Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip

Shawn A. Phillips

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree: Electrical Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Shawn A. Phillips

and have found that it is complete and satisfactory in all respects, and that any and all revisions required by the final examining committee have been made.

Chair of the Supervisory Committee:

_____

Scott Hauck

Reading Committee:

_____

Scott Hauck

_____

Carl Ebeling

_____

Larry McMurchie

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature _____

Date _____

University of Washington

Abstract

Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip

by Shawn A. Phillips

Chair of the Supervisory Committee:

Professor Scott Hauck
Electrical Engineering

As technology scales, engineers are presented with the ability to integrate many devices onto a single chip, creating entire systems-on-a-chip. When designing SOCs, a unique opportunity exists to add custom reconfigurable fabric, which will provide an efficient compromise between the flexibility of software and the performance of hardware, while at the same time allowing for post-fabrication modification of circuits. Unfortunately, manually generating a custom reconfigurable fabric would make it impossible to meet any reasonable design cycle, while adding considerably to design costs. The goal of the Totem Project is to reduce the design time and effort in the creation of a custom reconfigurable architecture.

This thesis is focused on automating the layout generation of custom reconfigurable devices for systems-on-a-chip. Towards this end, we present three methods, namely the Template Reduction, Circuit Generator, and Standard Cell Methods. The Template Reduction Method begins with a full-custom layout as a template that is a superset of the required resources, and removes those resources that are not needed by a given application domain. The Circuit Generator Method takes advantage of the regularity that exists in FPGAs using circuit generators to create the custom reconfigurable devices. Finally, the Standard Cell Method automates the creation of circuits by using a standard cell library that has been optimized for reconfigurable devices.

Table of Contents

List of Figures

List of Graphs

List of Tables

Acknowledgements

This dissertation would not have been possible if not for all of the people that have contributed to this work. First, I would like to thank the RaPiD group, especially Carl Ebeling, Chris Fisher, Larry McMurchie, and Darren Cronquist for their work on the RaPiD project, which was an essential starting point for this research. More specifically, Chris Fisher provided the RaPiD I layouts and much needed guidance on their function and purpose.

Larry McMurchie was a lifesaver many times over when the "tools broke again". Larry also provided guidance and was always willing to help me use and optimize the tools and my tool flows. The sleepless nights would have been more frequent without all of his help.

Many graduate students also contributed to this work. Katherine Compton provided the Architecture Generator along with invaluable insight on the structure of the RaPiD II template. Akshay Sharma not only supplied the place and route tool, but also provided the reduction list generator for the Template Reduction Method. Kim Montonaga and Ken Eguro both provided the area and performance numbers for some of the standard cell functional units.

Other graduate students provided personal support during my graduate career. In the early years, Zhiyuan Li and Chandra Mulpuri were always willing to patiently listen as I ranted and raved about one topic after another. They were eventually replaced by Mark Chang, Akshay Sharma, Ken Eguro, and Mark Holland. Thanks guys.

Graduate school is tough, but financial support can take the edge off. This work was funded in part from grants from the National Science Foundation, the Defense Advanced Research Projects Agency, and the National Aeronautics and Space Administration. This work was also supported in part by an MIT Lincoln Laboratory Fellowship and the Northwestern University Walter P. Murphy Fellowship.

Additionally, I am forever indebted to my advisor Scott Hauck, who not only guided me through my graduate studies, but also helped me deal with all of the ups-and-downs that occurred in my life. No other advisor would have had the patience to stick with me, and I would not have gotten this Ph.D. without him.

Finally, my friends and family have been the driving force in my life. I would like to thank my mom for not only pushing me to pursue this Ph.D., but for being available to me for both emotional and financial support. I would have gotten nowhere without my sisters, Shannon and Shayne, who filled out my college application for me when I applied to undergraduate school, and the financial aid forms every year after. I would also like to thank my mother-in-law and Caryn Park for taking time out of their busy lives to help

watch Ella so that I could “get to work”. Finally, I would like to thank my beautiful wife, Joanie, and daughter, Ella, without whom none of this would have been worth it.

Dedications

To my mother for setting high standards, to my sisters for showing me the way, to my wife, Joanie, for getting me through, and to my daughter, Ella, for being my inspiration.

Chapter 1

Introduction

With the advent of new fabrication technologies, designers now have the ability to create integrated circuits utilizing over one hundred million gates, with operating frequencies in the GHz range. This large increase in transistor count has increased the complexity of devices, but it is also enabling designers to move away from the well known system-on-a-board to a heterogeneous system-on-a-chip (SOC) [1]. This evolution in integration is driven by the need to reduce the overall cost of the design, increase inter-device communication bandwidth, reduce power consumption, and remove pin limitations.

Unfortunately, there are a number of drawbacks to the SOC design methodology. Designers of SOCs have a larger design space to consider, an increase in prototype cost, a more difficult job of interfacing components, more complex design verification, and a longer time-to-market. There is also a loss in post-fabrication flexibility. In the system-on-a-board approach, designers have the ability to customize the system by careful selection of components, with easy component modification or replacement in late stages of the design cycle. However, in the current SOC design methodology framework, in which only application specific integrated circuit (ASIC) type components are used, very

tight integration is the goal. Therefore, component changes late in the design cycle are not feasible.

This loss of post-fabrication flexibility can be alleviated with the inclusion of general-purpose reconfigurable hardware into the SOC. However, general-purpose reconfigurable hardware is often several times slower, larger, and less energy efficient than ASICs, making them a less ideal choice for high performance, low power designs. Domain-specific reconfigurable hardware can be utilized to bridge the gap that exists between flexible general-purpose reconfigurable hardware and high performance ASICs.

Domain-specific reconfigurable hardware consists of a reconfigurable array that is targeted at specific application domain(s), instead of the multiple and often times wide ranging domains targeted by general-purpose reconfigurable hardware. Creating custom domain-specific reconfigurable hardware is possible when designing an SOC, since early in the design stage designers are aware of the computational domain in which the device will operate.

Possible application domains could include signal processing, cryptography, image analysis, or any set of algorithms that a design needs to support. In essence, the more that is known about the target applications, the more inflexible and ASIC-like the custom reconfigurable hardware can be. On the other end of the spectrum, if the domain space is only vaguely known, or the design calls for increased flexibility, then the custom reconfigurable hardware would need to provide the ability to run a wide range of

applications, and thus be required to look and perform more like a standard general-purpose reconfigurable device.

If the designer has knowledge of the specific application domain at which the device is targeted, designers could then remove from the reconfigurable array hardware and programming points that are not needed and would otherwise reduce system performance and increase the design area. Architectures such as RaPiD [2], PipeRench [3], and Pleiades [4], have followed this design methodology in the digital signal processor (DSP) computational domain, and have shown improvements over general-purpose reconfigurable hardware within this space. This ability to utilize custom reconfigurable hardware instead of ASICs in high performance SOC designs will provide the post-fabrication flexibility that general-purpose reconfigurable hardware provides, while also meeting stringent performance requirements that until now could only be met through the use of ASICs.

In the traditional system-on-a-board design methodology, the total cost of fabricating custom reconfigurable hardware devices would inherently be subject to high nonrecurring engineering (NRE) costs, and thus be infeasible in most circumstances. However, since all of the components in an SOC need to be fabricated after integration, the SOC design methodology provides designers with a unique opportunity to insert application specific reconfigurable hardware into their devices, without accruing additional NRE cost. Unfortunately, if designers were forced to create custom logic for each application domain, it would be difficult if not impossible to meet any reasonable design cycle.

However, by automating the generation of the application specific reconfigurable hardware, designers would avoid this increased time-to-market and would also decrease the overall design cost, moving the creation and integration of these types of devices from an expensive proposition to a worthwhile endeavor.

These factors have led us to start the Totem Project [5, 6, 7, 8, 9, 10], which has the ultimate goal of reducing the design time and effort in the creation of custom reconfigurable architectures. The architectures that are created by Totem are based upon the applications and constraints specified by the designer. Since the custom reconfigurable architecture is optimized for a particular set of applications and constraints, the designs that are created by the Totem Project are smaller in area and perform better than a standard general-purpose reconfigurable processor, while retaining enough flexibility to support the specified application set.

The work presented here describes my research in methodologies, practices, and tools for the effective automatic generation of VLSI layouts of domain specific reconfigurable hardware. This dissertation is organized as follows:

- *Chapter 2: Reconfigurable Computing* provides a technical background discussion of the technologies involved, which includes an introduction to general purpose reconfigurable devices and, in particular, the RaPiD architecture.

- *Chapter 3: Reconfigurable Hardware in SOCs* highlights similar research efforts by both academia and industry in embedding reconfigurable logic into SOCs.

- *Chapter 4: Totem* provides an overall view of the Totem Project and, more specifically, how this thesis fits into the tool flow of the Totem Project.

- *Chapter 5: Research Framework* provides the structural basis for the work presented here, including the common benchmark sets and an introduction to the metrics used to evaluate the various methods of automating the VLSI layout of application specific reconfigurable devices.

- *Chapter 6: Template Reduction Method* presents a method of automating the VLSI layout generation of reconfigurable devices, which begins with a full-custom layout as a template that is a superset of the required resources, and removes those resources that are not needed by a given application domain.

- *Chapter 7: Circuit Generator Method* presents a method of automating the VLSI layout of reconfigurable devices by taking advantage of the regularity that exists in FPGAs through the use of circuit generators to create the custom reconfigurable devices.

- *Chapter 8: Standard Cell Method* presents a flexible method of automating the VLSI layout of reconfigurable devices by using a standard cell library that has been optimized for reconfigurable devices, to fill the gaps that exist between the templates and to alleviate the need for a wide range of generators.

- *Chapter 9: Conclusions and Future Work* concludes this dissertation with a comparison of the three methods and a brief discussion of future research directions.

Chapter 2

Reconfigurable Hardware

With increasing adaptability and performance, and a trend toward lower cost devices, reconfigurable hardware is being used in a wider range of applications than ever before [47]. This chapter presents a technical background on reconfigurable hardware, beginning with an architectural overview. The next two sections are in-depth case studies of particular reconfigurable devices. The final section is a brief outline of the software used to implement algorithms onto reconfigurable hardware.

2.1 Architectural Overview

Hardware logic devices can be divided into two broad categories, namely fixed logic devices and reconfigurable devices. Fixed logic devices, be they application specific integrated circuits (ASICs) or a board-level solution containing multiple components, are optimized at design time to perform a specific task or group of tasks efficiently. On the other hand, reconfigurable hardware is designed with the ability to handle multiple tasks and can be changed, or reconfigured, at any time.

Fixed logic is customarily the preferred device type when the application domain is well known in advance, the design is targeted at a high volume application domain, and high

performance, low power, or small device size are important design goals. However, while fixed logic devices are used in a wide range of application domains, there are several drawbacks associated with these types of devices. One of the main drawbacks of fixed hardware is the fact that once the device is manufactured it is, as the name implies, fixed, and therefore cannot be modified. Consequently, if the device does not work as expected or there is a change in the design requirements, the device must be redesigned, which, as we will see, can be an extremely costly proposition. Another drawback to the use of fixed logic devices is the long and costly time-to-market. It can take several months to years to design and verify a fixed logic device, depending upon the size and complexity of the device. And, depending upon the size and complexity of the design, the upfront, or nonrecurring engineering (NRE), costs can range from hundreds of thousands to several millions of dollars.

Reconfigurable logic devices are typically standard off-the-shelf components that offer a wide range of logic capacity, I/O capabilities, performance, and power characteristics. With the device specific software tools provided by the logic vendor, designers are able to quickly prototype and test their designs on a working circuit, with the knowledge that the reconfigurable logic device that they are testing their design on is the same device that will be embedded into the final system. In addition, if there is any modification to the design, even after the final system has been shipped, the reconfigurable logic can be modified to reflect these changes. This removes almost all of the NRE cost, and allows for designs that have an extremely short time-to-market and a bug fix or upgrade path. In

the next sections, two different types of reconfigurable hardware are detailed: the field programmable gate array (FPGA) and the reconfigurable pipelined datapath (RaPiD).

