconnect edges as buffers in BLIF thus allowing it to output the used interconnect within and outside a complex block.

Although useful, the BLIF outputter feature is not yet fully featured. For example, it does not output the BLIF subcircuits and hence cannot check memories. We plan to add more support for it in future work.

4.5 Software Organization

Toolchain organization is an important part of software engineering so we provide an overview of the AAPack toolchain here. The AAPack source files are based off T-VPack and merged into the VPR 5.0 code base. This allows packing, placement, and routing to share the same data structures.

We also created a library that reads and processes the architecture description file (specified in UTFAL). The new VPR code links to that library to obtain architecture information. Other tools, such as the previously discussed ODIN II [21], can also link to that library to obtain information about the FPGA architecture. These changes are

a significant improvement to the previous backend VPR toolchain where updates to the FPGA architecture file required third-party tools to create a new parser to read that architecture file.

4.6 Summary

In this chapter, we described a new packing tool called AAPack. The AAPack algorithm packs blocks of a technology-mapped netlist into a complex block. This process entails the selection of candidate blocks, the placement of those candidates within the physical block hierarchy of a complex block, and the routing of the nets packed within the complex block. After describing the general algorithm, we presented the special handling of memories. Afterwards, we covered software engineering issues. For example, AAPack includes a BLIF output feature for correctness checking.

AAPack performs greedy, area-driven packing which places limits on its exploration. We plan to enable hill-climbing capabilities and timing-driven features in future work.

The next section investigates the quality of results for the AAPack algorithm and presents a preliminary architecture exploration.

Chapter 5

Experiments and Results

In this chapter, we describe a set of experiments and results whose purpose is two-fold: 1) to validate the proposed FPGA architecture modeling language (UTFAL) and demonstrate its utility in modeling a variety of complex block architectures, and 2) to evaluate the quality of the proposed packing algorithm, AAPack. We begin by outlining our experimental methodology. We then move on to describe a set of experiments that exercise both features of UTFAL, as well as our packing algorithm. We first model and pack circuits into simple LUT-based logic blocks, thus permitting a direct comparison with prior work on FPGA packing. We then model and pack circuits into more sophisticated architectures that until now, could not be handled by any publicly-available FPGA architectural exploration tools.

5.1 Experimental Methodology

We take an empirical approach to evaluating the proposed FPGA architecture description language and packing algorithm. Specifically, we first develop a model (in UTFAL) of the target FPGA architecture that we intend to map circuits into. We then use a set of CAD tools to map a suite of benchmark circuits into the target architecture. The result is a complete implementation of each benchmark circuit in the target architecture. We

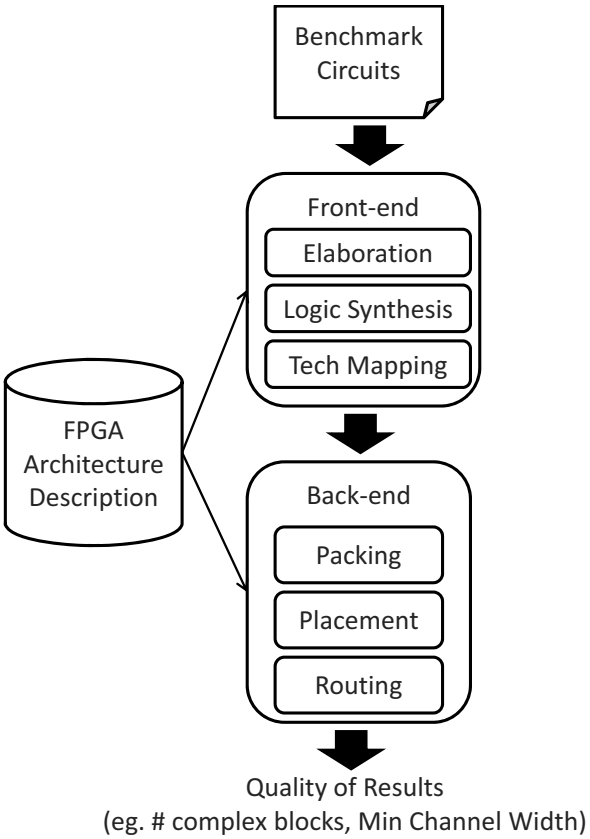

Figure 5.1: Experimental method to measure quality of an architecture/CAD algorithm

evaluate the quality of the implementation using a number of metrics (to be described below). The overall experimental methodology is illustrated in Figure 5.1; we describe the different pieces of the methodology below.

We employ two suites of benchmark circuits in this research. The first suite consists of the 20 largest MCNC [47] circuits that are commonly used in FPGA architecture and CAD research. These circuits contain only LUTs, flip-flops and I/Os. These are small circuits and so a caveat must be placed on interpretting results based on the MCNC benchmarks for larger benchmarks. This work is a step towards enabling experiments with large, modern benchmarks in academia but full support for them is not yet ready in the full CAD flow and so we use the MCNC in our experiments. The second benchmark suite consists of a single benchmark, called the OpenRISC 1200 (or1200) [25]. The or1200 circuit contains two dual-port memories, one 32x32 multiplier, LUTs and flip-flops. We

Table 5.1: Routing architecture parameters

Parameter	Value
Fc_in	0.15
Fc_out	0.125
L	4

include this benchmark as it allows us to demonstrate the capability of our language and packer to handle circuits with large hard-IP-like complex blocks, like those present in today's commercial FPGAs.

We limit our investigation to square FPGA architectures having an equal number of rows and columns of complex blocks. I/O tiles surround the two-dimensional array of complex blocks, forming a ring at the chip periphery. An I/O tile is present at the end of each row/column of core logic, and each I/O tile contains 7 I/O pads. The FPGA contains a programmable routing network that allows complex blocks be connected with one another and allows complex blocks to connect to/from I/Os. The architecture of the programmable routing network is held constant across all experiments conducted, which permits us to evaluate the impact of our packer changes alone.

Detailed parameters of the routing architecture we use are shown in Table 5.1. Each pin on a complex block is associated with a multi-track routing channel adjacent to the block. *Fc_in* specifies the fraction of neighbouring routing tracks that can connect to an input pin of a complex block. *Fc_out* specifies the fraction of routing tracks that an output pin of a complex block can connect to. *L* specifies length of the metal routing wire segments. As shown, we target an architecture with wire segments whose length spans 4 complex block tiles. We use the *single driver* routing wire model, as is used in modern commercial FPGAs. In the single driver model, each wire segment is unidirectional and a driver switch is present at only one of the segment's two ends.

As illustrated in Figure 5.1, the FPGA CAD flow includes a front-end (comprising HDL elaboration, logic synthesis and technology mapping) and a back-end (comprising

packing, placement and routing). Front-end synthesis accepts a design written in a hardware description language (HDL) and produces technology-mapped BLIF files that are then passed to the back-end flow. For the MCNC benchmarks, HDL elaboration was unnecessary, as the circuits were already available in a form that permitted them to be fed directly into logic synthesis. ABC version 70930 [44] was used for logic synthesis and technology mapping. ABC was set to perform technology mapping with structural choices turned on [35][1]. WireMap was used for technology mapping [22] and its parameters were set to minimize the number of LUTs in mapped solutions (area-directed mapping). WireMap produces LUTs that use fewer inputs than competing mappers and this allows us to meaningfully measure fracturability for LUTs. The HDL elaboration was only needed for the or1200 circuit. ODIN II [21] was used to elaborate or1200 into BLIF format then ABC was used for logic synthesis and technology mapping. ABC treats memories and multipliers as black-boxes and performs no logic optimization on these blocks nor does it change the synthesis of the surrounding logic based on the contents of the black box.

In the back-end of the flow, the packer we use depends on the experiment being run: we use our own packer for some experiments, and we use a previously-published packer for other experiments. For placement and routing, we use VPR 5.0, and we set VPR to operate in non-timing driven mode. All experiments were run on a 3 GHz Intel Xeon 5160 proceessor with 4 MB cache and 8 GB of memory. Only one processor core was used.

Four quality metrics are used to assess packing quality. The first metric is packer runtime. We expect that AAPack will have longer runtimes versus prior work (such as T-VPack 5.0 [33]) because AAPack is designed to support a much wider range of complex block architectures. The second metric is the number of complex blocks in the packed

[1]Structural choices gives the technology mapper a more comprehensive view of the solution space, leading to better results.

circuit. This measurement determines the minimum dimensions of the FPGA and is a proxy for total chip area. The third metric is the minimum channel width (min W) needed to successfully route the circuit, which is determined by repeatedly invoking the router with differing number of tracks, in order to find the fewest needed. This is a reasonable metric to measure the overall demand of the circuit on the routing architecture. Channel width is also a major factor in determining the total area of an FPGA. The last metric is the number of external nets (nets that connect two or more complex blocks) after packing. This metric is also a proxy for routing demand.

We note that it would have been better to evaluate our packing algorithm *and* architectures using the silicon area needed to implement each benchmark circuit. Silicon area depends strongly on the sizes of transistors used in the FPGA's logic and routing. Transistor sizing is typically done with specific speed performance goals in mind. As our current flow is non-timing driven, a realistic measure of silicon area is difficult to ascertain. We therefore leave silicon area evaluation as a topic for future work.

5.2 Results

5.2.1 Comparison of Algorithms on a Simple LUT-Based Complex Block

We begin by comparing AAPack against another Packing algorithm on a fairly simple complex block (as shown in Figure 2.2). The complex block we target has 8 LUT/FF pairs ($N = 8$), 27 input pins ($I = 27$), and has LUTs with 6 inputs ($K = 6$). The results of the experiment appear in Table 5.2. The meaning of the table columns is as follows: The first column gives the name of the benchmark circuit. The second column, Min W, is the minimum channel width needed to successfully route the circuit. The next

Table 5.2: AAPack results for a basic GPCB archictecture with N = 8, K = 6, I = 27.

Circuit	Min W	Pack T	Place T	Route T	Total T	Num Ext Nets	Num CB
alu4	54	1.2	1.75	12.62	15.57	410	103
apex2	64	1.41	3.37	17.5	22.28	601	125
apex4	66	1.2	1.85	13.53	16.58	485	103
bigkey	34	1.37	3.99	10.56	15.92	436	115
clma	64	4.4	15.81	94.7	114.91	1732	367
des	38	0.85	5.12	1.91	7.88	576	88
diffeq	34	1.49	1.98	2.48	5.95	502	117
dsip	34	1.65	5.47	12.33	19.45	600	115
elliptic	36	4.53	6.92	38.75	50.2	1073	293
ex1010	88	4.31	8.41	210.77	223.49	1227	324
ex5p	50	0.94	1.54	6.91	9.39	384	82
frisc	48	4.42	7.48	19.6	31.5	972	288
misex3	56	1.1	1.73	5.92	8.75	419	97
pdc	80	4.05	9.15	131.3	144.5	1217	313
s298	44	0.95	1.03	6.4	8.38	303	82
s38417	28	7.64	10.93	23.5	42.07	1545	454
s38584.1	38	6.75	14.77	10.29	31.81	1956	426
seq	64	1.33	3.14	13.2	17.67	542	115
spla	70	3.08	5.98	67.6	76.66	954	247
tseng	26	1.4	2.32	2.93	6.65	512	128
geoeman	47.90	2.09	4.20	15.46	24.01	706.17	166.25
stdev	17.74	2.06	4.36	53.45	56.53	489.43	125.53

three columns, labeled Pack T, Place T, and Route T[2], respectively, give the runtime (in seconds) of the packer, placer, and router steps. The next column, labeled Total T, gives the total runtime for the entire back-end CAD flow. The two right-most columns present the number of external nets for each circuit and the number of complex blocks in the packed solution, respectively. At the bottom of each column, we provide the geometric mean and standard of deviation for each metric.

It is difficult to glean any picture of AAPack quality by looking at the data in Table 5.2 in isolation. We therefore compare the results produced by using AAPack with those produced by a previous published timing-driven packer, T-VPack 5.0 [33]. T-VPack

[2]This represents the time VPR needs to find the minimum W for a circuit, which involves routing the circuit multiple times, each with a different value for W. Commercial FPGAs have fixed W and therefore need only be routed once. Consequently, the apparent dominance of router runtime in Table 5.2 can be misleading if read outside of this context.

Table 5.3: AAPack vs T-VPack 5.0. Values are presented as AAPack/T-VPack.

Circuit	Min W	Pack T	Place T	Route T	Total T	Num Ext Nets	Num CB
alu4	1.00	120.00	1.03	0.71	0.80	0.81	1.00
apex2	0.89	141.00	1.00	0.79	0.87	0.86	0.99
apex4	0.94	120.00	1.01	0.71	0.79	0.82	1.00
bigkey	1.00	137.00	0.97	0.99	1.07	0.95	1.00
clma	0.94	110.00	1.14	0.86	0.93	0.87	1.00
des	1.12	85.00	1.27	0.15	0.48	0.95	1.00
diffeq	1.00	149.00	1.06	0.70	1.09	0.86	1.05
dsip	1.00	82.50	1.25	0.69	0.87	1.15	1.00
elliptic	0.86	90.60	1.05	2.20	2.07	0.84	1.10
ex1010	0.88	71.83	0.94	0.55	0.57	0.80	0.99
ex5p	0.89	94.00	1.05	1.03	1.14	0.82	1.00
frisc	0.77	88.40	0.87	0.53	0.69	0.74	1.02
misex3	0.97	55.00	1.01	0.85	1.01	0.81	1.00
pdc	0.93	67.50	1.03	1.12	1.15	0.82	0.99
s298	0.88	47.50	1.04	0.98	1.11	0.83	1.00
s38417	0.70	152.80	0.97	1.50	1.56	0.73	1.01
s38584	0.86	112.50	0.99	0.61	1.00	0.86	1.01
seq	0.94	133.00	0.98	1.02	1.09	0.84	1.00
spla	0.95	77.00	0.97	1.66	1.64	0.82	1.00
tseng	0.81	70.00	1.01	0.92	1.21	0.88	1.01
geomean	0.91	95.22	1.03	0.83	1.00	0.85	1.01
stdev	0.09	31.82	0.09	0.45	0.37	0.09	0.02

is a commonly used baseline in research on FPGA packing, and it produces packed circuits with less area than its predecessor, area-driven VPack [4]. Table 5.3 shows results comparing AAPack with T-VPack. The meaning of the columns in Table 5.3 is the same as in Table 5.2; however, the data values in the table are ratios of the data for AAPack divided by the corresponding data for T-VPack. As such, numbers less than 1 in the table are indicative of superior AAPack results; numbers larger than 1 are indicative of superior T-VPack results.

Looking first at the results for channel width (Min W column) and the number of external nets (Num Ext Nets column), we see that AAPack reduces minimum channel width by 9% and the number of external nets by 15%, respectively. We suspected that this improvement in routability by AAPack was primarily caused by the cost function

used to select candidate blocks described in section 4.3.4. This cost function has a term *net_absorption_gain* which gives higher priority to candidates with nets that get completely absorbed into the complex block thus reducing the number of external nets. The cost function in T-VPack, on the otherhand, did not show a preference towards these nets. When we modified T-VPack to use the same cost function as AAPack, the minimum channel width and the number of external nets of the modified T-VPack and AAPack were within 1%. We observe that AAPack improves routability, without increasing the number of complex blocks (Num CB column), which lies within 1% of T-VPack, on average.

Turning to the runtime results, we observe that AAPack is 95 times slower than T-VPack (two orders of magnitude). This time is primarily spent performing full detailed routing every time a netlist block is packed into the complex block. Placer time is largely unaffected by using AAPack versus T-VPack, while router time is reduced with AAPack. The smaller router time is a consequence of AAPack reducing the number of external nets between complex blocks, leading to fewer pins to route and less work for the router. We see that the increase in packer time and the decrease in router time balance each other – the total back-end CAD flow time is equal, on average, for AAPack versus T-VPack for a minimum channel width experiment.

The balanced total back-end runtime for the end-user will differ from the above results because the end-user will have an FPGA with a fixed channel width and hence will only run routing once. To measure the runtime impact of the AAPack algorithm on the total time seen by an end-user, we first fixed the channel width of each circuit by 20% over its minimum channel width (so that the circuit has some difficulty routing but is not at the limit of routability [33]). We then re-ran the previous experiment with the routing stage set to one iteration. The results are presented in Table 5.4.

The average router time of AAPack is less than that of T-VPack by 18%, this is similar to the minimum channel width experiment. The total back-end CAD flow time

Table 5.4: AAPack vs T-VPack 5.0 single route normalized runtimes.

Circuit	Single Route T	Total T
alu4	0.77	1.28
apex2	0.98	1.23
apex4	1	1.3
bigkey	0.79	1.18
clma	0.68	0.99
des	1.03	1.24
diffeq	0.73	1.62
dsip	1	1.42
elliptic	0.68	1.3
ex1010	0.56	0.67
ex5p	1.07	1.38
frisc	0.34	0.77
misex3	1.47	1.57
pdc	0.68	0.83
s298	0.96	1.55
s38417	0.66	1.3
s38584	0.98	1.32
seq	0.94	1.25
spla	0.71	0.94
tseng	1.1	1.5
geomean	0.82	1.2
stdev	0.25	0.27

of the AAPack flow is 20% greater than the T-VPack flow since routing is no longer as dominant in the total back-end runtime.

The prinicipal aim of this work is to enable architecture exploration. Runtime is not the main objective of our work and so we are willing to accept a high runtime for AAPack compared to T-VPack. There are a few different methods we can use to improve the runtime of the AAPack algorithm for future work and they are as follows: First, the algorithm performs a runtime intensive detailed route of a complex block every time a candidate is packed into a complex block. We can reduce runtime by packing multiple candidates into a complex block before performing a detailed route. Second, the algorithm re-routes all nets in a complex block which may not always be necessary. We can reduce runtime by selectively determining which nets need to be re-routed.

5.2.2 Fracturable LUT Architectures

We now demonstrate the utility of UTFAL and AAPack by using them to model, and pack circuits into, complex blocks that contain *fracturable* LUTs. Recall that a fracturable LUT is a LUT that can be broken (fractured) into two smaller LUTs that share input pins (see Figure 2.4). Modern commercial FPGAs incorporate fracturable LUTs for the purpose of improving logic density – many LUTs in circuits are small and can therefore be paired together and implemented in a single fracturable LUT.

Figure 5.2 depicts a complex block based on fracturable LUTs. The block has the same structure as a basic LUT-based complex block, however, in this case each BLE has two outputs. A fracturable BLE contains a fracturable (dual-output) LUT and two bypassable registers – one register for each LUT output. Figure 5.3 shows a fracturable BLE with 7 inputs and bypassable registers. A fracturable LUT has two modes of operation: 1) as a single K-input LUT, or 2) as two LUTs that together use at most FI inputs. In the dual-LUT mode, parameter FI determines the amount of pin sharing that is required between the pair of LUTs that are implemented in the fracturable LUT.

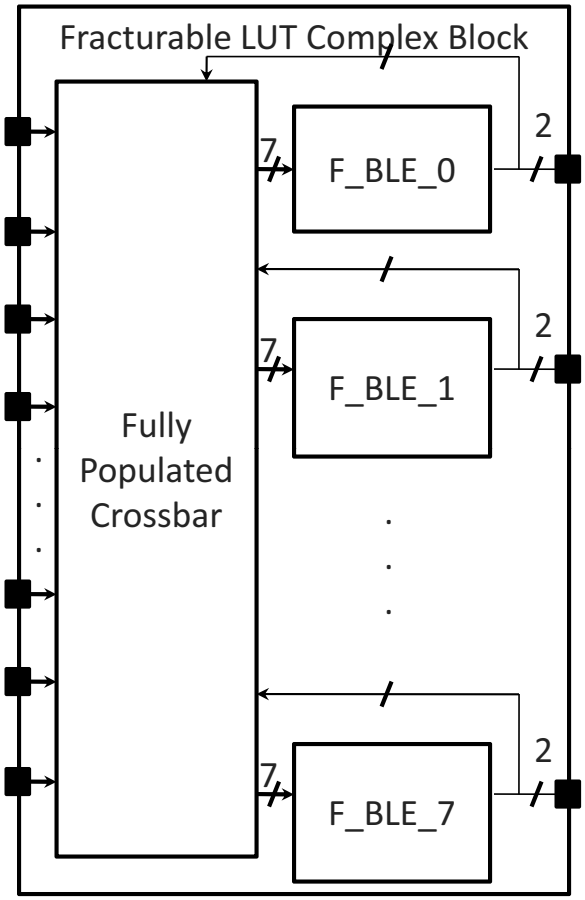

Figure 5.2: Complex block for a fracturable BLE FPGA architecture

Figure 5.4 shows an example of a fracturable LUT. This fracturable LUT can operate as either one 6-LUT ($K = 6$) or two 5-LUTs that share 3 inputs ($FI = 7$).

A key architectural question for fracturable LUT architectures concerns the "right" value for FI. Larger values for FI will permit more packing flexibility likely at a higher area cost, whereas lower values of FI will reduce packing flexibility. We use UTFAL and AAPack to investigate the impact of different FI values in fracturable LUT architectures. For this experiment, we assume that $K = 6$ (LUTs have 6-inputs when used in single-output mode) and $N = 8$ (there are 8 fracturable BLEs per complex block). We sweep FI from 5 to 10, covering all pin sharing possibilities for $K = 6$. The meaning of $K = 6$ and $FI = 5$ deserves some elaboration. In this case, when the LUT is used in dual-output mode, the two LUTs implemented therein can together use no more than 5 distinct input

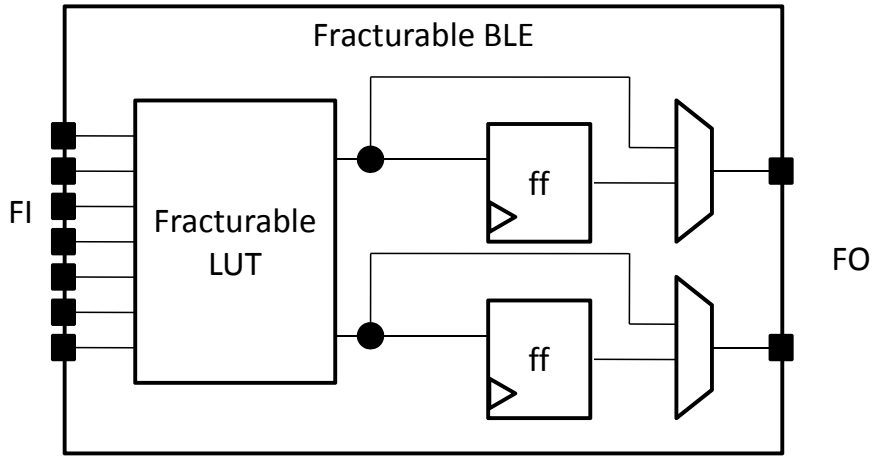

Figure 5.3: A fracturable BLE with 7 inputs, 2 outputs, and optional output registers