2.2 Field Programmable Gate Array (FPGA)

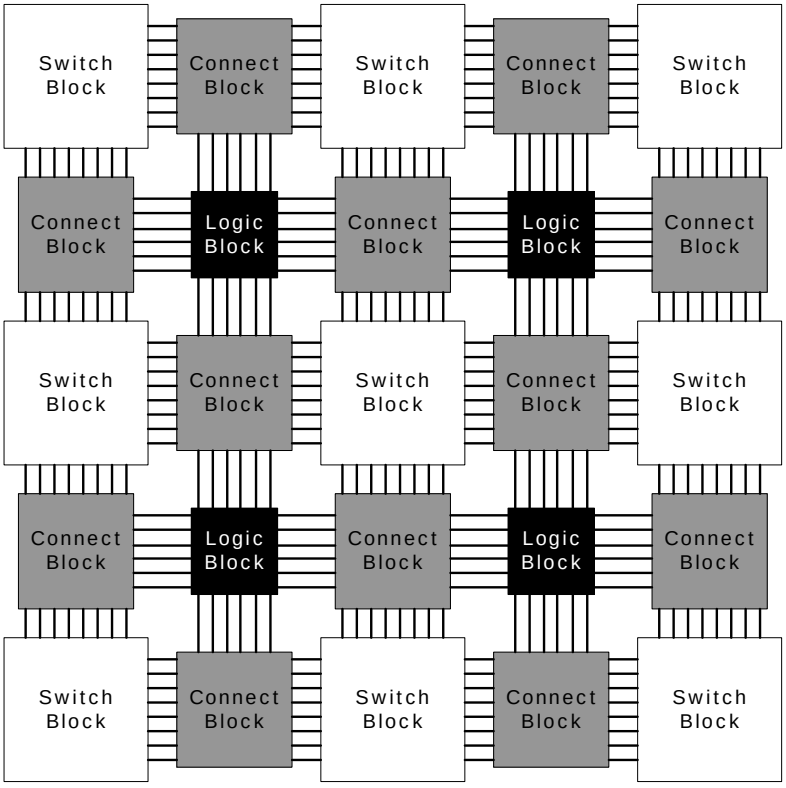


Figure 2-1: Xilinx style FPGA architecture. It contains an array of CLBs, switchboxes, and vertical and horizontal routing channels.

An FPGA is a multi-level logic device that can be reconfigured to support a wide variety of application domains. Conceptually, an island-style FPGA (which is the most common type of FPGA manufactured today) is an array of configurable logic blocks (CLBs). CLBs comprise the computational fabric within the FPGA and are embedded within horizontal and vertical channels of routing [11]. The routing channels, like CLBs, have the capability to be personalized to interconnect any one CLB with any other CLB within

the array by the use of connection boxes and switch boxes. The configuration of the FPGA is commonly held in static RAM (SRAM) cells that are distributed across the chip. By placing the configuration bits in SRAM cells, the FPGA can be reprogrammed many times over the life of the device. Thus, the FPGA has the ability to run many different configurations, akin to how a general-purpose processor can run many different programs. The ability of an FPGA to run such a wide range of programs is only limited by the number CLBs that are in the array and by the richness of the routing fabric. Figure 2-1 shows an island-style FPGA, which was first popularized by Xilinx in 1984 [6].

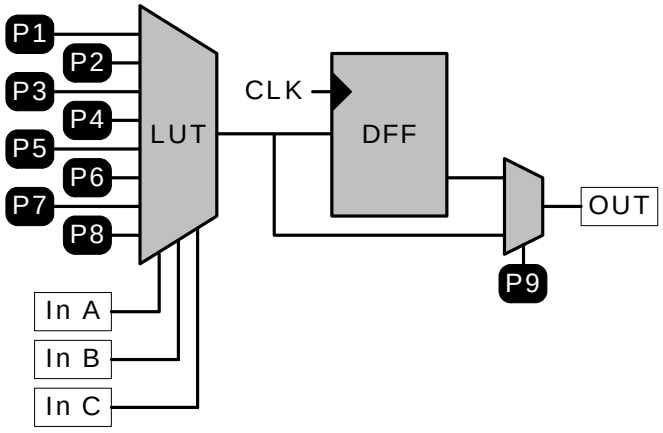


Figure 2-2: A basic configurable logic block (CLB) containing a three input LUT and a D-type flip-flop with bypass.

In an island-style FPGA, a CLB (an example of which is shown in Figure 2-2) typically consists of a lookup table (LUT), a state holding element such as a flip-flop, and multiplexers. LUTs are small memories to which the truth table for the desired function can be written when the FPGA is configured. Multiplexers within the CLB, an example of which is shown in Figure 2-2 in the form of the bypassing multiplexer, are controlled by programming bits that enable different functional blocks within the CLB to be chosen,

depending on the application that is being mapped onto the FPGA. By using LUTs, flip-flops, and multiplexers, island-style FPGAs are capable of implementing arbitrary logic functions, with each CLB capable of handling functions that typically have no more than three or four inputs.

The routing structure shown in Figure 2-1 enables point-to-point connections between CLBs, which is achieved by using connection blocks and switch blocks. Connection blocks, which utilize programmable switch points, enable input and output signals to travel from logic blocks into the routing channel and vice versa. Once signals are in the routing channels, switch boxes carry them throughout the FPGA by enabling corner turning and in some cases switching between routing tracks. Switch boxes, like connection boxes, are made up of user programmable switch points.

2.3 Case Study: The Altera Cyclone II FPGA

The island-style FPGA presented thus far is a simplified version of a modern FPGA, and therefore does not adequately characterize the complexity and functionality present in state-of-the-art devices. Modern devices available from vendors such as Xilinx and Altera have begun to blur the line between the fine-grained island-style device described above, and a more medium-grained device. This change in the granularity has been brought about by the inclusion of coarse-grained components into the reconfigurable fabric like word width multipliers and, more recently, embedded general-purpose processors. An understanding of how these medium-grained devices compare to both fine-grained island-style devices and coarse-grained devices like RaPiD will help the

reader understand the types of reconfigurable devices that the Totem Project will produce. Therefore, an evaluation of the Cyclone II FPGA family of devices, which has been developed by the Altera Corporation [46], is presented.

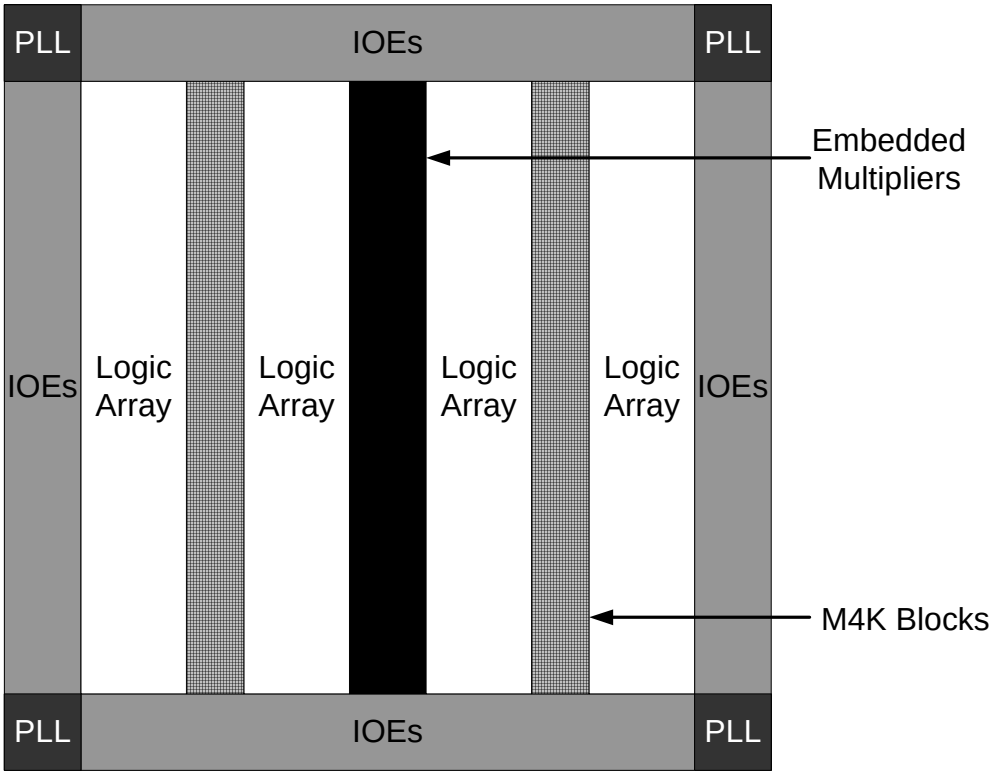


Figure 2-3: A block diagram of the Cyclone II EP2C20 Device [46].

The Cyclone II FPGA is an island-style device that contains logic array blocks (LABs), embedded memories, embedded floating-point multipliers, multi-function I/O elements (IOEs), and up to 4 phase-locked loops (PLLs). All of these functional elements are connected to one another by a two-dimensional multi-track interconnect fabric. A block diagram highlighting all of the functional elements of the Cyclone II device is shown in Figure 2-3.

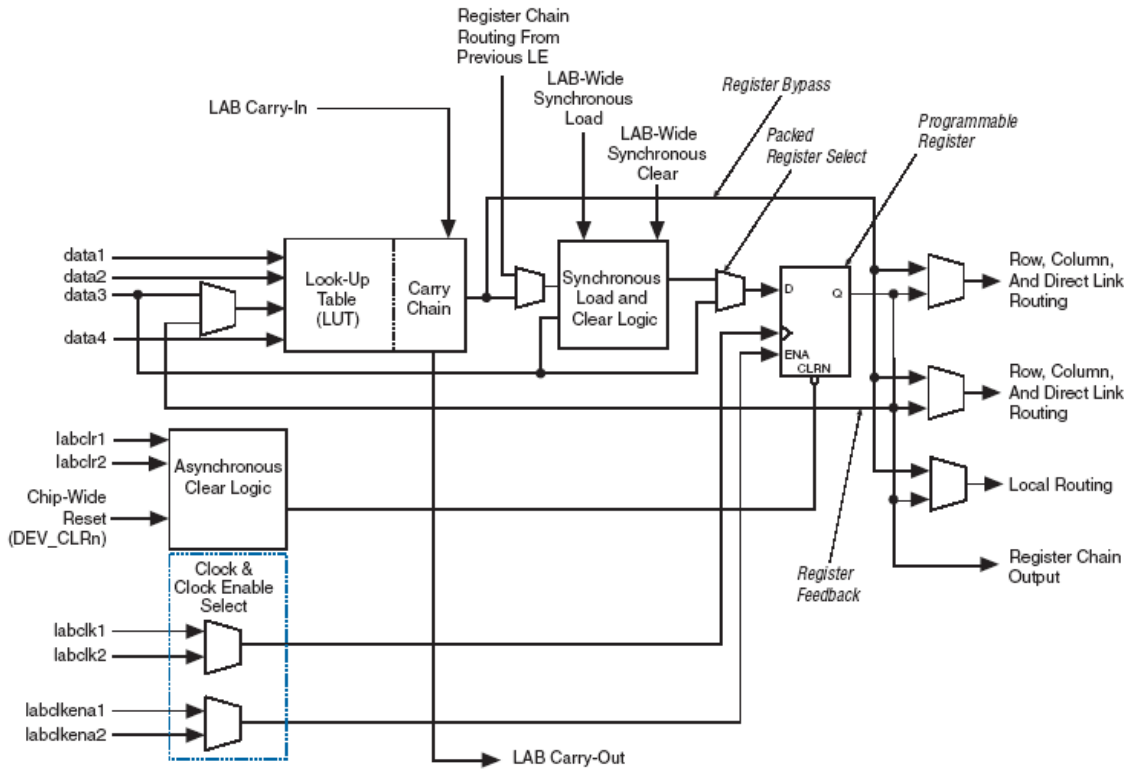


Figure 2-4: Block Diagram of the Cyclone II logic element (LE) [46]. The LE is the smallest functional unit on the Cyclone II device. Each LE contains the following: a four-input LUT, a programmable register, a carry chain connection, and a register chain connection.

2.3.1 Cyclone II Logic Elements

LABs in the Cyclone II are made up of sixteen individual logic elements (LEs). The LE is the smallest functional unit in the Cyclone II, and a block diagram is shown in Figure 2-4. LEs contain the following: a four-input LUT, a programmable register that can be configured for D, T, JK, or SR operation, a carry chain connection, and a register chain connection. Each LE is capable of register packing. Register packing occurs when the LUT drives one output while the register drives another output. This means that the LUT and the register can be used for unrelated functions, thus enabling the device to more

efficiently utilize its resources. Each LE is also capable of register feedback. The register feedback mode enables the register to receive as input its own LUT's output, providing both registered and unregistered LUT outputs. The register chain output enables registers within the same LAB to be daisy-chained together, and when coupled with register packing enables the LAB to perform one combinational function with its LUTs while also allowing for an unrelated shift register function. LEs can operate in either normal or arithmetic mode, depending upon what type of function the LEs implement.