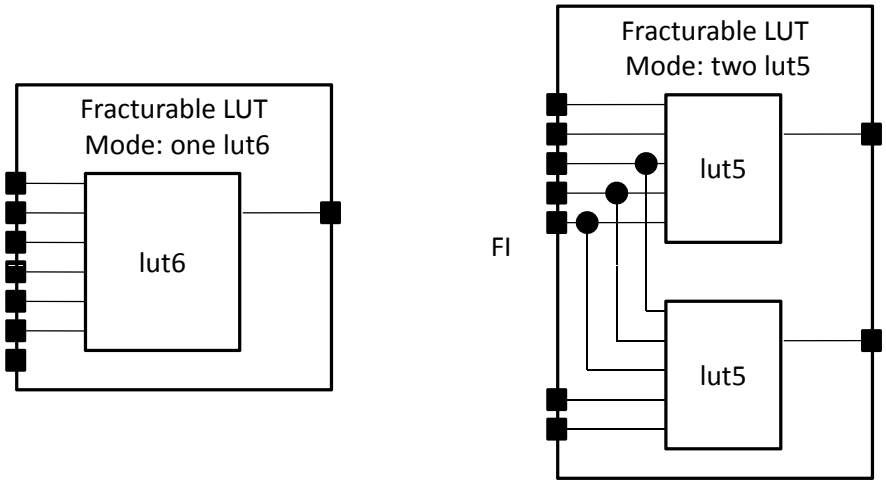

Figure 5.4: The structure of fracturable 6-LUT with 7 inputs.

signals. We set the number of external complex block inputs, I, equal to $FI \times N$, which implies that no pin sharing requirements are imposed *between* BLEs within the complex block.

Comparison Against Lower-Bound

We first evaluate the logic density offered by various fracturable LUT architectures by comparing the number of complex blocks in packing solutions, with a lower bound on the optimal number of complex blocks needed. The lower bound is computed as follows:

```
# CB = ceiling((# 5-LUTs or smaller / 16) + (# 6-LUTs / 8))
```

The "# 5-LUTs or smaller" is the number of LUTs in the input (to the packer) netlist that use 5 or few inputs. The #6-LUTs is the number of LUTs that use *exactly* 6 inputs. The bound was developed through a counting argument: There are 8 fracturable BLEs in a complex block. Each fracturable BLE can implement one 6-LUT, so the number of 6-LUTs in a design increases the complex block count by 1/8. Each fracturable BLE can alternatively implement (*at most*) two 5-LUTs, so each LUT in a benchmark circuit that uses 5 or fewer inputs increases the complex block count by 1/16. We ignore flip-flops in the lower bound. Ignoring flip-flops does not affect the correctness of the bound but it does affect how tight the lower bound is to the optimal solution. Almost all flip-flips in the MCNC benchmark circuits can be paired with a LUT in the BLE, so ignoring flip-flops has a negligible impact on the tightness of the bound. Having defined a lower bound, we measure logic utilization for fracturable LUT-based complex blocks as follows:

```
# CB AAPack / # CB lower bound
```

The logic utilization results attained by AAPack is shown in Figure 5.5. The X-axis is FI, the number of inputs to the BLE. The Y-axis gives the logic utilization as defined above. Each point is the geometric average logic of the 20 MCNC benchmarks. The bars

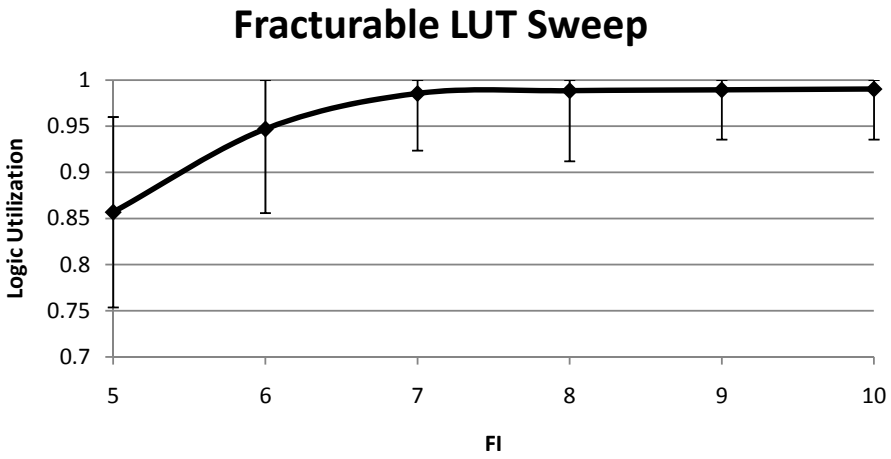

Figure 5.5: Logic utilization of AAPack on a sweep of FI.

show the minimum and maximum value at each point. Observe that AAPack is able to pack to within 2% of the lower bound of the optimal solution when FI is 7 or greater.

We analyzed why logic utilization in Figure 5.5 peaked at 99% (relative to the lower bound). A reason for this is that AAPack is uses heuristics and can make poor packing choices under certain conditions. Consider the example in Figure 5.6, and assume that the highlighted $lut1$ is the seed of a new fracturable BLE in a complex block. Figure 5.7 shows how AAPack packs the blocks associated with $lut1$ until the fracturable BLE is full (the circles in the figure correspond to BLEs in the packing solution). AAPack takes a greedy approach and both flip-flops get packed into a single BLE. The other LUT, $lut6$, cannot pack into the BLE and must take up an entirely new BLE. A third BLE is then needed for $lut2$. In contrast, Figure 5.8 shows an optimal packing with just 2 BLEs.

Comparison of Architectures with and without Fracturable LUTs

An architecture with fracturable LUTs uses the variation of LUT sizes in user circuits to save area. Two small LUTs in a user circuit might map into one fracturable LUT whereas for non-fracturable LUT architectures, two LUTs is always needed. If this situation happens often enough, and if one fracturable LUT is smaller than two non-fracturable LUTs, then this may result in an area savings. This experiments investigates the impact

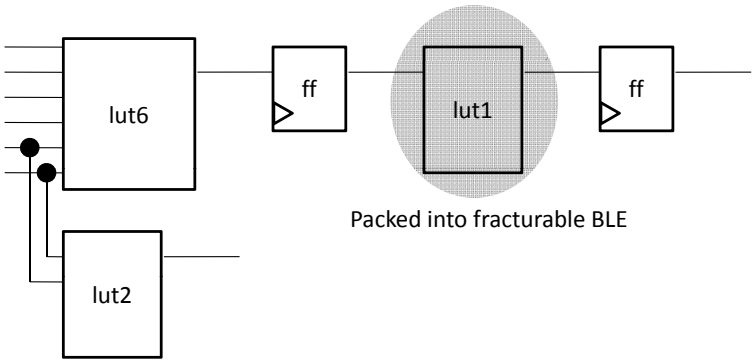

Figure 5.6: The highlighted $lut1$ is packed into an empty fracturable BLE.

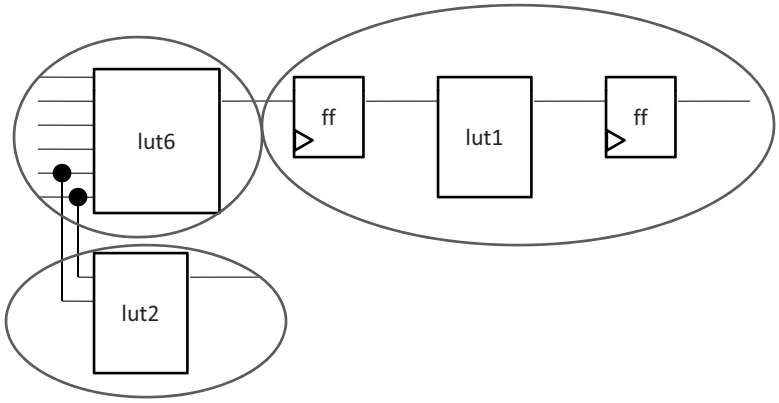

Figure 5.7: AAPack packs the neighbour primitives of $lut1$.

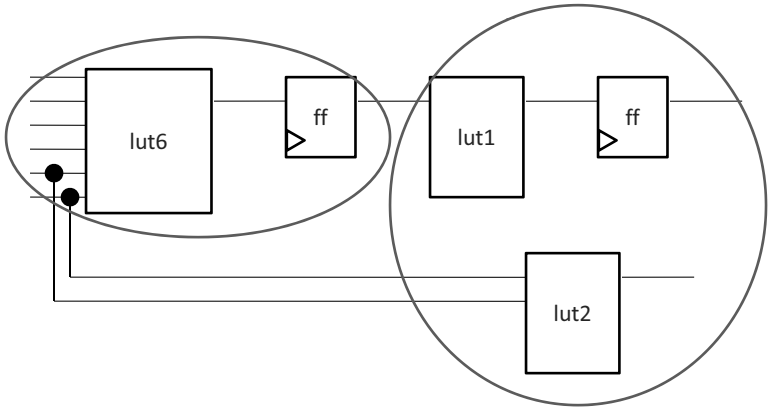

Figure 5.8: A more area efficient packing selects different candidates.

Table 5.5: Fracturable LUT Architectures vs Non-fracturable.

FI	Min W	Pack T	Place T	Route T	Total T	Num Ext Nets	Num CB
5	1.35	4.19	0.95	0.97	1.61	1.11	0.72
6	1.46	6.88	0.91	0.84	1.98	1.12	0.66
7	1.45	13.26	0.92	0.97	3.23	1.13	0.63
8	1.43	22.70	0.90	0.85	5.16	1.12	0.63
9	1.38	23.11	0.91	0.95	5.39	1.09	0.63
10	1.37	16.49	0.90	0.96	4.53	1.09	0.63
geomean	1.41	12.21	0.91	0.92	2.57	1.11	0.65
stdev	0.05	7.89	0.02	0.06	1.86	0.02	0.04

on different area metrics by architectures with and without fracturable LUTs.

Table 5.5 gives results comparing fracturable and non-fracturable LUT architectures. The comparative baseline is a basic (non-fracturable) LUT-based complex block with $K = 6$, $N = 8$, and $I = 48$. We selected $I = 48$ to remove the requirement of pin-sharing between BLEs for the (non-fracturable) LUT-based architecture because we removed pin-sharing at between BLEs for fracturable LUT-based architectures. With the exception of the FI column, each data point in the table is a ratio of two geometric mean values: the first mean corresponding to a fracturable LUT-based architecture, the second mean corresponding to the non-fracturable baseline. The meanings of the columns is as follows: FI represents the number of inputs used by the fracturable LUT when it is operating in dual-LUT mode. Min W is the minimum channel width needed to route the benchmark circuits. The next four columns give the pack, place, route and total runtime. This is followed by the number of external nets. The last column is the number of complex blocks in the packing solutions. The last two rows show the geometric mean and the standard of deviation of the fracturable LUT-based complex block architectures.

Looking first at the Min W column of Table 5.5, we see that routing demand from the complex blocks increases dramatically with fracturable architectures compared to a non-fracturable architecture. The minimum channel width averages a 41% increase, due to the greater packing density (and hence higher routing demand from each block) afforded by fracturable LUT-based blocks.

A second observation is that fracturable BLEs provide a 28% to 37% reduction in the number of complex blocks needed to implement circuits (versus using a basic non-fracturable LUT-based complex block) (see the Num CB column). Among fracturable LUT architectures, varying $FI = 5$ to $FI = 6$ yields the highest incremental reduction in the number of complex blocks. Increasing FI beyond six offers little additional benefit. Xilinx Virtex-5 FPGAs have complex blocks that are similar to the $FI = 5$ scenario; Altera Stratix IV FPGAs have complex blocks that are similar to the $FI = 8$ scenario. A further architectural study with timing and silicon area values is needed to conclusively show which value of FI is best given our benchmarks.

A third observation pertains to runtime. Packing runtime is increased 4-23× when fracturable LUTs are used versus non-fracturable LUTs. AAPack incorporates a router that is called to check packing feasibility. The complexity of the routing problem (during packing) increases with FI, leading to longer runtimes. An exception to this is when $FI = 10$. When $FI = 10$, LUTs packed into the same fracturable LUT do not need to share any inputs, permitting a more straightforward checking of packing feasibility.

We also use fracturable LUT architectures to assess the quality of the AAPack router (which is used for packing feasibility checking). The AAPacker router is based on negotiated congestion routing [34], which is a heuristic approach, and therefore, may erroneously reject certain packing solutions that are in fact legal. To gauge this possibility, the same fracturable LUT sweep was done on a *relaxed* version of AAPack that ignores the routing step inside the packer – packing feasibility checks are based solely on signal counts. Packings produced through this approach may be infeasible and hence, this produces an upper bound on packing density. Table 5.6 shows the results of this comparison. The relaxed version of AAPack packs slightly more tightly and produces, on average, 1% better utilization. The increase in utilization comes at a cost of 4% more external nets and a 3% increase in the minimum channel width. Such changes are relatively small, and we conclude that the router's heuristic nature has little impact on packing quality.

Table 5.6: Relaxed AAPack vs AAPack

FI	W	num nets	num clb
5	1.01	1.01	0.98
6	1.01	1.03	0.99
7	1.01	1.03	0.99
8	1.03	1.04	0.99
9	1.07	1.07	0.99
10	1.06	1.07	0.99
geomean	1.03	1.04	0.99
stdev	0.03	0.02	0.00

5.2.3 Depopulated Crossbar

A simple LUT-based complex block, such as the one shown in Figure 2.2, contains a crossbar with inputs from the complex block and from the BLE outputs. The outputs of this crossbar feed the BLE inputs. This crossbar is fully populated, meaning that any input of the crossbar can connect to any output of the crossbar. A *depopulated* crossbar attempts to save area removing some of the input-to-output connections. The reduction in crossbar area will lead to a reduction in packing flexibility that may have a negative area impact on the rest of the architecture, for example, it may reduce the utilization of the complex blocks. Prior work has shown a net area reduction for certain depopulated crossbars [26] and commercial FPGAs use depopulated crossbars [27]. A packer should therefore be able to model and pack to complex blocks containing depopulated crossbars. In this section, we use UTFAL to model depopulated crossbars and then use AAPack to pack circuits into complex blocks containing them.

We evaluate using a depopulated crossbar in a simple LUT-based complex block with the following parameters: $K = 6$, $N = 8$, $I = 27$. The crossbar used in the experiment is characterized through four parameters: the number of inputs into the crossbar from the complex block inputs (xCI), the number of inputs into the crossbar from the BLE outputs ($xBLI$), the number of outputs of the crossbar (xO), and the fraction of crossbar inputs that a single crossbar output can connect to (xP). For example, a *fully* populated

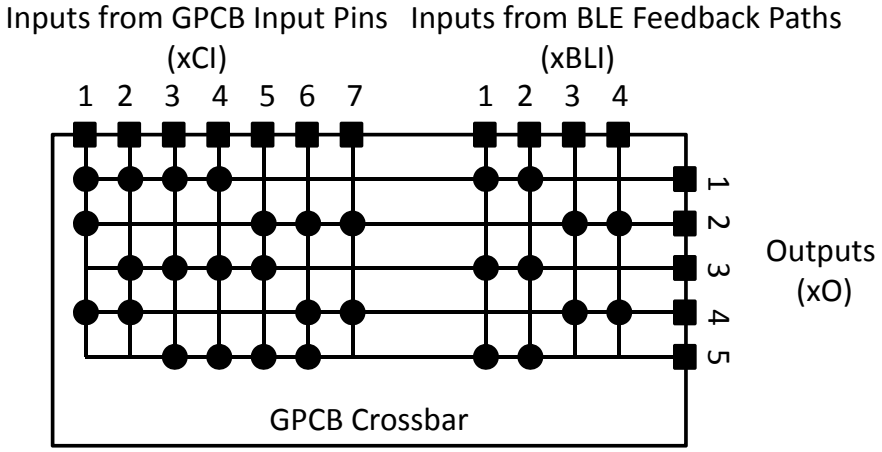

Figure 5.9: Depopulated crossbar with a staggered connection pattern and a xP of 0.5

crossbar for the complex block used in this experiment has $xCI = 27$, $xBLI = 8$, $xO = 48$, and $xP = 1$. By varying xP in the range of 0 to 1, we can model a range of depopulated crossbars. Aside from xP, the other three crossbar parameters are held constant in this experiment.

The depopulated crossbars considered in this experiment follow a staggered connection pattern. An example of the pattern for a small crossbar with $xCI = 7$, $xBLI = 4$, $xO = 6$, and $xP = 0.5$ is shown in Figure 5.9. A solid circle in the diagram represents a programmable connection from an input to an output. The staggered connection pattern connects inputs from xCI and $xBLI$ separately. Only the pattern generation for xCI will be described, because $xBLI$ uses the same pattern generation. Each output connects to $\lceil xCI \times xP \rceil$ inputs ($\lceil 7 \times 0.5 \rceil = \lceil 3.5 \rceil = 4$). Connections to inputs are made consecutively; the next output begins making connections where the previous output stopped. For example, output pin 1 connects to the first 4 input pins from xCI, and output pin 2 connects to input pins 5 to 7, and then wraps around and connects to pin 1 of xCI.

We measure the impact of xP on logic utilization, runtime, and the number of external nets. We acknowledge that min W should also be measured, but this was not done owing to limitations with the tool. In [26], Lemieux showed that an accurate min W

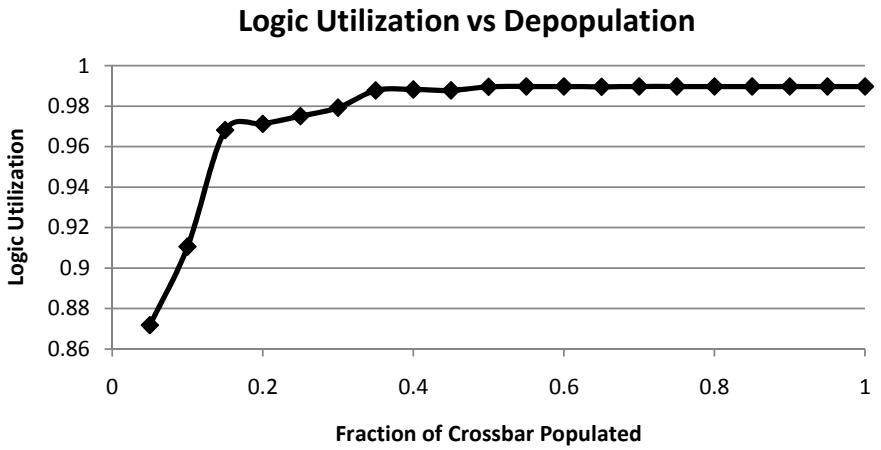

Figure 5.10: Logic utilization vs depopulation of crossbar

measurement requires that the routing stage of the CAD flow is able to access the routing inside of a complex block. This feature is not part of VPR 5.0 so we do not present results for min W.

Figure 5.10 graphs the impact of depopulation on logic utilization (logic utilization is the ratio of the minimum number of complex blocks necessary with the number used as described in 5.2.2). The x-axis shows depopulation while the y-axis shows logic utilization. AAPack is able to maintain high logic utilization, near 99% when the crossbar population is higher than 30%. Logic utilization degrades severely below the 15% crossbar population threshold. Observe that utilization remains above 85% even when the crossbar is only 5% populated. At 5% population, each BLE input can connect to two complex block input pins and one feedback pin. It is surprising that such high logic utilization can be maintained despite the very inflexible crossbar.

Figure 5.11 graphs the impact of depopulation on the number of external nets when compared to a complex block with a complete crossbar. The x-axis shows the population of the crossbar; the y-axis shows the average relative increase in the number of external nets. The packer starts to have difficulty packing related blocks together when the crossbar population (xP) is lower than 0.15, thus increasing the number of external nets.

Figure 5.12 shows the impact of depopulation on runtime using the full crossbar as

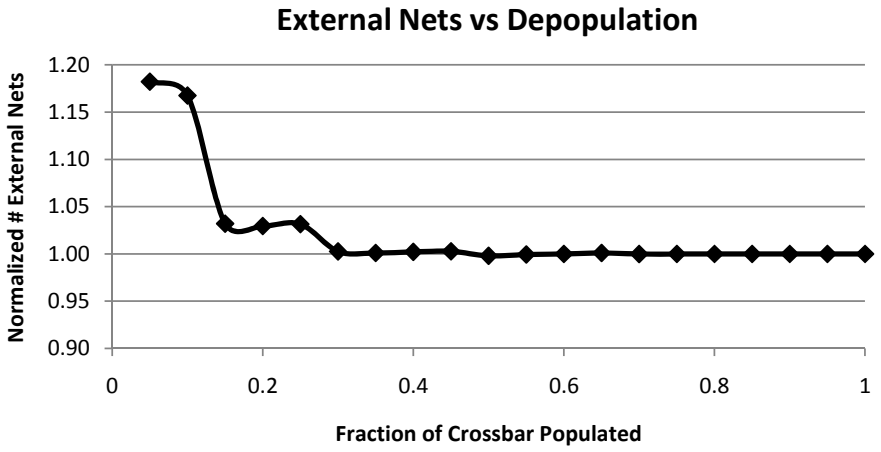

Figure 5.11: Number of external nets vs depopulation of crossbar

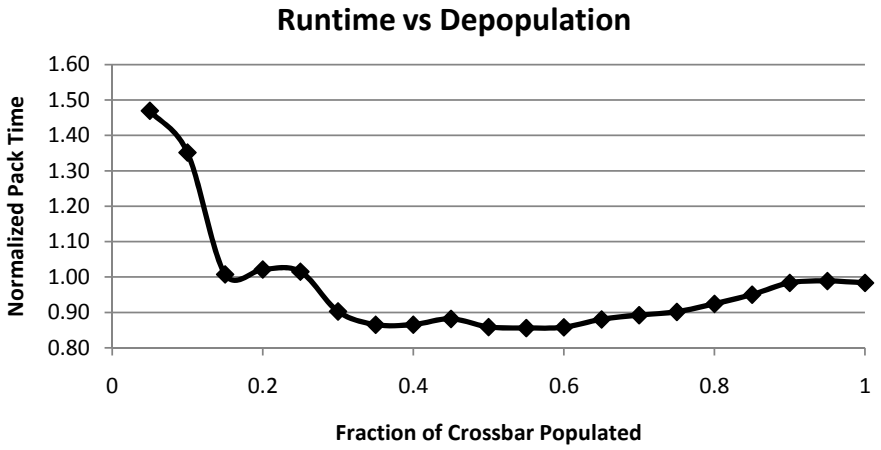

Figure 5.12: Runtime vs depopulation of crossbar

the baseline. The x-axis shows population (xP); the y-axis shows runtime. There are two factors at play in the runtime results. When xP is low, the crossbar is sparse, and AAPack's internal router must spend several iterations to find a legal routing. On the other hand, when xP is high, the number of programmable connections within the crossbar is large, again increasing the runtime of AAPack's internal router. The graph shows that there is a large range of xP values, 0.4 to 0.7, where runtime is reduced by more than 10% compared to the full crossbar. Runtime is approximately the same as the full crossbar for an xP value of 0.3.