2.3.3 Logic Array Block (LAB)

The LAB, shown in Figure 2-5, consist of sixteen LEs, as well as control signals, LE carry chains, register chains, and local interconnect. The control signals that are present in each LAB consists of two clocks, two clock enables, two asynchronous clears, and one synchronous clear. This provides the LAB with seven independent control signals, along with a chip wide register clear that overrides all other control signals. The register chain enables one LE to transfer its register output to the input of an adjacent LE register, with the carry chain operating in a similar manner. The LAB local interconnect consists of local connections within the LAB, and direct connections between adjacent LABs, PLLs, M4K RAM blocks, and embedded multipliers. Local and direct link connections also enable each LE to drive up to 48 LEs.

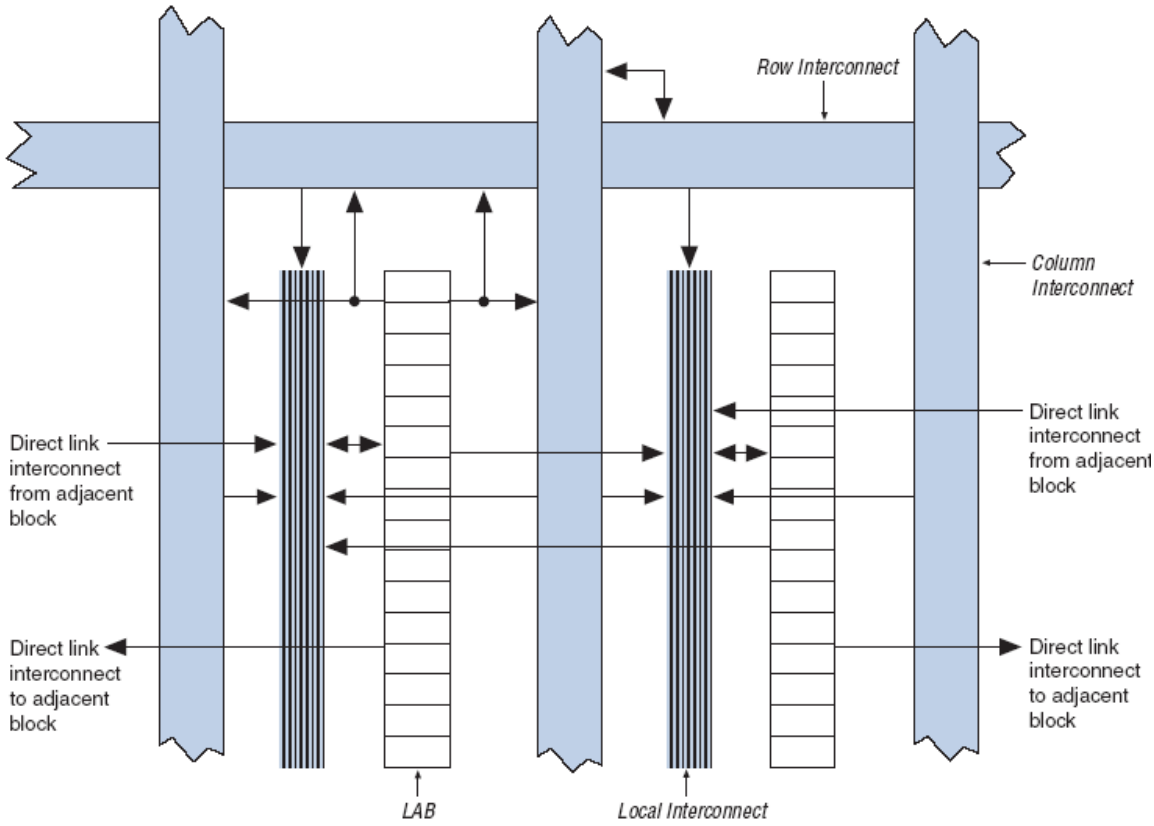


Figure 2-5: LAB Structure [46], which provides local interconnect for LEs in the same LAB as well as direct connections between adjacent LE's within an LAB.

2.3.4 Embedded Multipliers

The Cyclone II device contains embedded multiplier blocks that are capable of operating as one 18-bit multiplier or as two independent 9-bit multipliers. The maximum operating frequency for the multiplier blocks is 250 MHz, and is independent of the multiplier operation mode. Each multiplier block is the height of one LAB, and each device can contain one to three multiplier blocks, depending upon the size of the Cyclone II device. The inputs and the outputs of the multiplier block can be registered or unregistered, and can be either signed or unsigned values. The multiplier block can be connected to the

LABs on its left and right with either column or row resources, as shown in Figure 2-6. By using direct link resources instead of column or row resources, the multiplier can have up to 16 input connections to the LAB on the right, and up to 16 input connections to the LAB on the left. In addition, by using direct link, the multiplier block can drive up to 18 output connections to the LAB on the left and to the LAB on the right.

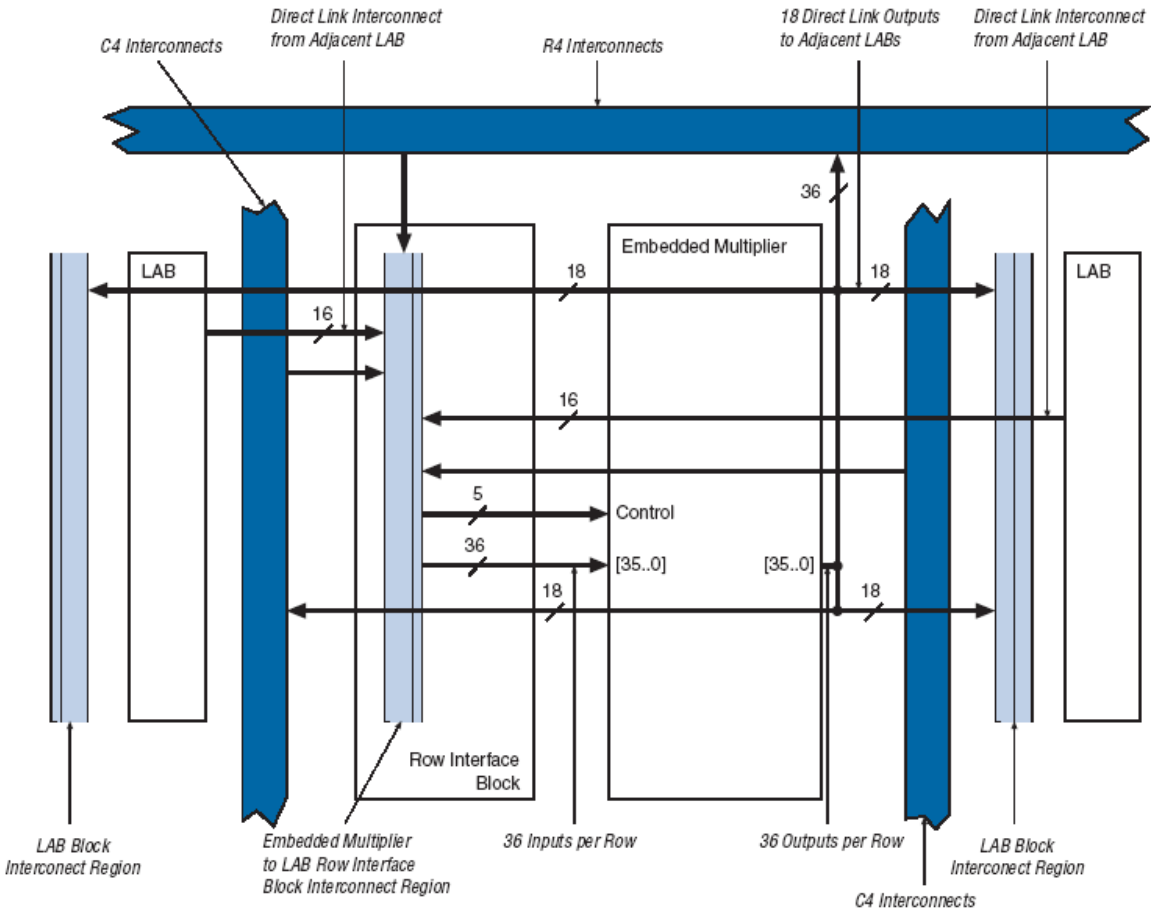


Figure 2-6: An embedded multiplier and its corresponding LAB row interface [46].

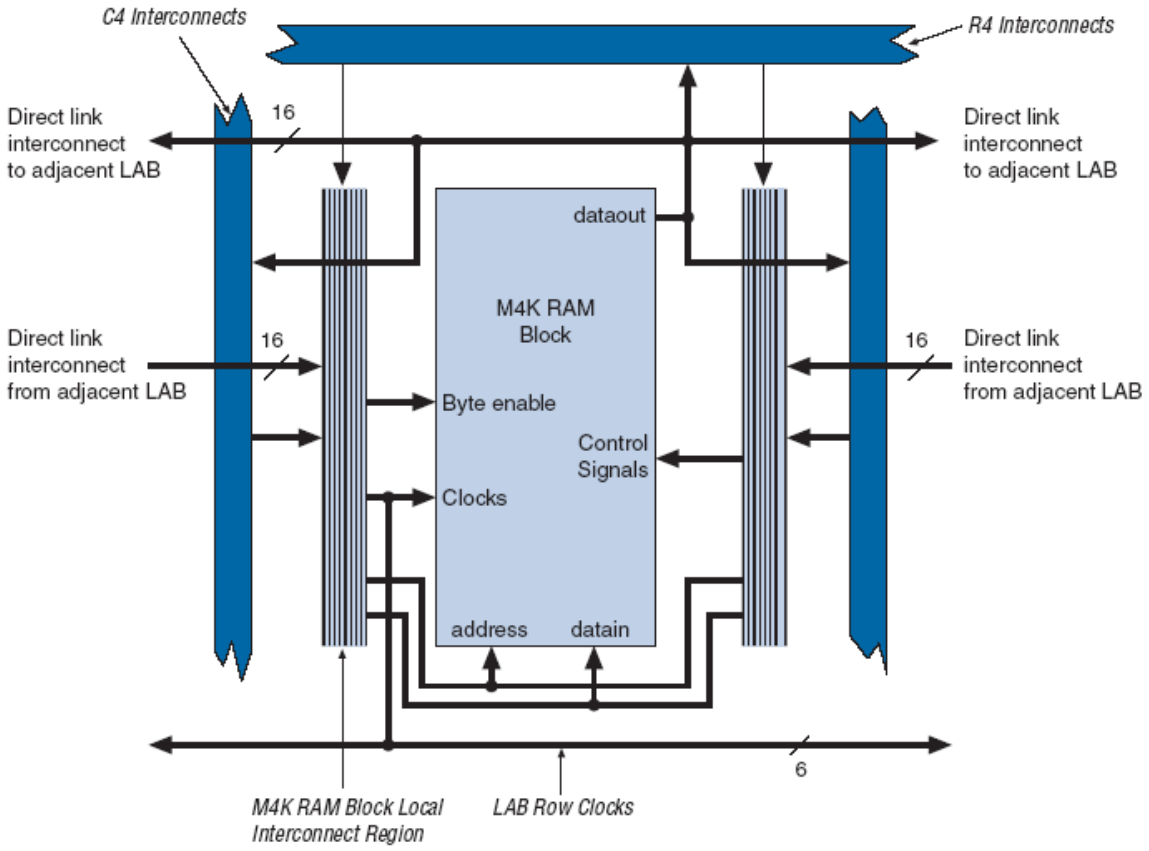


Figure 2-7: An embedded four Kb memory block (M4K), and its corresponding LAB row interface [46].

2.3.5 Embedded Memory

The Cyclone II family of devices also contains embedded memory blocks (M4K), with each block containing 4,608 total memory bits that are broken into 4,096 functional bits and 512 parity bits, as shown in Figure 2-7. The memories can operate at 250 MHz, and can be configured as a true dual-port memory, simple dual-port memory, single-port memory, byte enable, shift register, first-in first-out (FIFO) buffer, and read only memory (ROM). Like the embedded multiplier, the M4K blocks can be feed data from LABs on the left or on the right with either column and row resources, or with a direct link

connection. When using direct link connections, 16 bits from the LAB directly to the left of the M4K block and 16 bits from the LAB directly to the right of the M4K block can be feed directly into the memory block. The M4K block can drive outputs to the column and row resources or it can utilize up to 16 direct link connections.