5.2.4 Heterogeneous FPGA

The final experiment is a proof-of-concept that AAPack can be used to pack heterogeneous circuits. We use AAPack to pack the or1200 benchmark circuit (described in section 5.1 to an architecture that contains fracturable 36x36 multipliers, single-port configurable memories, and dual-port configurable memories. The dual-port memories can be configured as 2048x2 or to 1024x4. This architecture also includes traditional LUT-based complex blocks with the following parameters: $K = 4$, $N = 10$, $I = 22$.

The circuit contains one 32x32 multiplier and should therefore use one physical fracturable multiplier. The circuit also contains two 32-bit dual-port RAMs. Each dual-port RAM is split into single-bit wide RAMs by ODIN II; hence, the technology mapped circuit contains 64 dual-port RAMs, each of which is 1-bit wide. These RAMs are sized 32x1 so four of such single-bit wide RAMs should pack into one 1024x4 RAM. Finally, the circuit also contains 3423 LUTs and 579 flip-flops.

For the or1200 circuit, AAPack generates a packing with 349 LUT-based complex blocks (98% logic utilization), 16 dual-port memory complex blocks, and 1 multiplier complex block. The minimum channel width is 48 and the number of external nets is 2149 (2323 nets were absorbed into the LUT-based complex blocks). The placement of the blocks on the FPGA is shown in Figure 5.13 and a close-up of the routing near a dual-port memory is shown in Figure 5.14. In the placement, the multipliers have a height of 3; dual-port memories have a height of 2. The lightly shaded 1x1 squares are single-port memories; the white squares are *unused* LUT-based complex blocks and the grey squares are *used* LUT-based complex blocks. Lightly shaded blocks are unused, and darker blocks are used. These graphics are generated by the VPR 5.0 graphics engine.

In conclusion, AAPack is capable of correctly packing a heterogeneous circuit (that contains multipliers and memories) into a heterogeneous FPGA architecture while maintaining a high logic utilization.

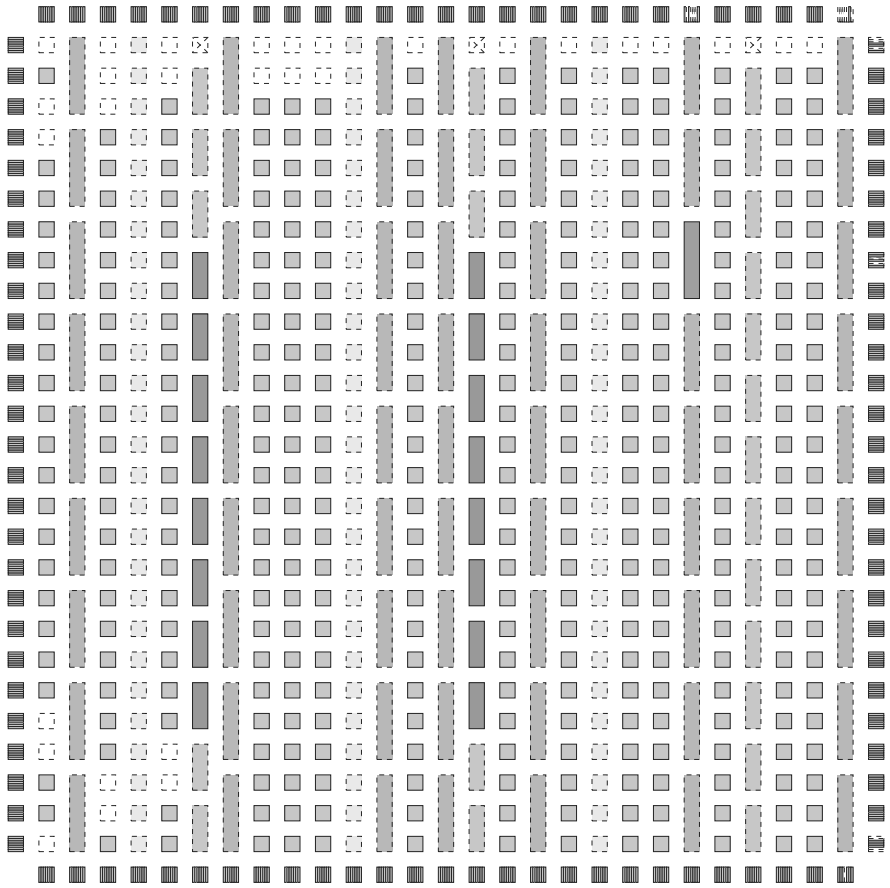

Figure 5.13: Placement of the or1200 circuit on a heterogeneous FPGA

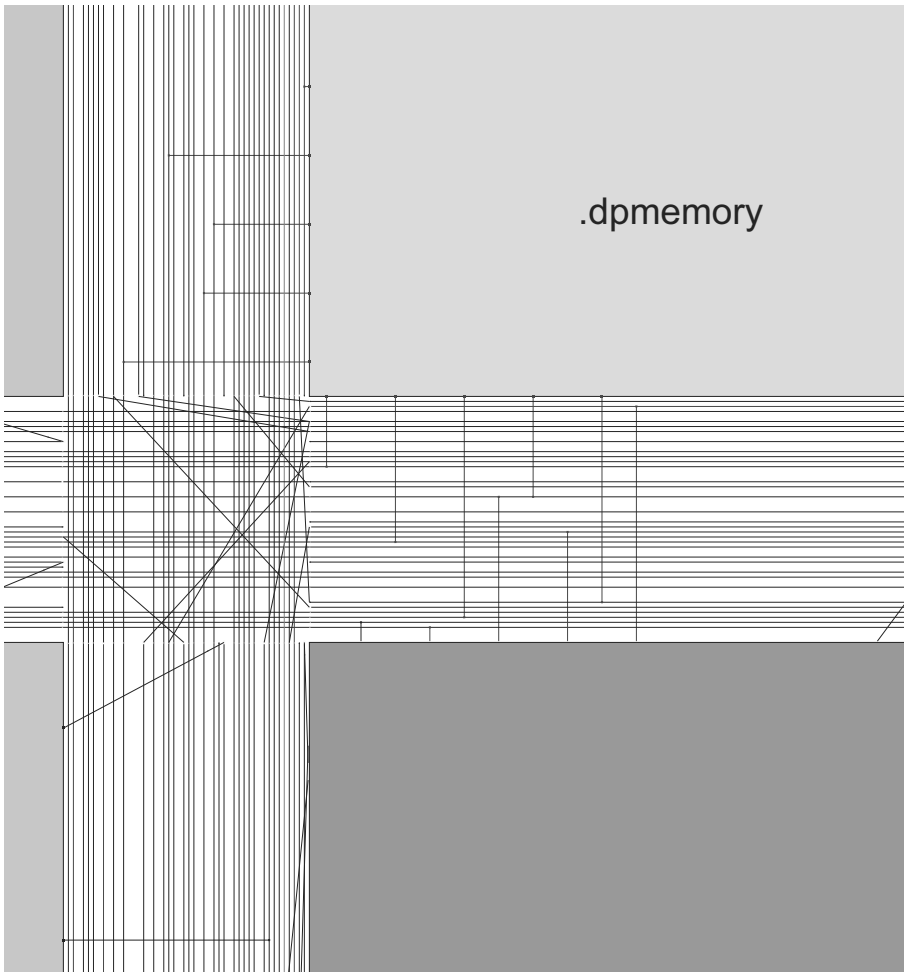

Figure 5.14: Part of the routing for the or1200 circuit on a heterogeneous FPGA

5.3 Summary

This chapter investigated the quality of results of the AAPack tool followed by an exploration of different architectures using AAPack. We described an experimental methodology that consisted of running benchmark circuits through a CAD flow and measuring the results from that flow. We first showed that AAPack has comparable quality of results with a previously published tool. Across the 20 largest MCNC benchmarks, AAPack produces circuits with 9% lower minimum channel width and 1% higher complex block count versus T-VPack. We then showed that AAPack can be used for preliminary architectural experiments. For fracturable LUT-based complex blocks, we showed that 99% complex block utilization can be achieved if the 6-LUTs in a complex block can be fractured into two 5-LUTs that share 3 or fewer inputs. We also showed the impact of using fracturable LUTs on routing channel width. For depopulated crossbars within complex blocks, we showed that 85% complex block utilization can be achieved with just 5% switch population. Finally, we showed that AAPack can be used to pack to an architecture with fracturable multipliers and configurable memories.

Chapter 6

Conclusions

FPGAs are a widely used media for implementing digital circuits. The complex block architecture of an FPGA has a significant impact on the cost, performance, and power of circuits implemented on the FPGA. Despite this, innovation in complex block architecture has been held back by a lack of publicly-available architecture exploration infrastructure. This thesis takes a first step towards enabling the comprehensive exploration of modern complex block architectures. In particular, we make three contributions:

1. We presented a new modeling language, UTFAL, that can concisely describe a large space of complex block architectures. We illustrated the capabilities of UTFAL for complex block modeling using three examples: 1) a LUT-based complex block, 2) a complex block with configurable memories, and 3) a complex block with fracturable multipliers. The examples demonstrate that UTFAL can model complex blocks that are representative of those in modern commercial FPGAs.

2. We presented a new area-driven FPGA packing tool, AAPack, that, to our knowledge, is the first publicly-available "generic" packer, capable of packing circuits into a range of complex block architectures. AAPack accepts two inputs: 1) the circuit to be packed, and 2) a description (in UTFAL) of the target FPGA's complex block architecture. Prior FPGA packing tools hard-code the architecture model

and implement a packing algorithm suitable for a small range of complex block architectures. AAPack, on the other hand, receives the architecture model as input data, and can pack into a much wider range of architectures. Specifically, we showed that AAPack provides improved quality-of-result versus T-VPack when targeting basic LUT-based complex block architectures. Across the 20 largest MCNC benchmarks, AAPack produces circuits with 9% lower minimum channel width and 1% higher complex block count versus T-VPack. We also showed that AAPack can be used to explore general-purpose complex block architectures that contain fracturable LUTs and depopulated crossbars (see the next item below). Lastly, as a proof-of-concept, we showed that AAPack can be used to pack to a heterogeneous FPGA with fracturable multipliers and configurable memories.

3. We conducted a preliminary architecture investigation of fracturable LUT-based complex block architectures and complex blocks with partially populated intra-block routing crossbars. For fracturable LUT-based complex blocks, we showed that 99% complex block utilization can be achieved if the 6-LUTs in a complex block can be fractured into two 5-LUTs that share 3 or fewer inputs. We also showed the impact of using fracturable LUTs on routing channel width. For depopulated crossbars within complex blocks, we showed that 85% complex block utilization can be achieved with just 5% switch population.