2.3.6 MultiTrack Interconnect

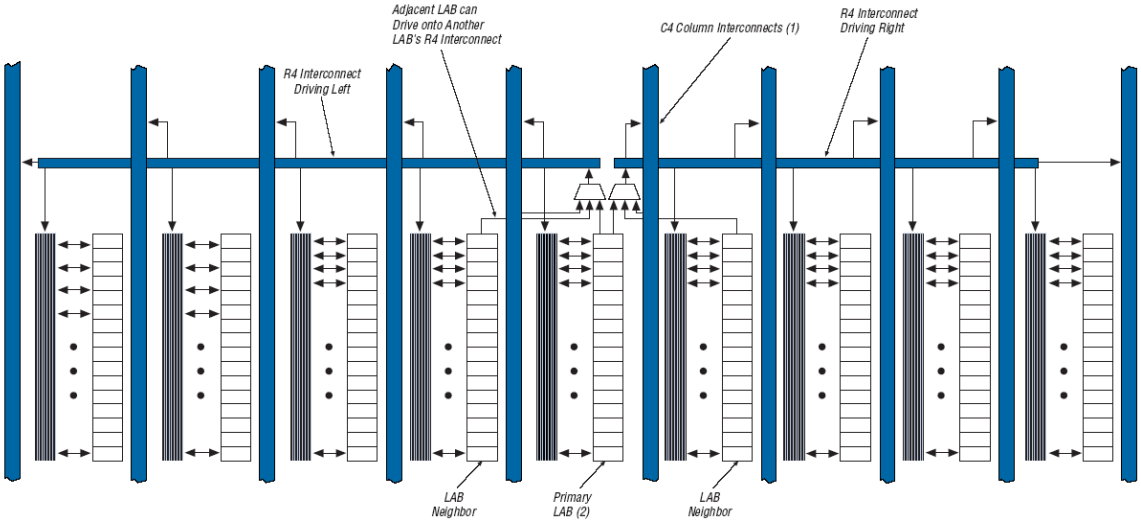


Figure 2-8: R4 row interconnect connections. R4 interconnects span four LABs, three LABs and one M4K memory block, or three LABs and one embedded multiplier to the right or left of a source LAB [46].

All of the functional units mentioned thus far are embedded within a two dimensional routing fabric that has been dubbed MultiTrack by Altera. The MultiTrack routing fabric is comprised of row and column interconnects, as well as row direct link and register chain connections.

The row interconnect consists of both R4 and R24 connections, as well as direct link connections. R4 connections, as shown in Figure 2-8, span four functional units in the

horizontal direction, namely four LABs, three LABs and one embedded multiplier, or three LABs and one embedded memory block. R24 connections span 24 LABs, but can only be driven by R4 or column interconnects, not directly by LABs. Direct link connections, as mentioned above, allow direct connections to the left or right of LABs, embedded multipliers, or embedded memories within the same row.

The column interconnect consists of both C4 and C16 connections, as well as register chain connections. C4 connections span four functional units in the vertical direction, as shown in Figure 2-9. The functional units that are connected in this way are four LABs, four embedded multipliers, or four embedded memory blocks. The C16 interconnect spans sixteen functional units in the vertical direction, but can only be driven by R4, R24, C4, and other C16 interconnects. The final column interconnect consist of the register chain connections. Register chain connections allow for fast and efficient communication between LEs in the same LAB, as discussed previously in the LAB section.

In this brief outline of the Cyclone II family of devices, several features and details have been left out, since they are beyond the scope of this dissertation. Please see [47] for more information.

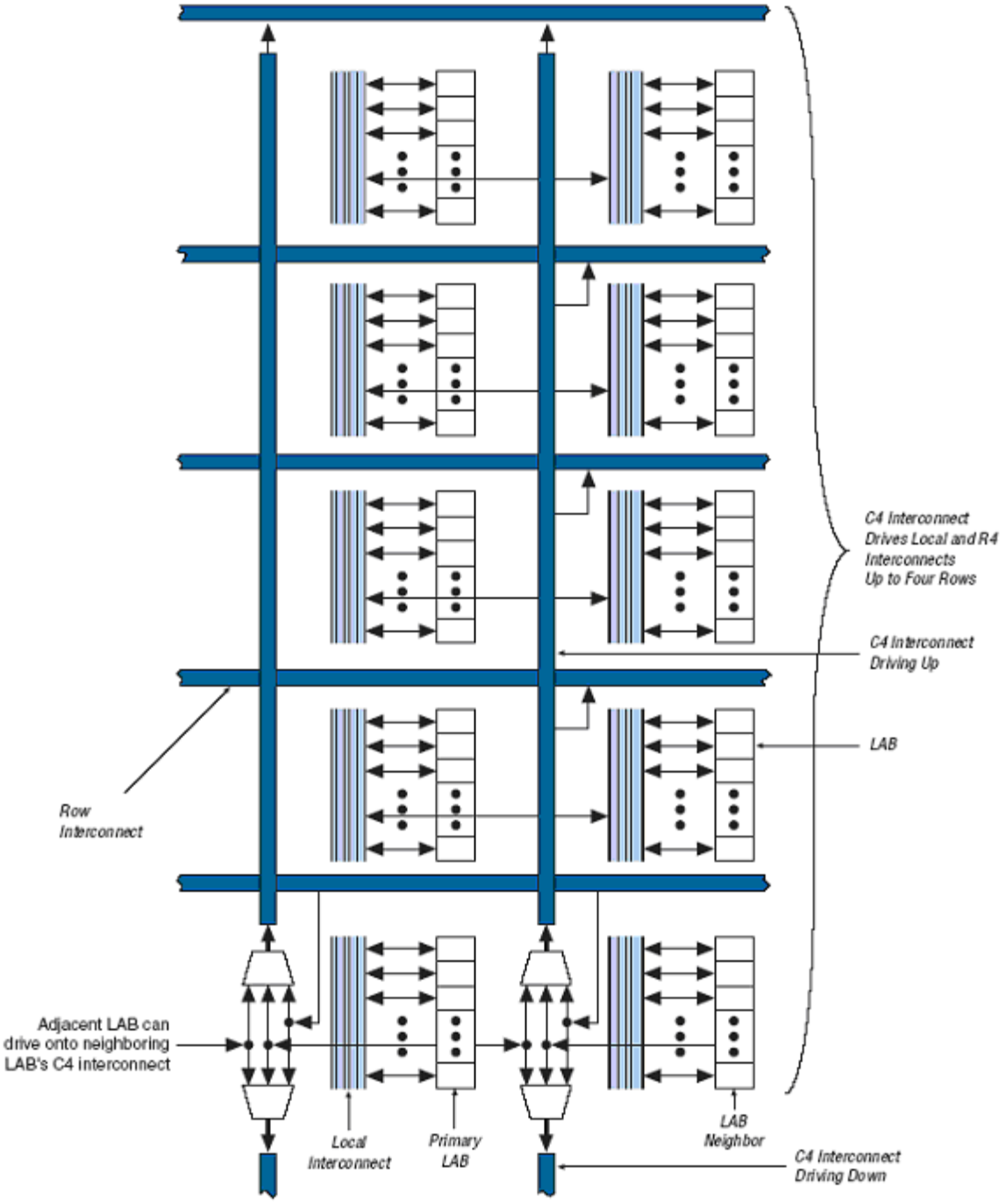


Figure 2-9: C4 interconnect, which traverses four blocks in the vertical direction [46].

2.4 Case Study: Reconfigurable-Pipelined Datapath (RaPiD)

We are using the reconfigurable-pipelined datapath (RaPiD) architectural style as a starting point for the circuits that we will be generating [49]. Therefore, it is important that we have a thorough understanding of what distinguishes RaPiD devices from more general-purpose reconfigurable devices like FPGAs.

RaPiD does not refer to any particular architecture, but instead refers to an architectural style that is targeted at highly repetitive, computationally intensive applications with large data sets [2], like digital signal processing, graphics, scientific computing, and communications. Towards this end, RaPiD architectures are composed of coarse-grained components, a one-dimensional routing interconnect, and static configuration with dynamic control. This mix of components enables RaPiD architectures to create very deep computational pipelines.

The computational units in a RaPiD device are specialized for the particular application domain at which the device is targeted, and can include word width ALUs, multipliers, and memories. The computational units in RaPiD are arranged along the horizontal axis, and connected to each other via a one-dimensional word width routing interconnect. One of the advantages of a one-dimensional structure is a reduction in the complexity, especially in the communications network. Another advantage is the ability to map systolic arrays very efficiently. Finally, while a two dimensional RaPiD is possible, most two-dimensional algorithms can be mapped onto a one-dimensional array by using

memory elements. There are several versions of RaPiD devices, which include the original RaPiD-I architecture [2], a more recent version, RaPiD-Benchmark [49], and RaPiD-II [63], a version that was created specifically for this work.

2.4.1 RaPiD Datapath

The original version of RaPiD, RaPiD-I [2], which is shown in Figure 2-10, consists of 16 bit wide memories and general purpose registers, 16 bit ALUs, and 16x16 multipliers. Each functional unit has 16 bit multiplexers on each of its inputs, and 16 bit demultiplexers on each of its outputs. This enables functional units to choose between different tracks for input and output signals. Data flows into functional units on the left, and flows out of functional units on the right. Figure 2-10 shows an example of one RaPiD-I cell. Multiple cells are tiled along the horizontal axis to create longer, more capable arrays.

Functional units are connected by a one-dimensional segmented routing structure. Data flows through the array along the horizontal axis, with the vertical axis only being used to provide connections between functional units. Each track in RaPiD-I is 16 bits wide, which corresponds to the bit width of the functional units. There are 14 routing tracks in RaPiD-I, made up of four local, and ten long-distance tracks. The local tracks allow fast, short-distance communication between functional units. The ten long tracks allow for long-distance communications. The long tracks are broken by bus connectors, which are represented by the black boxes in Figure 2-10. Bus connectors are bidirectional switches that enable either a break in the track or a connection. In RaPiD-I, bus connectors also

contain three delay registers, which help with the pipelining of signals. Along with the 14 routing tracks, there is a feedback track that enables functional units to route their output signals into their input. Finally, each of the input multiplexers is capable of providing a logical zero to functional units. While we have detailed the RaPiD-I datapath in this section, the ideas can be generalized to other versions of RaPiD.

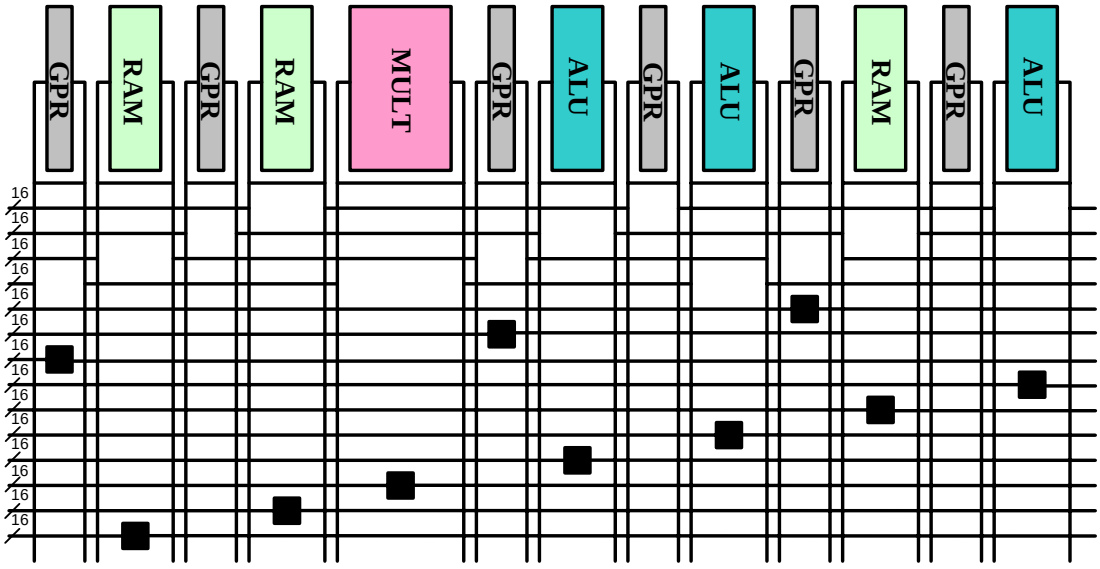


Figure 2-10: A block diagram of a basic RaPiD-I cell. The functional units in this cell are located at the top of the diagram, with the routing resources located at the bottom. The black boxes represent bus connectors, which are bidirectional switches that enable the creation of long lines. Multiple cells are tiled along a one-dimensional horizontal axis.

2.4.1 RaPiD Control Architecture

The control architecture of RaPiD consists of both static and dynamic parts. The static configuration is held in SRAM cells, which are initialized when the application is first mapped onto the RaPiD array. The static configuration, referred to as hard control, is fixed for the duration of the application, and is similar to how an FPGA operates. The

dynamic control, or soft control, is capable of changing every cycle, unlike the fixed hard control. To implement the soft control RaPiD utilizes small parallel instruction generators, which execute a series of simple instructions. These simple instructions are decoded at various points throughout the RaPiD array to create the soft control signals. The ratio between hard and soft control is approximately 75% and 25% respectively. By utilizing both hard and soft control, the RaPiD architecture is able to implement very complex control.

2.5 Developing for Reconfigurable Devices

To take advantage of reconfigurable devices, application developers require a software design environment that is capable of creating configurations for the reconfigurable hardware in a straightforward manner. If the creation of designs for reconfigurable devices is too complex, designers may overlook them in lieu of other types of devices like ASICs or general-purpose processors. Design software for reconfigurable devices can range from very basic software that helps in the manual creation of circuits, to software that is capable of handling algorithms written in a high-level language, such as C++, MATLAB, or Java.

As the complexity of reconfigurable devices has increased, the manual creation of circuits is falling out of favor. However, if a design has very tight timing and area constraints, manual circuit creation may be the only method capable of creating designs that can meet these constraints. The drawbacks associated with this method of circuit creation are the need for an extensive knowledge of the underlying reconfigurable

architecture and an increase in the design time. As an alternative to manual circuit creation, the automatic generation of circuits from high-level descriptions provides designers with a fast and easy way to create programs that can utilize reconfigurable devices, at a loss of quality. These types of tools have the potential to broaden the appeal of reconfigurable devices to a wider audience of users. One possible design flow is shown in Figure 2-11.

The design steps in the design flow depicted in Figure 2-11 are design entry using either schematic capture or through a hardware description language (HDL), synthesis from Register-Transfer Level (RTL) or behavioral HDL to gate level, and finally physical design. Place and route of the design is done using the FPGA vendor proprietary tools, which take into account the devices architecture and logic block structure. Back arrows represent possible improvement paths, depending upon whether a design has met specified timing or area constraints.

This brief introduction to FPGAs and to FPGA and the RaPiD architecture is sufficient to understand the context of the work in this dissertation. However, a more general and thorough survey of FPGAs and reconfigurable computing can be found in [62].

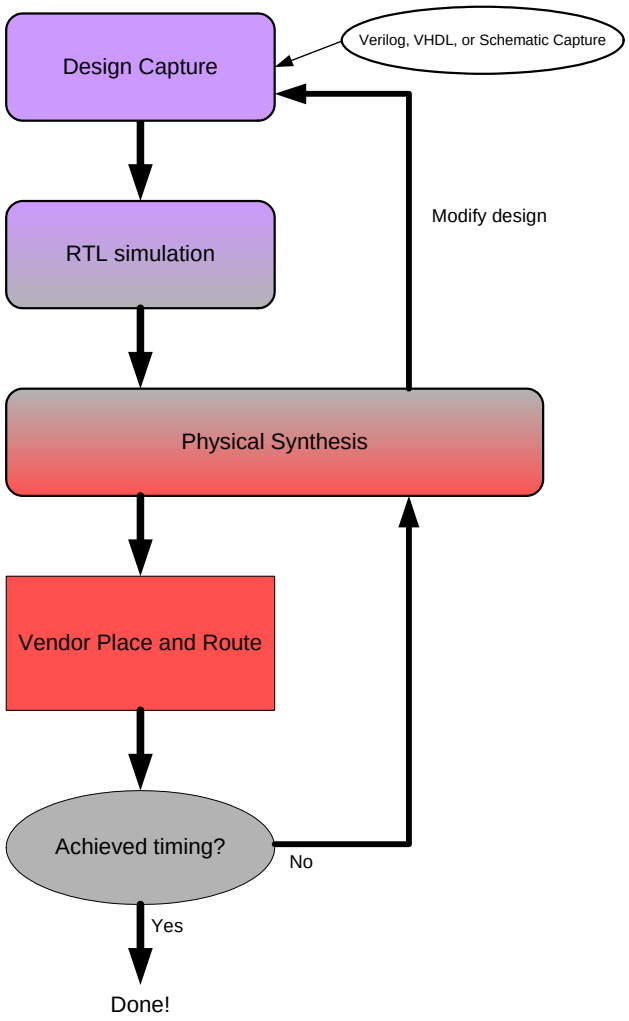


Figure 2-11: High-level view of a possible FPGA design flow. The steps in the process are: design entry, physical synthesis from RTL to gate level, and physical design. Place and route of the design is done using the FPGA vendor proprietary tools, which take into account the devices architecture and logic block structure. Back arrows represent possible improvement paths [57].

Chapter 3

Reconfigurable Hardware in SOCs

The integration of reconfigurable logic onto SOCs is currently being addressed by two different architectural approaches. The first architectural approach, discussed further in section 3.1, involves either embedding current "off-the-shelf" FPGA designs onto the SOC as one of the many individual SOC components, or creating domain specific SOCs that have reconfigurable components. The second architectural approach, discussed in section 3.2, is the creation of FPGAs that are powerful enough to support all of the functionality of an SOC. In essence, these devices, called systems-on-a-programmable-chip (SOPCs), have the potential to blur the line between reconfigurable devices and SOCs.

The two architectural methods have their advantages and disadvantages. SOCs that are targeted at a particular domain that also contain embedded reconfigurable subsystems have better performance, power, and area potential, since fixed IP blocks tend to be more efficient than generic reconfigurable logic that has been configured to perform the same function. Nevertheless, the performance, power, and area potential that targeted SOCs with embedded reconfigurable subsystems have, come at a loss of flexibility. On the other hand, SOPCs are highly flexible, and the same device can be targeted at many

different application domains, which removes the need to design and implement multiple SOCs for multiple application domains. The drawback to such flexibility is reduced performance, increased power consumption, and increased area. The following sections are a sampling of the projects that are currently being pursued.

3.1 Reconfigurable Subsystems

Embedding current "off-the-shelf" FPGA cores onto SOCs is driven by the desire of companies to leverage the massive amounts of time and resources that have been devoted to initially creating the cores, with the effect of creating new revenue streams. By opening up their IP libraries, companies are able to reuse designs without incurring large overhead costs. Xilinx, for example, provides some versions of their higher end FPGA cores to other companies such as IBM, though few specific details have been released. A cross-licensing agreement between the two companies has led to Xilinx embedding IBM PowerPC processors into its high-end parts, while IBM has embedded Xilinx reconfigurable blocks in some of its ASIC designs [53, 54].

Two other companies that are pursuing the idea of embedded FPGA cores are Actel and LSI. Actel has created a generic FPGA fabric that can be embedded into a SOC. They call it an embedded programmable gate array (EPGA), and it is part of their VariCore project [33]. The ASIC equivalent gate densities of VariCore EPGAs range from 5K to 40K, in 2.5K gate increments. LSI's approach is similar to Actel's, and is called LiquidLogic [34]. LiquidLogic consists of reconfigurable macro cells that have been

designed to be embedded into SOCs. The LiquidLogic core consists of reconfigurable logic in the form of a Multi-Scale Array (MSA) and the input-output bank that interfaces with other ASIC circuitry on the SOC. The smallest reconfigurable unit in a MSA is the Configurable Arithmetic Logic Units (CALU). The CALU is logically a 4-bit ALU that is created by four function cells and an ALU controller. To create an MSA, CALUs are tiled in 16x16 arrays called hex blocks, with the maximum size being a 4x4 array of hex blocks. Once the reconfigurable logic is in place, various softcores can be downloaded to it.

While the offerings from Xilinx, Actel, and LSI are programmable using SRAM bits, other companies are pursuing cores that are either mask or fuse programmable. Elixent and eASIC fall into this category, both offering devices that are designed for high-performance applications. Elixent is offering their D-Fabrix, which is based on a sea-of-ALUs [31]. A D-Fabrix array consists of up to several thousand 4-bit ALUs with registers. eASIC is capable of producing circuits that have a gate density that is as high as twenty times that of a conventional FPGA [36, 37]. To achieve this density, eASIC has shifted all routing from the bottom layers of diffusion to the top four layers. In effect, eASIC has removed most of the programmable interconnect while leaving the CLBs intact. As a result, eASIC eliminates most of the area overhead associated with FPGAs, creating a reconfigurable device that is similar to an anti-fuse based design. To route the FPGA, vias are inserted on the sixth layer of metal in the eight-metal-layer design. This means that post-fabrication modification of the interconnect in the circuit is not possible.

Reducing the interconnect enables designers to create very fast and efficient structures, but at the considerable loss of reconfigurability. Even though the interconnect of the circuit is set after fabrication, the reconfigurable cells can still be mapped to as needed.

The cases listed above in this section were concerned with providing reconfigurable subsystems to designers as a separate SOC component. However, there is another class of SOCs that already incorporate reconfigurable subsystems. These types of SOCs are designed with reconfigurable subsystems as an important, but not the central component. (Unlike the devices presented in the next section.) Atmel has created their Field Programmable System Level Integrated Circuit (FPSLIC™) [30] family of devices, which utilize a version of their configurable 8-bit microcontroller. This family of devices combines all the basic system building blocks, which include custom logic, memories, reconfigurable logic, and a RISC processor. Atmel enables designers to plug-in very specific custom logic into their device. Triscend offers a very similar device to that of Atmel, which is also a configurable 8-bit microcontroller [55, 56]. It contains a 32-bit ARM processor core, as well as a reconfigurable fabric and some hardwired peripherals.

3.2 Systems-on-a-Programmable-Chip (SOPCs)

Companies, including Altera [16], Lattice Semiconductor [17], and Xilinx [19], are trying to move the emphasis away from SOCs by providing systems-on-a-programmable-chip (SOPCs). The devices that are being created are capable of supporting the functionality

that was typically the province of SOCs. In essence, the reconfigurable device is the SOC.

The increase in the transistor count in devices has not only made SOCs possible, but has also enabled FPGA designers to create reconfigurable devices that are SOCs in their own right. One such part is the Virtex II Pro X [24] from Xilinx. This FPGA contains a multitude of features, including up to four PowerPCs [25], 24 embedded Rocket IO multi-gigabit transceivers, 12 digital clock managers, 556 18x18 multipliers, and ten megabits of block RAM. This is an expensive and large part, but it is fully capable of handling a wide range of applications that have previously been exclusively the domain of SOCs or systems-on-a-board.

Altera, like Xilinx, has a device that is capable of emulating a multitude of SOCs, specifically its Stratix APEX II™ [26]. Unlike Xilinx, Altera also offers what it brands as HardCopy devices [27]. HardCopy is a tool that enables designers to remove unwanted hardware from a feature rich reconfigurable device. To achieve this, a designer places and routes their design on an existing Altera device, and HardCopy creates a corresponding mask layout. This will fix the device to the desired application set, with the effect of removing any additional flexibility. The final HardCopy devices that are created are smaller in area by an average of 70%, and consume less power and perform better than the design mapped onto the original Altera Stratix device.

Lattice Semiconductor has decided to differentiate their product by enabling designers to pick specific ASIC cores that will be embedded with the reconfigurable fabric. Lattice

has termed these devices as Field Programmable System Chips (FPSC) [28]. While the reconfigurable fabric is generic in nature, the ASIC macrocells range from bus interfaces, high-speed line interfaces, and high-speed transceiver cores. The macrocells that are currently provided all fall within the networking domain, with the possibility for Lattice to create more macrocells in the future if designer demand is high enough. QuickLogic, with their QuickMIPS [29] part, provides a device that is very similar to what Lattice Semiconductor is offering with their FPSCs, but with the addition of an embedded 32-bit RISC core.

Xilinx is introducing a new family of devices, called Virtex-4 [45]. The Virtex-4 family is based upon an architectural approach that Xilinx has created to reduce the cost of developing multiple FPGA platforms, each with different combinations of feature sets. Xilinx has dubbed this new architectural approach as the Advanced Silicon Modular Block (ASMBL) architecture, though no details are currently available. The initial release of the Virtex-4 family will include devices that are targeted at four application domains, which include a Logic Domain, a DSP Domain, a Connectivity Domain, and an Embedded Processing Domain. These devices mark a shift for Xilinx in the creation of targeted reconfigurable devices, but they still fall into the family of powerful FPGAs that remove the need for a SOC. It is too early to ascertain how often and how quickly Xilinx will be able to create new devices targeting new application domains.