6.1 Future Work

In the future, we plan to improve the FPGA architecture exploration capabilities of UTFAL and AAPack.

Timing-Driven CAD Flow

Timing-driven packing can have a significant impact on the performance of a circuit implemented on an FPGA [33] and thus, timing-driven packing is a fundamental part of a timing-driven CAD flow. This motivates the need to extend AAPack so that it can use delay information to make better packing decisions, and thereby permit architectural exploration based on circuit speed performance. We envision that UTFAL can be extended to incorporate delay information in the architecture model, specifically, delay values for the interconnect and block constructs. AAPack's packing algorithm will then be extended to make use of the delay information to produce speed-optimized packing solutions.

Architecture Exploration

UTFAL and AAPack are intended to enable FPGA complex block architectural investigation. While the proposed tools are already able to measure certain trade-offs between complex block architectures (such as routing channel width and utilization), a more thorough architecture study requires an accurate picture of silicon area, speed and power. Further work is needed to extend the proposed tools (as well as the front-end synthesis and back-end layout tools) to produce a complete FPGA architecture evaluation system – a system from which one can draw accurate architectural conclusions.

Heterogeneous architecture exploration is another interesting future research direction with many open questions: What are the "best" ratios and types of special-purpose complex blocks in an FPGA? What functions should the different special-purpose blocks in an FPGA implement? What fixed-functionality computations should be incorporated into general-purpose complex blocks? To answer these questions and others, AAPack may need to support additional features, an example being support for bus-based routing to explore the types of floating point-based complex blocks proposed in [16]. Another feature in a future version of AAPack may be a smarter way to decide which complex

block type to use when there are multiple valid choices available.

6.1.1 Improve Quality and Runtime for the AAPack Algorithm

The AAPack algorithm uses greedy heuristics which may result in a poor quality of results on some architectures. We plan to add some hill-climbing capabilities such that AAPack can explore solutions past locally optimal points.

The runtime of the CAD flow affects engineering productivity and also affects the number of different architectures an architect can explore given finite time and compute resources. The runtime of AAPack is two orders of magnitude greater than the basic GPCB packer T-VPack and there are at least two approaches to mitigate the runtime. First, the current packing algorithm performs a runtime intensive detailed route of a complex block every time a candidate primitive is packed into the complex block. We can reduce runtime by packing multiple candidates into the complex block before performing a detailed route. Second, the current algorithm re-routes all nets in a complex block each time a new primitive is packed. We can reduce this runtime using an incremental routing strategy.

6.1.2 Carry Chain Support

Carry chains are fundamental to implementing arithmetic circuits in commercial FPGAs [12] [17]. A carry chain is a set of multiplexers, connected serially, that realizes the propagate/generate concepts in binary addition. Since all commercial FPGAs contain carry chains, architecture evaluation tools should be able to model and use them (including support throughout the flow from HDL synthesis to routing). In UTFAL, it may be necessary to define a special new primitive to represent a carry chain. Or, an alternative approach would be to describe carry chains using the generic constructs that are already part of UTFAL. The latter approach may require that the synthesis and packing CAD tools "infer" carry chains from the circuit's HDL and/or Boolean network representation.

Carry chains in today's commercial FPGAs are closely tied to LUTs: each multiplexer in a carry chain is linked to a specific neighbouring LUT. Packing of LUTs and carry chains must therefore occur in tandem, and as such, handling carry chains will undoubtedly require that changes be made to AAPack's packing algorithm.

Bibliography

[1] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 12(3):288–298, March 2004.

[2] Taneem Ahmed, Paul D. Kundarewich, Jason H. Anderson, Brad L. Taylor, and Rajat Aggarwal. Architecture-specific packing for Virtex-5 FPGAs. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 5–13, New York, NY, USA, 2008. ACM.

[3] UC Berkeley. Berkeley logic interchange format. Technical report, Technical report, University of California at Berkeley, 1998.

[4] Vaughn Betz and Jonathan Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. *IEEE 1997 CUSTOM INTEGRATED CIRCUITS CONFERENCE*, 1997.

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

[6] E. Bozorgzadeh, S. Ogrenci 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.

[7] S.D. Brown. *Fundamentals of digital logic with Verilog design*. Tata McGraw-Hill, 2007.

[8] K. Chaudhary. FPGA having logic element carry chains capable of generating wide XOR functions, March 30 1999. US Patent 5,889,411.

[9] Doris T. Chen, Kristofer Vorwerk, and Andrew Kennings. Improving Timing-driven FPGA Packing with Physical Information. *International Conference on Field Programmable Logic and Applications*, pages 117–123, Aug 2007.

[10] Altera Corporation. Altera Fact Sheet. `http://www.altera.com/corporate/news_room/factshe` 2009.

[11] Altera Corporation. Stratix IV Device Family Overview. 2009.

[12] Altera Corporation. Stratix Series FPGA Fracturable Look-Up Table Logic Structure. `http://www.altera.com/products/devices/stratix-fpgas/about/alm-logic-structu` 2009.

[13] C. Ebeling, D. Cronquist, and P. Franklin. RaPiDReconfigurable pipelined datapath. *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 126–135, 1996.

[14] S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips, and A. Sharma. Totem: domain-specific reconfigurable logic. *IEEE Transactions on VLSI Systems*, pages 1–25, 2006.

[15] S. Hauck, MM Hosler, and TW Fry. High-performance carry chains for FPGA's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):138–147, 2000.

[16] Chun Hok Ho, Chi Wai Yu, Phillip Leong, Wayne Luk, and Steven J. E. Wilton. Floating-point FPGA: architecture and modeling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(12):1709–1718, 2009.

[17] Xilinx Incorporated. Virtex-6 Family Overview. 2009.

[18] Xilinx Incorporated. Xilinx Corporate Fact Sheet. `http://www.altera.com/corporate/news_room/factsheet/nr-factsheet.html`, 2009.

[19] P. Jamieson and J. Rose. A verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field Programmable Logic and Applications*, volume 2005, pages 305–310, 2005.

[20] P. Jamieson and J. Rose. Mapping multiplexers onto hard multipliers in FPGAs. *Proc. of IEEE-NECAS*, pages 323–326, 2005.

[21] Peter Jamieson, Kenneth Kent, and Jonathan Rose. Odin II. `http://www.users.muohio.edu/jamiespa/odin_II.html`, 2009.

[22] S. Jang, B. Chan, K. Chung, and A. Mishchenko. WireMap: FPGA technology mapping for improved routability. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 47–55. ACM, 2008.

[23] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. *IEEE Transactions on Very Large Scale Inegration (VLSI) Systems*, 7(1):69–79, Mar 1999.

[24] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[25] D. Lampret. OpenRISC 1200 IP Core Specification. *September June*, 2001.

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

[27] 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 *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20. ACM New York, NY, USA, 2003.

[28] J.Y. Lin, D. Chen, and J. Cong. Optimal simultaneous mapping and clustering for FPGA delay optimization. In *Proceedings of the 43rd annual Design Automation Conference*, page 477. ACM, 2006.

[29] A.C. Ling, J. Zhu, and S.D. Brown. Scalable Synthesis and Clustering Techniques Using Decision Diagrams. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 27(3):423, 2008.

[30] 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 *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 133–142, New York, NY, USA, 2009. ACM.

[31] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. VPR and T-Vpack User's Manual (Version 5.0). `http://www.eecg.utoronto.ca/vpr/VPR_5.pdf`, 2009.

[32] Jason Luu, Keith Redmond, William Lo, Paul Chow, Lothar Lilge, and Jonathan Rose. Fpga-based monte carlo computation of light absorption for photodynamic cancer therapy. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:157–164, 2009.

[33] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. *Proceedings of the Seventh International Symposium on Field Programmable Gate Arrays*, pages 37–46, 1999.

[34] L. McMurchie and C. Ebeling. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117. ACM, 1995.

[35] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for LUT-based FPGAs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 41–49, New York, NY, USA, 2006. ACM.

[36] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *System-on-chip: next generation electronics*, page 187, 2006.

[37] T. Ngai, J. Rose, and S.J.E. Wilton. An sram-programmable field-configurable memory. pages 499–502, May 1995.

[38] Gang Ni, Jiarong Tong, and Jinmei Lai. A new fpga packing algorithm based on the modeling method for logic block. In *ASIC, 2005. ASICON 2005. 6th International Conference On*, volume 2, pages 877–880, Oct. 2005.

[39] J. Oliveira Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel. CGADL: An Architecture Description Language for Coarse-Grained Reconfigurable Arrays. *IEEE Transactions on VLSI Systems*, 17(9):1247–1259, 2009.

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

[41] Baber A Pervez, Alastair Smith, and George Constantinides. Fourth Year Design Project, 2009.

[42] II Quartus. Version 9.0 Handbook. *San Jose, Ca: Altera. Available at¡http://www.altera.com/¿. Accessed in*, 2009.

[43] Amit Singh, Ganapathy Parthasarathy, and Malgorzata Marek-Sadowksa. Efficient Circult Clustering for Area and Power Reduction in FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 7(4):643–663, Nov 2002.

[44] B.L. Synthesis. Verification Group. *ABC: A system for sequential synthesis and verification*, 2005.

[45] W3C. Extensible Markup Language (XML). `http://www.w3.org/XML/`, 2003.

[46] Steve J.E. Wilton. Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 19(1):56–68, January 2000.

[47] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. *MCNC, Jan*, 1991.

Appendix A

FPGA_CBS Examples

A.1 Complete Specification of Basic GPCB Block

```
<!--
Example of a Basic GPCB with
N = 10, K = 4, I = 22, O = 10
BLEs consisting of a single LUT followed by a flip-flop that can be
    bypassed
-->

<pb_type name="clb">
  <input name="I" num_pins="22"/>
  <output name="O" num_pins="10"/>
  <clock name="clk"/>

  <pb_type name="ble" num_pb="10">
    <input name="in" num_pins="4"/>
    <output name="out" num_pins="1"/>
    <clock name="clk"/>

    <pb_type name="lut_4" blif_model=".names" num_pb="1" class="lut">
      <input name="in" num_pins="4" port_class="lut_in"/>
      <output name="out" num_pins="1" port_class="lut_out"/>
    </pb_type>
    <pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>

    <interconnect>
      <direct input="lut_4.out" output="ff.D"/>
```

```
      <direct input="ble.in" output="lut_4.in"/>
      <mux input="ff.Q lut_4.out" output="ble.out"/>
      <direct input="ble.clk" output="ff.clk"/>
    </interconnect>
  </pb_type>

  <interconnect>
    <complete input="{clb.I ble[9:0].out}" output="ble[9:0].in"/>
    <complete input="clb.clk" output="ble[9:0].clk"/>
    <direct input="ble[9:0].out" output="clb.O"/>
  </interconnect>
</pb_type>
```

A.2 Complete Specification of Memory with Reconfigurable Aspect Ratio

```
<!--
Example of a memory with reconfigurable aspect ratios and optional,
bus-based registers at its inputs and outputs.
The three aspect ratios that it can be configured to are 2048x1,
   1024x2, 512x4
-->

<pb_type name="block_RAM">
  <input name="addr" num_pins="11"/>
  <input name="din" num_pins="4"/>
  <input name="wen" num_pins="1"/>
  <output name="dout" num_pins="4"/>
  <clock name="clk"/>

  <!-- 2 sets of bypassable registers -->
  <pb_type name="ff_reg_in" blif_model=".latch" num_pb="16"
     class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>

  <pb_type name="ff_reg_out" blif_model=".latch" num_pb="4"
     class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>

  <pb_type name="mem_reconfig" num_pb="1">
    <input name="addr" num_pins="11"/>
    <input name="din" num_pins="4"/>
    <input name="wen" num_pins="1"/>
```

```
<output name="dout" num_pins="4"/>

<!-- Declare a 2048x1 memory type -->
<mode name="mem_2048x1_mode">
  <pb_type name="mem_2048x1" blif_model=".subckt sp_mem"
     class="memory">
    <input name="addr" num_pins="11" port_class="address"/>
    <input name="din" num_pins="1" port_class="data_in"/>
    <input name="wen" num_pins="1" port_class="write_en"/>
    <output name="dout" num_pins="1" port_class="data_out"/>
  </pb_type>
  <interconnect>
    <direct input="mem_reconfig.addr[10:0]"
       output="mem_2048x1.addr"/>
    <direct input="mem_reconfig.din[0]" output="mem_2048x1.din"/>
    <direct input="mem_reconfig.wen" output="mem_2048x1.wen"/>
    <direct input="mem_2048x1.dout" output="mem_reconfig.dout[0]"/>
  </interconnect>
</mode>

<!-- Declare a 1024x2 memory type -->
<mode name="mem_1024x2_mode">
  <pb_type name="mem_1024x2" blif_model=".subckt sp_mem"
     class="memory">
    <input name="addr" num_pins="10" port_class="address"/>
    <input name="din" num_pins="2" port_class="data_in"/>
    <input name="wen" num_pins="1" port_class="write_en"/>
    <output name="dout" num_pins="2" port_class="data_out"/>
  </pb_type>
  <interconnect>
    <direct input="mem_reconfig.addr[9:0]"
       output="mem_1024x2.addr"/>
    <direct input="mem_reconfig.din[1:0]" output="mem_1024x2.din"/>
    <direct input="mem_reconfig.wen" output="mem_1024x2.wen"/>
    <direct input="mem_1024x2.dout"
       output="mem_reconfig.dout[1:0]"/>
  </interconnect>
</mode>

<!-- Declare a 512x4 memory type -->
<mode name="mem_512x4_mode">
  <pb_type name="mem_512x4" blif_model=".subckt sp_mem"
     class="memory">
    <input name="addr" num_pins="9" port_class="address"/>
    <input name="din" num_pins="4" port_class="data_in"/>
    <input name="wen" num_pins="1" port_class="write_en"/>
    <output name="dout" num_pins="4" port_class="data_out"/>
  </pb_type>
  <interconnect>
    <direct input="mem_reconfig.addr[9:0]"
       output="mem_512x4.addr"/>
    <direct input="mem_reconfig.din[3:0]" output="mem_512x4.din"/>
    <direct input="mem_reconfig.wen" output="mem_512x4.wen"/>
    <direct input="mem_2048x1.dout"
```

```
            output="mem_reconfig.dout[3:0]"/>
      </interconnect>
    </mode>
  </pb_type>

  <!-- connect up the memory, mux the block RAM outputs so that the
     memory configurations are mutually exclusive -->
  <interconnect>
    <direct input="{block_RAM.wen block_RAM.din block_RAM.addr}"
       output="ff_reg_in[15:0].D"/>

    <mux input="{block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]}
        ff_reg_in[15:0].Q"
         output="{mem_reconfig.wen mem_reconfig.din
            mem_reconfig.addr}"/>

    <direct input="mem_reconfig.dout" output="ff_reg_out[3:0].D"/>
    <mux input="mem_reconfig.dout ff_reg_out[3:0].Q"
       output="block_RAM.dout"/>

    <complete input="block_RAM.clk" output="ff_reg_in[15:0].clk"/>
    <complete input="block_RAM.clk" output="ff_reg_out[3:0].clk"/>
  </interconnect>

</pb_type>
```

A.3 Fracturable Multiplier

A.3.1 Multiplier SPCB Example

The third complex example is a fracturable multiplier SPCB shown in Figure A.1. The large *block_mult* can implement one 36x36 multiplier cluster called a *mult_36x36_slice* or it can implement two divisble 18x18 multipliers. A divisible 18x18 multiplier can implement a 18x18 multiplier cluster called a *mult_18x18_slice* or it can be fractured into two 9x9 mulitplier clusters called *mult_9x9_slice*.

Figure A.2 shows a multiplier slice. Pins belonging to the same input or output port of a multiplier slice must be either all registered or none registered. Pins belonging to different ports or different slices may have different register configurations. A multiplier primitive itself has two input ports (A and B) and one output port (OUT).

First, we describe the *block_mult* complex block as follows:

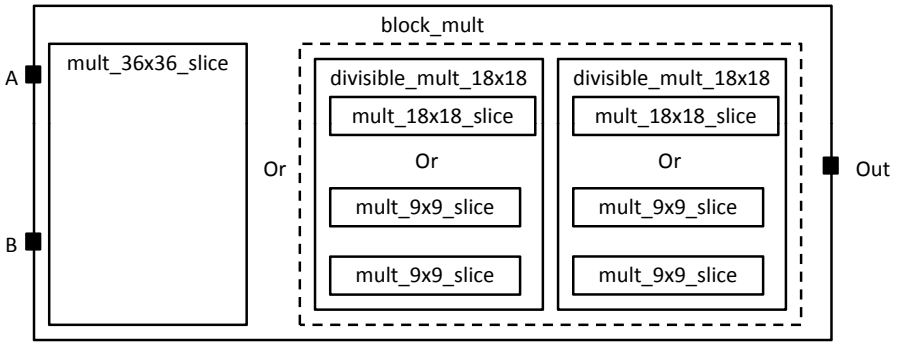

Figure A.1: Reconfigurable Embedded Multiplier.

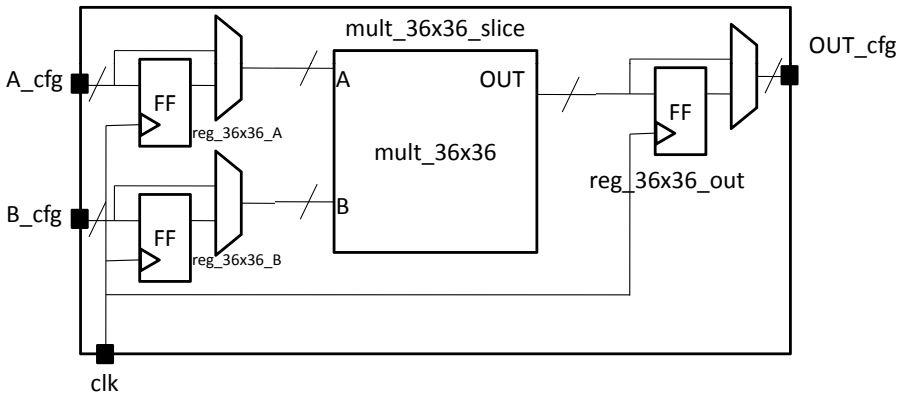

Figure A.2: 36x36 reconfigurable embedded multiplier slice.

```
<pb_type name="block_mult">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
  <clock name="clk"/>
```

The *block_mult* complex block has two modes: a mode containing a 36x36 multiplier slice and a mode containing two fracturable 18x18 multipliers. The mode containing the 36x36 multiplier slice is described first. The mode and slice is declared here:

```
  <mode name="mult_36x36">
    <pb_type name="mult_36x36_slice" num_pb="1">
      <input name="A_cfg" num_pins="36"/>
      <input name="B_cfg" num_pins="36"/>
      <input name="OUT_cfg" num_pins="72"/>
      <clock name="clk"/>
```

This is followed by a description of the primitives within the slice. There are two sets of 36 flip-flops for the input ports and one set of 64 flip-flops for the output port.

There is one 36x36 multiplier primitive. These primitives are described by four *pb_type*s as follows:

```
<pb_type name="reg_36x36_A" blif_model=".latch" num_pb="36"
   class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
<pb_type name="reg_36x36_B" blif_model=".latch" num_pb="36"
   class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
<pb_type name="reg_36x36_out" blif_model=".latch" num_pb="72"
   class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>

<pb_type name="mult_36x36" blif_model=".subckt mult" num_pb="1">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
</pb_type>
```

The slice description finishes with a specification of the interconnection. Using the same technique as in the memory example, bus-based multiplexers are used to register the ports. Clocks are connected using the complete tag because there is a one-to-many relationship. Direct tags are used to make simple, one-to-one connections.

```
<interconnect>
  <direct input="mult_36x36_slice.A_cfg"
     output="reg_36x36_A[35:0].D"/>
  <direct input="mult_36x36_slice.B_cfg"
     output="reg_36x36_B[35:0].D"/>
  <mux input="mult_36x36_slice.A_cfg reg_36x36_A[35:0].Q"
     output="mult_36x36.A"/>
  <mux input="mult_36x36_slice.B_cfg reg_36x36_B[35:0].Q"
     output="mult_36x36.B"/>

  <direct input="mult_36x36.OUT" output="reg_36x36_out[71:0].D"/>
  <mux input="mult_36x36.OUT reg_36x36_out[71:0].Q"
     output="mult_36x36_slice.OUT_cfg"/>

  <complete input="mult_36x36_slice.clk"
```

```
          output="reg_36x36_A[35:0].clk"/>
        <complete input="mult_36x36_slice.clk"
          output="reg_36x36_B[35:0].clk"/>
        <complete input="mult_36x36_slice.clk"
          output="reg_36x36_out[71:0].clk"/>
      </interconnect>
    </pb_type>
```

The mode finishes with a specification of the interconnect between the slice and its parent.

```
    <interconnect>
      <direct input="block_mult.A" output="mult_36x36_slice.A_cfg"/>
      <direct input="block_mult.B" output="mult_36x36_slice.A_cfg"/>
      <direct input="mult_36x36_slice.OUT_cfg"
        output="block_mult.OUT"/>
      <direct input="block_mult.clk" output="mult_36x36_slice.clk"/>
    </interconnect>
  </mode>
```

After the mode containing the 36x36 multiplier slice is described, the mode containing two fracturable 18x18 multipliers is described:

```
<mode name="two_divisible_mult_18x18">
  <pb_type name="divisible_mult_18x18" num_pb="2">
    <input name="A" num_pins="18"/>
    <input name="B" num_pins="18"/>
    <input name="OUT" num_pins="36"/>
    <clock name="clk"/>
```

This mode has two additional modes which are the actual 18x18 multiply block or two 9x9 mulitplier blocks. Both follow a similar description as the mult_36x36_slice with just the number of pins halved so the details are not repeated.

```
    <mode  name="two_divisible_mult_18x18">
      <pb_type name="mult_18x18_slice" num_pb="1">
        <!-- follows previous pattern for slice definition -->
      </pb_type>
      <interconnect>
        <!-- follows previous pattern for slice definition -->
      </interconnect>
    </mode>

    <mode name="two_mult_9x9">
      <pb_type name="mult_9x9_slice" num_pb="2">
        <!-- follows previous pattern for slice definition -->
```