Chapter 4

Totem

Reconfigurable hardware is a very efficient bridge that fills the gap between software implementations running on general-purpose processors and ASIC implementations. However, standard reconfigurable hardware targets the general case, and therefore must contain a very generic mix of logic elements and routing resources that are capable of supporting all types of applications. This creates a device that is very flexible. Yet, if the application domain is known in advance, optimizations can be made to make a compact design that is more efficient than commercial FPGAs.

While the benefits of creating a unique FPGA for each application domain are apparent, in practice the design of a new FPGA architecture for every application space would require an increase in design time and create significant additional design costs. The goal of the Totem Project is the automatic generation of domain-specific FPGAs, giving designers a tool that will enable them to benefit from a unique FPGA architecture without high cost and a lengthy design cycle. The automatic generation of FPGAs can be broken into three major parts: high-level architecture generation, VLSI layout generation, and place-and-route tool creation, all of which will be discussed in this section. Figure 4-1 shows the tool flow between the major parts of Totem.

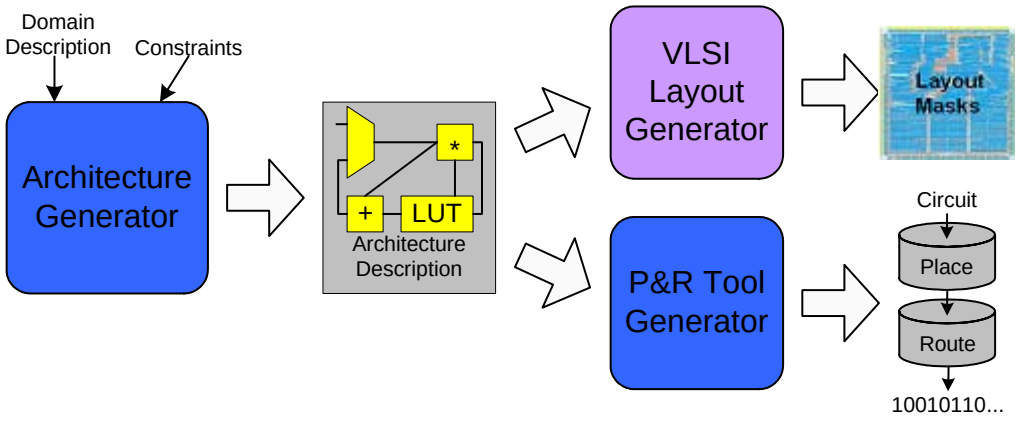


Figure 4-1: Totem tool flow.

4.1 Architecture Generation

The high-level architecture generator is the first phase of creating a custom reconfigurable device, and was developed by Katherine Compton of Northwestern University [7, 8, 51]. The generator will receive, as input from the designer, the target algorithms and any associated constraints, such as area or performance. The high-level architecture generator will then create a Verilog representation of the architecture that meets the designer's requirements. The architecture generator creates a range of different architectures, from FPGA-like to ASIC-like, depending upon how much flexibility is needed. If the specified design consists of many diverse algorithms, the final architecture will contain more flexibility, resembling a traditional FPGA. Conversely, if the design only needs to support a few very similar algorithms, the final architecture will have limited flexibility, resembling an ASIC.

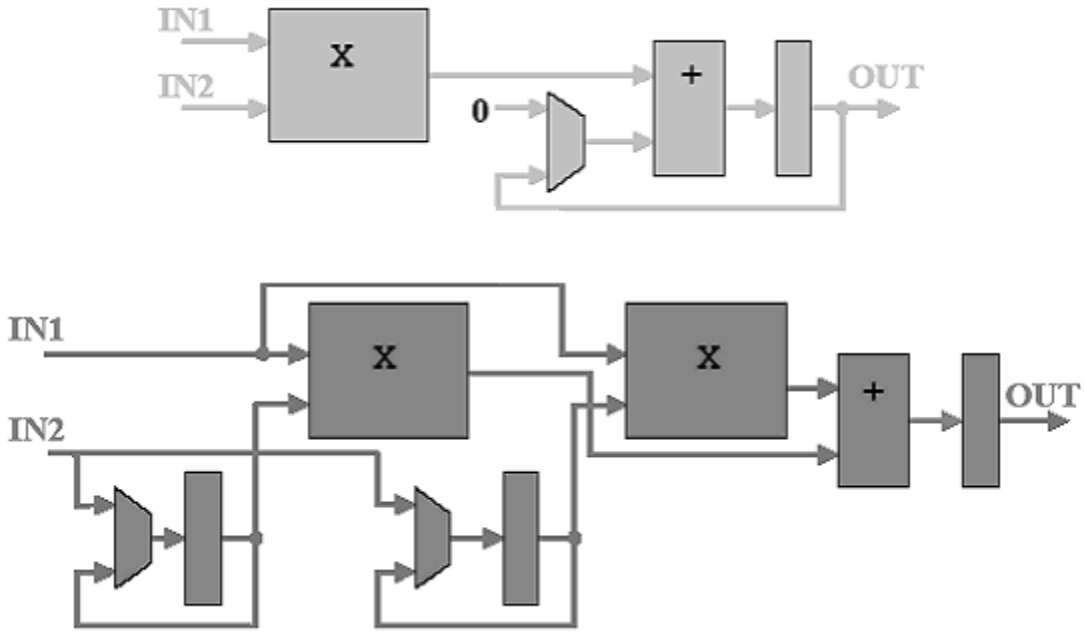


Figure 4-2: Netlist 0, the top and light netlist, and Netlist 1, the dark and bottom netlist, are two netlists that the custom architecture is required to support. Netlist 0 is a multiply-accumulate, while Netlist 1 is a two tap FIR filter [51].

4.1.1 Logic Generation

The architecture generator creates a range of custom reconfigurable devices, ranging from devices that have limited flexibility and devices that are RaPiD-like in nature with more flexibility. The method used to generate the logical units for both of these types of devices is basically the same, while the generation of the routing fabric is different. Therefore, the methodologies discussed in this section on logic generation can be applied to the creation of either type of device.

The first step in generating custom reconfigurable hardware is the creation of the set of target netlists. Towards this end, the netlists are synthesized by the RaPiD compiler from a C-like Hardware Design Language (HDL), namely RaPiD-C [40]. Figure 4-2 is a block

diagram representation of two netlists, which will be used as an example throughout this section.

Once the netlists are created, they are fed into the architecture generator, which is able to establish the number and type of functional units that are needed to support the specified application domain. One of the main goals in this step of architectural generation is the maximization of resource sharing, which in turn minimizes the number of functional units and routing resources needed to represent the application domain. Designers still have the option of adding more flexibility to devices by increasing the number and type of functional units instead of using the minimum number needed to support the application domain.

Currently all of the netlists in the application domain must be able to run in their entirety without any type of rescheduling. This is a big drawback with the current version of the Totem Project, since it can lead to larger and slower architectures, however, this will be addressed in future versions. In essence, the current version of the Totem Project does not optimize for any particular netlist in the application. Therefore, it is possible for one netlist to dominate all of the other netlists in an application domain. This is a potential problem, and the ability to reschedule netlists and optimize for a particular netlist may offer a viable solution.

The absence of rescheduling implies that the minimum number of a particular type of functional unit needed in silicon is the maximum number needed by any netlist in the application domain. For example, if an application domain is represented by two netlists

A and B, and netlist A needs 10 multipliers and 15 data-registers, and netlist B needs 5 multipliers and 30 data-registers, then the generated architecture will consist of 10 multipliers and 30 data-registers. Once the type and number of functional units is established, they then must be placed.

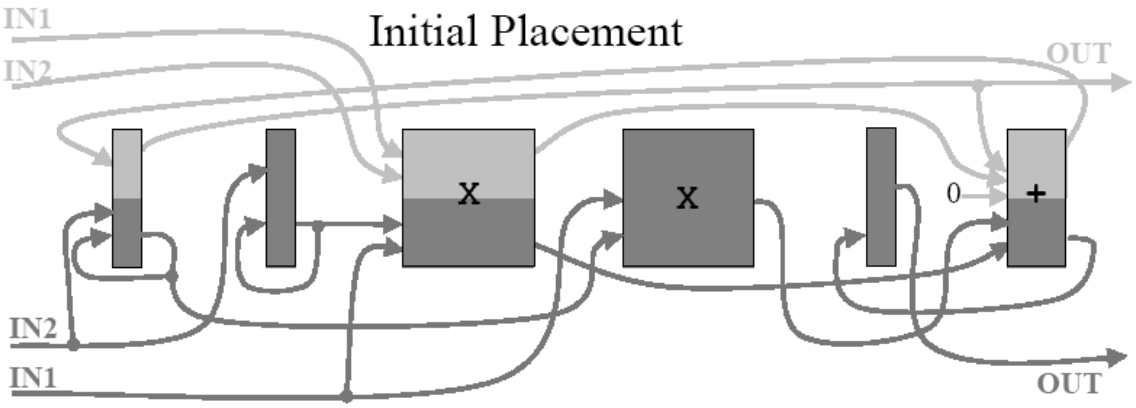


Figure 4-3: An initial placement of both the physical units and netlist bindings of the two netlists shown in Figure 4-2. When Netlist 0 is in operation only the light wires and components are used. When Netlist 1 is in operation only the dark wires and components are used. Any components that are both light and dark are shared [51].

The final step in the logic generation is the placement and ordering of the functional units along the x-axis. The placement of the components needed to support the application domain is performed by simulated annealing [12]. Simulated annealing is an algorithm that is commonly used in the placement of cells in the creation of standard cell designs, and when binding a netlist to the physical components of an FPGA (the placement half of FPGA place-and-route). The simulated annealing algorithm is analogous to how a metal cools and freezes into a minimum energy crystalline structure. Thus, the algorithm starts with a random initial placement and a high temperature. As the temperature cools, the algorithm swaps the locations of a large number of randomly selected elements, accepting moves if they improve the overall cost of the placement, which is shown in

Figure 4-3 and Figure 4-5. The algorithm will also occasionally accept a "bad" move that results in a higher overall placement cost, with the intention of avoiding being trapped in local minima. The temperature of the annealing algorithm determines the probability of accepting these bad moves, with higher temperatures having a higher probability of accepting a bad move. As the algorithm progresses the probability of accepting a bad move is gradually reduced until no bad moves are accepted.

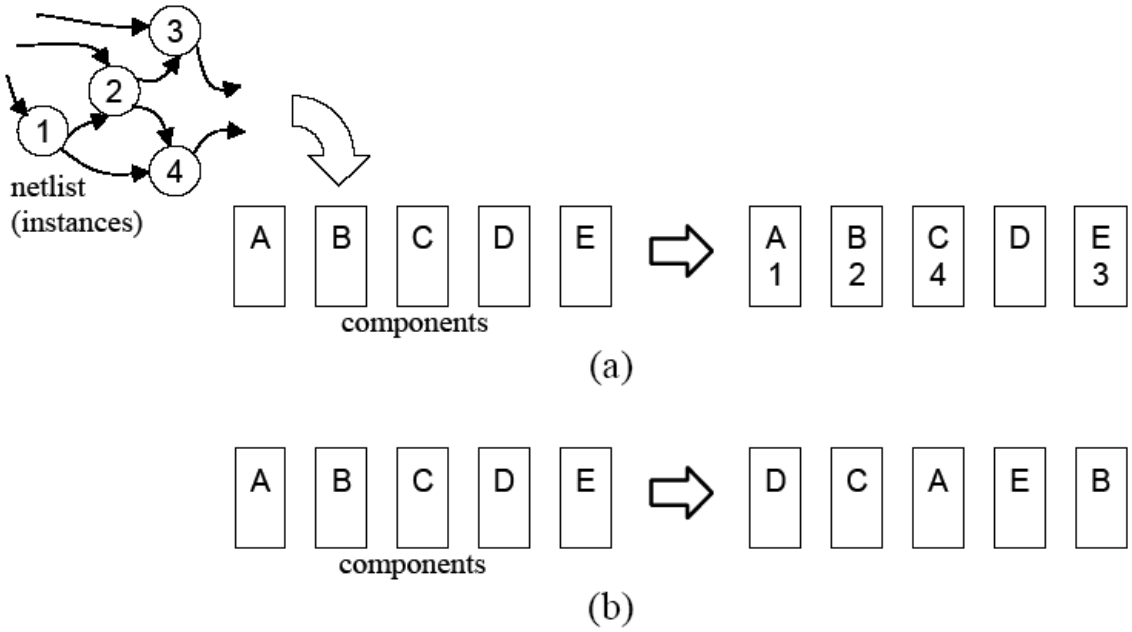


Figure 4-4: Both (a) netlist binding and (b) physical placement are shown. During netlist binding, netlist instances are assigned to a physical unit. During physical placement, the physical units are reordered [51].