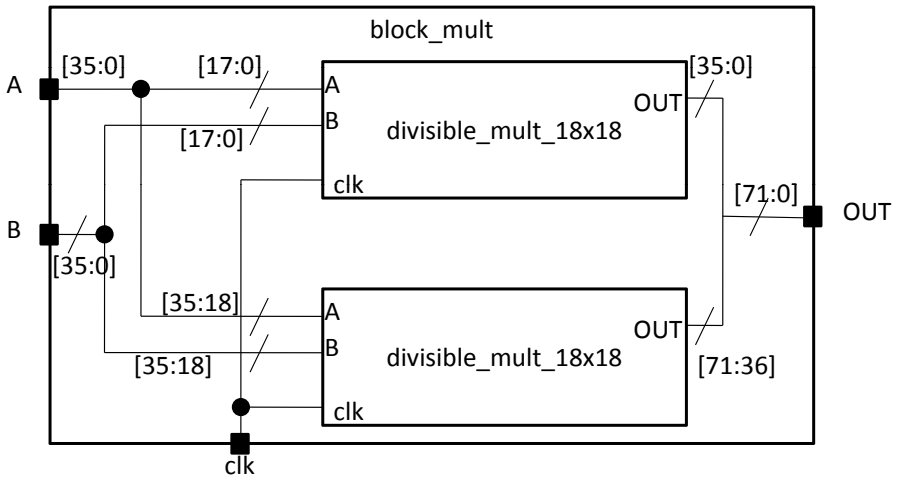

Figure A.3: Interconnect between complex block and two divisible 18x18 multipliers.

```
      </pb_type>
      <interconnect>
        <!-- follows previous pattern for slice definition -->
      </interconnect>
    </mode>

  </pb_type>
```

The interconnect for the divisible 18x18 mode is shown in Figure A.3. The unique characteristic of this interconnect is that the input and output ports of the parent is split in half, one half for each child. A convenient way to specify this is to use the syntax *divisible_mult_18x18[1:0]* which will append the pins of the ports of the children together. The interconnect for the fracturable 18x18 mode is described here:

```
    <interconnect>
      <direct input="block_mult.A"
         output="divisible_mult_18x18[1:0].A"/>
      <direct input="block_mult.B"
         output="divisible_mult_18x18[1:0].B"/>
      <direct input="divisible_mult_18x18[1:0].OUT"
         output="block_mult.OUT"/>
      <complete input="block_mult.clk"
         output="divisible_mult_18x18[1:0].clk"/>
    </interconnect>
  </mode>
</pb_type>
```

A.3.2 Complete Specification

```
<!-- Example of a fracturable mutliplier whose inputs and outputs may
    be optionally registered
The multiplier hard logic block can implement one 36x36, two 18x18, or
    four 9x9 multiplies
-->
<pb_type name="block_mult">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
  <clock name="clk"/>

  <mode name="mult_36x36">
    <pb_type name="mult_36x36_slice" num_pb="1">
      <input name="A_cfg" num_pins="36"/>
      <input name="B_cfg" num_pins="36"/>
      <input name="OUT_cfg" num_pins="72"/>
      <clock name="clk"/>

      <pb_type name="reg_36x36_A" blif_model=".latch" num_pb="36"
         class="flipflop">
        <input name="D" num_pins="1" port_class="D"/>
        <output name="Q" num_pins="1" port_class="Q"/>
        <clock name="clk" port_class="clock"/>
      </pb_type>
      <pb_type name="reg_36x36_B" blif_model=".latch" num_pb="36"
         class="flipflop">
        <input name="D" num_pins="1" port_class="D"/>
        <output name="Q" num_pins="1" port_class="Q"/>
        <clock name="clk" port_class="clock"/>
      </pb_type>
      <pb_type name="reg_36x36_out" blif_model=".latch" num_pb="72"
         class="flipflop">
        <input name="D" num_pins="1" port_class="D"/>
        <output name="Q" num_pins="1" port_class="Q"/>
        <clock name="clk" port_class="clock"/>
      </pb_type>

      <pb_type name="mult_36x36" blif_model=".subckt mult" num_pb="1">
        <input name="A" num_pins="36"/>
        <input name="B" num_pins="36"/>
        <output name="OUT" num_pins="72"/>
      </pb_type>

      <interconnect>
        <direct input="mult_36x36_slice.A_cfg"
           output="reg_36x36_A[35:0].D"/>
        <direct input="mult_36x36_slice.B_cfg"
           output="reg_36x36_B[35:0].D"/>
        <mux input="mult_36x36_slice.A_cfg reg_36x36_A[35:0].Q"
           output="mult_36x36.A"/>
```

```
        <mux input="mult_36x36_slice.B_cfg reg_36x36_B[35:0].Q"
           output="mult_36x36.B"/>

        <direct input="mult_36x36.OUT" output="reg_36x36_out[71:0].D"/>
        <mux input="mult_36x36.OUT reg_36x36_out[71:0].Q"
           output="mult_36x36_slice.OUT_cfg"/>

        <complete input="mult_36x36_slice.clk"
           output="reg_36x36_A[35:0].clk"/>
        <complete input="mult_36x36_slice.clk"
           output="reg_36x36_B[35:0].clk"/>
        <complete input="mult_36x36_slice.clk"
           output="reg_36x36_out[71:0].clk"/>
      </interconnect>
    </pb_type>
    <interconnect>
      <direct input="block_mult.A" output="mult_36x36_slice.A_cfg"/>
      <direct input="block_mult.B" output="mult_36x36_slice.A_cfg"/>
      <direct input="mult_36x36_slice.OUT_cfg"
         output="block_mult.OUT"/>
      <direct input="block_mult.clk" output="mult_36x36_slice.clk"/>
    </interconnect>
  </mode>

  <mode  name="two_divisible_mult_18x18">
    <pb_type name="divisible_mult_18x18" num_pb="2">
      <input name="A" num_pins="18"/>
      <input name="B" num_pins="18"/>
      <input name="OUT" num_pins="36"/>
      <clock name="clk"/>

      <mode name="mult_18x18">
        <pb_type name="mult_18x18_slice" num_pb="1">
          <input name="A_cfg" num_pins="18"/>
          <input name="B_cfg" num_pins="18"/>
          <input name="OUT_cfg" num_pins="36"/>
          <clock name="clk"/>

          <pb_type name="reg_18x18_A" blif_model=".latch" num_pb="18"
             class="flipflop">
            <input name="D" num_pins="1" port_class="D"/>
            <output name="Q" num_pins="1" port_class="Q"/>
            <clock name="clk" port_class="clock"/>
          </pb_type>
          <pb_type name="reg_18x18_B" blif_model=".latch" num_pb="18"
             class="flipflop">
            <input name="D" num_pins="1" port_class="D"/>
            <output name="Q" num_pins="1" port_class="Q"/>
            <clock name="clk" port_class="clock"/>
          </pb_type>
          <pb_type name="reg_18x18_out" blif_model=".latch"
             num_pb="36" class="flipflop">
            <input name="D" num_pins="1" port_class="D"/>
            <output name="Q" num_pins="1" port_class="Q"/>
```

```
      <clock name="clk" port_class="clock"/>
    </pb_type>

    <pb_type name="mult_18x18" blif_model=".subckt mult"
       num_pb="1">
      <input name="A" num_pins="18"/>
      <input name="B" num_pins="18"/>
      <output name="OUT" num_pins="36"/>
    </pb_type>

    <interconnect>
      <direct input="mult_18x18_slice.A_cfg"
         output="reg_18x18_A[17:0].D"/>
      <direct input="mult_18x18_slice.B_cfg"
         output="reg_18x18_B[17:0].D"/>
      <mux input="mult_18x18_slice.A_cfg reg_18x18_A[17:0].Q"
         output="mult_18x18.A"/>
      <mux input="mult_18x18_slice.B_cfg reg_18x18_B[17:0].Q"
         output="mult_18x18.B"/>

      <direct input="mult_18x18.OUT"
         output="reg_18x18_out[35:0].D"/>
      <mux input="mult_18x18.OUT reg_18x18_out[35:0].Q"
         output="mult_18x18_slice.OUT_cfg"/>

      <complete input="mult_18x18_slice.clk"
         output="reg_18x18_A[17:0].clk"/>
      <complete input="mult_18x18_slice.clk"
         output="reg_18x18_B[17:0].clk"/>
      <complete input="mult_18x18_slice.clk"
         output="reg_18x18_out[35:0].clk"/>
    </interconnect>
  </pb_type>
  <interconnect>
    <direct input="divisible_mult_18x18.A"
       output="mult_18x18_slice.A_cfg"/>
    <direct input="divisible_mult_18x18.B"
       output="mult_18x18_slice.A_cfg"/>
    <direct input="mult_18x18_slice.OUT_cfg"
       output="divisible_mult_18x18.OUT"/>
    <complete input="divisible_mult_18x18.clk"
       output="mult_18x18_slice.clk"/>
  </interconnect>
</mode>

<mode name="two_mult_9x9">
  <pb_type name="mult_9x9_slice" num_pb="2">
    <input name="A_cfg" num_pins="9"/>
    <input name="B_cfg" num_pins="9"/>
    <input name="OUT_cfg" num_pins="18"/>
    <clock name="clk"/>

    <pb_type name="reg_9x9_A" blif_model=".latch" num_pb="9"
       class="flipflop">
```

```
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    <pb_type name="reg_9x9_B" blif_model=".latch" num_pb="9"
       class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    <pb_type name="reg_9x9_out" blif_model=".latch" num_pb="18"
       class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>

    <pb_type name="mult_9x9" blif_model=".subckt mult"
       num_pb="1">
      <input name="A" num_pins="9"/>
      <input name="B" num_pins="9"/>
      <output name="OUT" num_pins="18"/>
    </pb_type>

    <interconnect>
      <direct input="mult_9x9_slice.A_cfg"
         output="reg_9x9_A[8:0].D"/>
      <direct input="mult_9x9_slice.B_cfg"
         output="reg_9x9_B[8:0].D"/>
      <mux input="mult_9x9_slice.A_cfg reg_9x9_A[8:0].Q"
         output="mult_9x9.A"/>
      <mux input="mult_9x9_slice.B_cfg reg_9x9_B[8:0].Q"
         output="mult_9x9.B"/>

      <direct input="mult_9x9.OUT" output="reg_9x9_out[17:0].D"/>
      <mux input="mult_9x9.OUT reg_9x9_out[17:0].Q"
         output="mult_9x9_slice.OUT_cfg"/>

      <complete input="mult_9x9_slice.clk"
         output="reg_9x9_A[8:0].clk"/>
      <complete input="mult_9x9_slice.clk"
         output="reg_9x9_B[8:0].clk"/>
      <complete input="mult_9x9_slice.clk"
         output="reg_9x9_out[17:0].clk"/>
    </interconnect>
  </pb_type>
  <interconnect>
    <direct input="divisible_mult_18x18.A"
       output="mult_9x9_slice[1:0].A_cfg"/>
    <direct input="divisible_mult_18x18.B"
       output="mult_9x9_slice[1:0].A_cfg"/>
    <direct input="mult_9x9_slice[1:0].OUT_cfg"
       output="divisible_mult_18x18.OUT"/>
    <complete input="divisible_mult_18x18.clk"
```

```
            output="mult_9x9_slice[1:0].clk"/>
        </interconnect>
      </mode>
    </pb_type>
    <interconnect>
      <direct input="block_mult.A"
         output="divisible_mult_18x18[1:0].A"/>
      <direct input="block_mult.B"
         output="divisible_mult_18x18[1:0].B"/>
      <direct input="divisible_mult_18x18[1:0].OUT"
         output="block_mult.OUT"/>
      <complete input="block_mult.clk"
         output="divisible_mult_18x18[1:0].clk"/>
    </interconnect>
  </mode>
</pb_type>
```