The simulated annealing algorithm that is used by the architecture generator has a move set of both physical units, similar to standard cell placement, and also netlist rebinding, similar to FPGA netlist mapping. An example of each is shown in Figure 4-4. It should be noted that the simulated annealing algorithm needs to be able to perform both netlist binding and physical placement simultaneously. This is because the locations of the

physical units must be known in order to find the best netlist binding, and the netlist binding must be known in order to find the best physical placement.

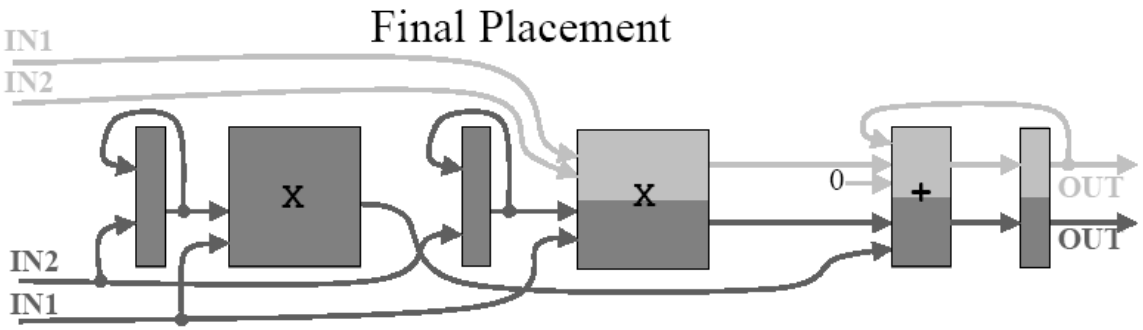


Figure 4-5: A possible final placement of Netlist 0 and Netlist 1 from Figure 4-2. The light colored wires are Netlist 0's signals, while the dark colored wires are Netlist 1's signals. Instance names from Figure 4-2 are inside the boxes in italics. The light colored components are Netlist 0's, while the dark colored components are Netlist 1's. Components that are both light and dark are shared between the two netlists. Since routing has not yet occurred, physical wires have not been created [51].

4.1.2 Configurable ASICs and Flexible Architectures

The Totem Project is capable of creating a range of architectures, depending upon how much flexibility is needed to support the application domain. This range of architectures can be covered by two different types of generated architectures, namely configurable ASICs and flexible architectures. Several heuristics have been explored to generate the routing structure for both of these types of architectures. The main difference between the configurable ASICs and the flexible architectures is how much additional flexibility is provided. Configurable ASICs have the minimal amount of resources needed to support the application domain. By reducing as much overhead as possible, configurable ASICs will be smaller and perform better than devices with more flexibility. This also implies that configurable ASICs are not well suited to handle additional netlists in the future. In

these types of circumstances, where the application domain may grow in the future, flexible architectures are better suited. As mentioned above, the logic generation is the same for both methods, with the caveat that designers are able to add additional functional units when creating flexible architectures.

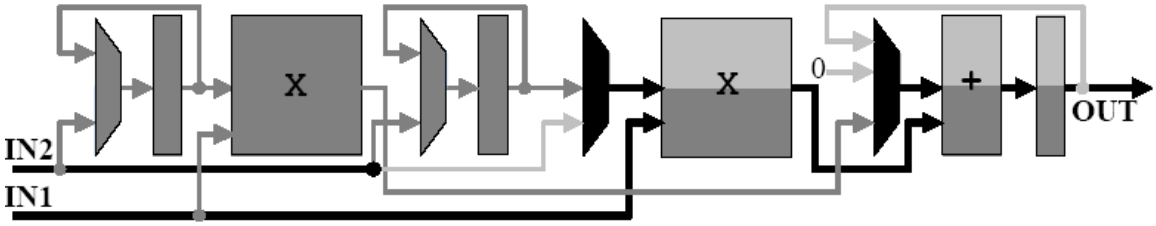


Figure 4-6: Configurable ASIC routing created for the two netlists from Figure 4-2. The light components and wires are used by Netlist 0, while the dark components and wires are used by Netlist 1. Components that are both light and dark are shared between the two netlists. Black wires and black muxes are used by both netlists [51].

The creation of the routing structure for configurable ASICs is performed by three different heuristics, with the common goal of maximizing wire sharing. The first heuristic is based upon a greedy approach, the second heuristic is based upon a recursive form of maximum weight bipartite matching, and the third heuristic is based upon a graph-based algorithm called clique partitioning. A configurable ASIC routing fabric for the netlists from Figure 4-2 is shown in Figure 4-6. All three methods allow the designer to add an additional level of flexibility above the bare minimum needed to support the application domain. The level of additional flexibility can be used for such things as bug-fixes or modifications to the number and types of algorithms in the application domain. The designer can also add even more flexibility if the application domain requires it.

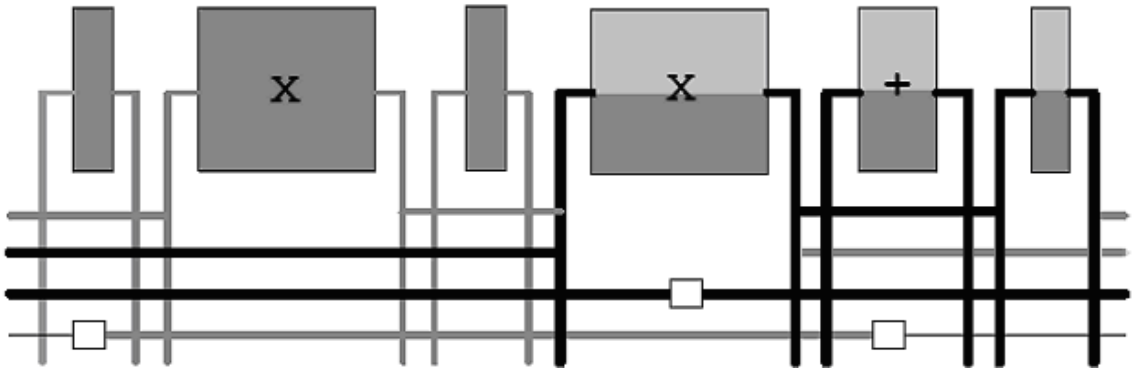


Figure 4-7: Flexible routing created for the two netlists from Figure 4-2. The vertical lines to the left and the right of a functional unit are muxes and demuxes on the input and output ports of the functional unit. The white boxes are bus connectors as seen in the RaPiD section of Chapter 2. The light functional units are used by Netlist 0, and the dark functional units are used by Netlist 1. Black lines are shared resources, as are functional units that are both light and dark.

The first heuristic that is used to create the routing for flexible architectures is called Greedy Histogram, and its goal is the reduction in the overall number of routing tracks. It also tries to use local tracks over long tracks, thus reducing the number of bus connectors needed. The connectivity of the routing structures created by this heuristic may be non-uniform along the x-axis, with regions of high connectivity interspersed with regions of low connectivity, characterized by an abundance or minimal amount of bus connectors respectively. This effect occurs because the wire length for a "new" track is chosen independently of other tracks, which leads to an uneven distribution of wire breaks and bus connectors. This characteristic may reduce the overall flexibility of the architectures that are created by this heuristic.

The second and third heuristics that create the routing structures for flexible architectures, namely Add Max Once (AMO) and Add Min Loop (AML), both try to provide more regular routing structures than the Greedy Histogram algorithm. The routing fabric

created by these algorithms is very uniform, and the level of connectivity at any point along the x-axis is very similar to all other points. To create structures that have regular routing, both of these algorithms try to evenly distribute breaks and bus connectors within each track and across all tracks. This is performed by choosing track offsets that provide a somewhat consistent level of connectivity regardless of the location within the array. To help facilitate regularity in the routing structures created, these algorithms restrict the length of wires to a power of two. Thus, local tracks can be 0, 2, or 4 units in length, while long tracks can be 8 or 16 units in length. By providing regularity in the routing, AMO and AML can increase the overall flexibility of the reconfigurable device, as compared to the Greedy Histogram method.

The difference between the AMO and AML algorithms lies in their emphasis on the use of either long distance or local routing tracks. The AMO algorithm tends to weight towards the use of distance routing tracks. This is because AMO only considers each routing length and type once, in the following order: length 0 feedback tracks, length 2 local tracks, length 4 local tracks, and length 8 distance tracks. The AML algorithm, by contrast, tends to weight towards local tracks over distance tracks. This occurs because the AML algorithm cycles through all types and lengths of tracks each time a signal is routed. It cycles through tracks in the following order: length 0 feedback tracks, length 2 local tracks, length 4 local tracks, length 16 distance tracks, and length 8 distance tracks. (Length 8 distance tracks are considered more expensive than length 16 distance tracks

because bus-connectors are expensive, and length 8 distance tracks require more bus-connectors.)

The final step involved in architecture generation is the creation of a Verilog representation of the final circuit. This Verilog is then sent to both the VLSI layout generator and the place-and-route tool generator, as is shown in Figure 4-1. Both of these generators are detailed in the next sections of this chapter. In this brief outline of the architecture generator, several features and details have been left out, since they are beyond the scope of this dissertation. Please see [51] for further information.

4.2 VLSI Layout Generation

The VLSI Layout Generator automates the creation of mask ready layouts from the circuits provided by the Architecture Generator. The layout generator must be able to create layouts for any conceivable circuit that the high-level architecture generator is capable of producing, without squandering any gains that have been achieved thus far. We have investigated three possible methods of automating the layout process: the Template Reduction Method, the Circuit Generator Method, and the Standard Cell Method [5, 9]. This thesis is concerned with this aspect of the Totem Project, and each of these methods will be discussed in Chapters 6, 7, and 8 respectively.

4.3 Place-and-Route Tool Generation

The Place-and-Route (P&R) Tool Generator creates tools that will enable the designer to utilize the new architecture, and was developed by Akshay Sharma of the University of Washington [6, 10]. The P&R Tool Generator must not only be flexible enough to support a wide array of possible architectures, but it also must be efficient enough to take advantage of diverse architectural features. This is due to the fact that Totem is able to generate very ASIC like to very RaPiD-like structures. The P&R Tool Generator creates mapping tools by using the Verilog provided by the high-level architecture generator, a description of the RaPiD netlists that need to be placed and routed onto the architecture, and the configuration bit-stream format provided by the layout generator.

Placement and routing, as referred to in this section, are the binding of netlist components onto the physical structure that has been created by the architecture generator. In essence, this type of P&R is a version of the typical FPGA placement and routing, and is referred to as netlist binding in section 4.1. As the name suggests, P&R tool generation can be broken into two distinct phases, namely the placement and the routing of netlists onto the physical architecture.

4.3.1 Placement

Simulated annealing [12], described in more detail in section 4.1, is used to determine a high-quality placement. The cooling schedule used is based upon the cooling schedule devised for the VPR tool-suite [13].

The choice of the placement cost-function is based upon the fact that the number of routing tracks in RaPiD is fixed. Therefore, the cost-function is based upon the number of signals that need to be routed across a vertical partition of the datapath for a given placement of datapath elements, which is referred to as the cutsize. The cost-function used is

$$cost = w*max\_cutsize + (1-w)*avg\_cutsize,$$

where $0 \leq w \leq 1$, *max_cutsize* is the maximum cutsize for any vertical partition, and *avg_cutsize* is the average cutsize across all vertical partitions. Through empirical analysis, it was shown that setting $w = 0.3$ yielded the best placements [6], leading to the following cost function:

$$cost = 0.3 * max\_cutsize + 0.7*avg\_cutsize.$$

4.3.2 Routing

Once placement of the functional units of a netlist has been performed, the next step is to route all of the signals between the functional units. A version of the Pathfinder [14] algorithm is used to route signals after the completion of the placement phase. There are two main parts to the routing algorithm, a signal router and a global router. The signal router is used to route individual signals, using Prim's algorithm [15]. The global router is used to adjust the cost of using each routing resource at the end of a routing iteration.

The cost of using a routing resource is based upon the number of signals that share that resource. The cost function used by the router for a particular node *n* is,

$$Cn = (Bn + Hn) * Pn,$$

Where *Bn* is the initial cost of using the node, *Hn* is a cost term that is related to the historical use of the resource, and *Pn* is the number of signals that are sharing that node during the current iteration [6]. To facilitate initial sharing of resources, *Hn* is initially set to zero and *Pn* is initially set to one. As the algorithm progresses, the value of *Pn* is slowly increased each iteration, depending upon the use of the node. The value of *Hn* is also slowly increased, depending upon the past use of the node. In essence, the initial cost of routing resources is low, which has the effect of encouraging signals to share units. But during subsequent iterations the cost steadily climbs, which has the effect of forcing signals to competitively negotiate for routing resources. This insures that resources are used by signals that have no other low-cost alternative. For a more detailed explanation of the Pathfinder algorithm, please refer to [14]. In this brief outline of the P&R tool generator, several features and details have been left out, since they are beyond the scope of this dissertation. Please refer to [6] for further information.

Chapter 5

Research Framework

In this initial version of the Totem Project, we are using the reconfigurable-pipelined datapath (RaPiD) architectural style as a starting point for the circuits that we will be generating [49], as mentioned in Chapter 2. RaPiD affords us with a ready source of full custom layouts that have been created by Carl Ebeling's group at the University of Washington. It also enables us to leverage the RaPiD compiler and all of the netlists that have been created for use on RaPiD architectures. For more information on the RaPiD architectural style please refer to Chapter 2 of this work, or [49].

The VLSI Layout Generator automates the creation of mask ready layouts from the circuits provided by the Architecture Generator. The layout generator must be able to create layouts for any conceivable circuit that the high-level architecture generator is capable of producing, without squandering any possible area gains that have been achieved thus far. We have investigated three possible methods of automating the layout process: standard cell generation [5, 9], circuit generators, and template reduction [63], which will be detailed in later chapters. This chapter is concerned with the methodology used to compare all three methods to each other and to the corresponding full custom RaPiD circuits.

5.1 Testing Framework

To evaluate the automatic generation of domain specific reconfigurable circuits we are using thirteen different application domains. All of the netlist sets that make up each application domain have been compiled using the RaPiD compiler [40]. Two of the netlist sets, RADAR and Image, are actual application domains. The RADAR application is used to observe the atmosphere using FM signals, while the Image application is a minimal image processing library. The other eleven applications represent the cross product of two domains, like the Image and RADAR application, domains of similar netlists, like FIR, Matrix Multiply, and Sorters, or reduced domains, like Reduced Image 1 through 4 and Reduced RADAR 4 through 6.

All of the application domains and their member netlists are shown in Table 5-1. In addition, Tables 5-2 and 5-3 break down the functional unit utilization for each application domain or each netlist respectively. Also, Table 5-4 is a combination of Tables 5-1 and 5-3. It is important to note hat an application domain is any set of netlists that a designer needs to support. Therefore, there is no compelling reason why the netlists within an application domain must be similar in type or function. For example, the functional unit utilization of each netlist in the FIR and Sorter application domains are very similar to each other within their respective domains, but dissimilar to each other across domains. (As seen in Table 5-2, none of the FIR netlists utilizes any embedded memories, while none of the Sorter netlists utilizes any multipliers) However, this fact

should not stop a designer from specifying a new domain that contains netlists from both the FIR and Sorter application domains.

Table 5-1: The benchmark application domains and their corresponding member netlists used to evaluate the Template Reduction, the Circuit Generator, and the Standard Cell Method, along with the full custom RaPiD II tile. The applications are ordered in the table by their percent utilization, from lower to higher values.

Application Domain	Member Netlist	Percent Utilization
Reduced RADAR 6	decnsr, psd	20.92
FIR	firsm2_unr, firsm3_unr, firsm4_unr, firsymeven	28.90
Reduced Image 1	firtm_2nd, matmult_unr	29.07
Reduced Image 2	1d_dct40, fft16_2nd, matmult_unr	29.15
Sorters	sort_g, sort_rb, sort_2d_g, sort_2d_rb	32.12
Image	1d_dct40, firtm_2nd, fft16_2nd, matmult_unr	37.05
Matrix Multiply	limited_unr, matmult_unr, matmult4_unr, vector_unr	37.43
Image and RADAR	1d_dct40, fft16_2nd, firtm_2nd, matmult_unr	41.21
Reduced RADAR 4	decnsr, fft16_2nd	50.88
RADAR	decnsr, fft16_2nd , psd	52.79
Reduced Image 4	1d_dct40, fft16_2nd	52.82
Reduced RADAR 5	fft16_2nd, psd	53.54
Reduced Image 3	1d_dct40, fft16_2nd, firtm_2nd	60.18

Table 5-2: The benchmark application domains and the number of RaPiD II cells needed to implement them. The type and number of functional units needed for each application domain is also listed. The applications are ordered in the table by their percent utilization, from lower to higher values.

Application Domain	Number of Cells	ALUs	Mults	RAMs	Data Registers	Percent Utilization
Reduced RADAR 6	5	18	5	0	5	20.92
FIR	20	35	18	0	85	28.90
Reduced Image 1	30	53	18	53	66	29.07
Reduced Image 2	30	53	18	53	66	29.15
Sorters	15	32	0	18	75	32.12
Image	30	53	18	53	66	37.05
Matrix Multiply	30	53	18	53	84	37.43
Image and RADAR	30	53	18	53	66	41.21
Reduced RADAR 4	7	25	14	14	28	50.88
RADAR	7	25	14	14	28	52.79
Reduced Image 4	7	25	14	14	28	52.82
Reduced RADAR 5	7	25	14	14	28	53.54
Reduced Image 3	7	25	14	14	28	60.18

Table 5-3: The various netlists from the application domains, and the number and type of RaPiD II functional units needed to support them.

Netlist	ALUs	Mults	RAMs	Data Registers
1d_dct40	4	3	0	4
decnsr	6	0	0	4
fft16_2nd	22	12	12	25
firsm2_unr	15	16	0	45
firsm3_unr	16	16	0	60
firsm4_unr	15	16	0	45
firsymeven	31	16	0	77
firtm_2nd	7	4	8	19
limited_unr	4	4	4	10
matmult4_unr	32	16	48	76
matmult_unr	48	16	48	60
psd	16	4	0	4
sort_g	29	0	16	62
sort_rb	29	0	9	68
sort_2d_g	22	0	12	45
sort_2d_rb	22	0	8	47
vector_unr	1	1	1	1

Table 5-4: This table is a combination of Tables 5-1 and 5-3. It lists the application domain, followed by the percent utilization, and a breakdown of the usage of the ALUs, multipliers, RAMs, and data-registers for the netlists in each application domain.

Application Domain	Percent Utilization	Member Netlist	ALUs	Mults	RAMs	Data Registers
Reduced RADAR 6	20.92	decnsr	6	0	0	4
		psd	16	4	0	4
FIR	28.9	firsm2_unr,	15	16	0	45
		firsm3_unr,	16	16	0	60
		firsm4_unr,	15	16	0	45
		firsymeven	31	16	0	77
Reduced Image 1	29.07	firtm_2nd,	7	4	8	19
		matmult_unr	48	16	48	60
Reduced Image 2	29.15	1d_dct40,	4	3	0	4
		fft16_2nd,	22	12	12	25
		matmult_unr	48	16	48	60

Table 5-4: Continued.

Application	Percent Utilization	Member Netlist	ALUs	Mults	RAMs	Data Registers
Sorters	32.12	sort_g,	29	0	16	62
		sort_rb,	29	0	9	68
		sort_2d_g,	22	0	12	45
		sort_2d_rb	22	0	8	47
Image	37.05	1d_dct40,	4	3	0	4
		firtm_2nd,	7	4	8	19
		fft16_2nd,	22	12	12	25
		matmult_unr	48	16	48	60
Matrix Multiply	37.43	limited_unr, ,	4	4	4	10
		matmult_unr,	48	16	48	60
		matmult4_unr	32	16	48	76
		vector_unr	1	1	1	1
Image and RADAR	41.21	1d_dct40,	4	3	0	4
		fft16_2nd,	22	12	12	25
		firtm_2nd,	7	4	8	19
		matmult_unr	48	16	48	60
Reduced RADAR 4	50.88	decnsr,	6	0	0	4
		fft16_2nd	22	12	12	25
RADAR	52.79	decnsr,	6	0	0	4
		fft16_2nd ,	22	12	12	25
		psd	16	4	0	4
Reduced Image 4	52.82	1d_dct40,	4	3	0	4
		fft16_2nd	22	12	12	25
Reduced RADAR 5	53.54	fft16_2nd,	22	12	12	25
		psd	16	4	0	4
Reduced Image 3	60.18	1d_dct40,	4	3	0	4
		fft16_2nd,	22	12	12	25
		firtm_2nd	7	4	8	19

5.1.1 Percent Utilization

The netlist in Table 5-1 are ordered by their percent utilization, which can be more clearly seen in Figure 5-1. Percent utilization is a measure of the resources that an array of full custom fixed tiles would need to support a particular application domain. Resources include multipliers, ALUs, wires, bus connectors, routing muxes and demuxes, data and pipeline registers, and memories. For example, an application domain that requires half of the resources provided by the full custom fixed tile would fall at 50% utilization. The percent utilization calculated in Table 5-1 was generated using the RaPiD II fixed tile, which will be discussed in more detail below.

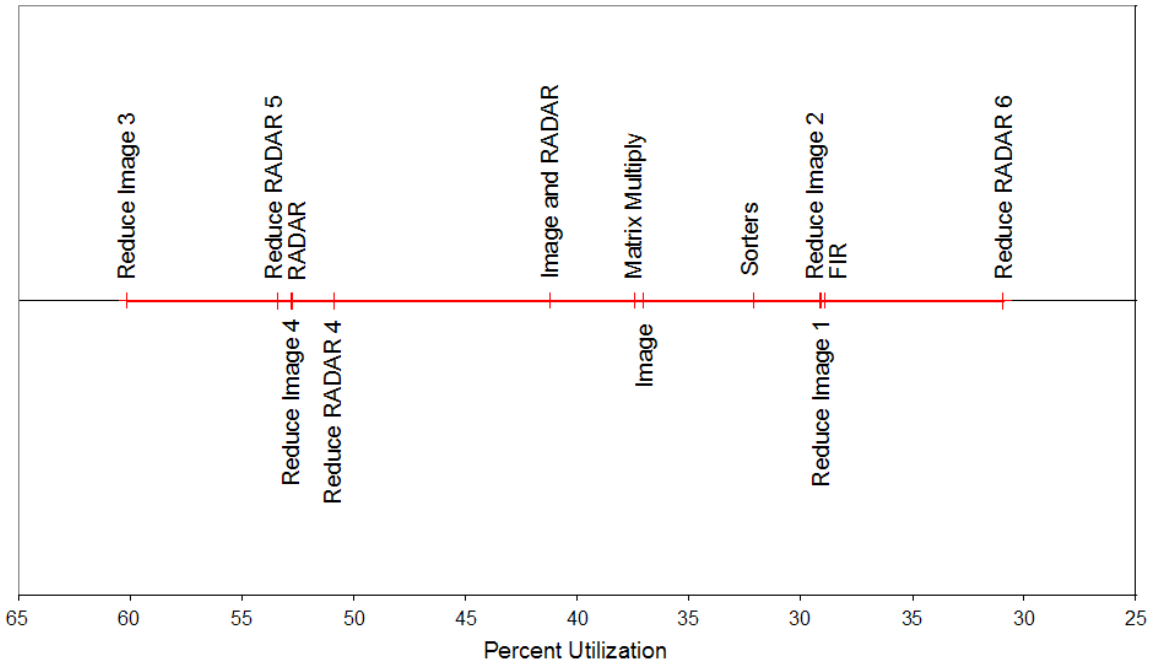


Figure 5-1: The thirteen application domains ordered along the horizontal axis by percent utilization. Application domains with higher percent utilizations are to the left, while application domains with lower percent utilizations are to the right.

To actually calculate the percent utilization we use the place-and-route tool to map the application domain onto an array of RaPiD II tiles. The length of the RaPiD II array is determined by iteratively adding another fixed RaPiD II tile to the array until the mapping is successful. If the size of the array becomes very large and the application domain still fails to map, we determine that the application domain fails, and will not map onto the resource mixed provided by the fixed tile. At this point, if possible, a different fixed tile with a different routing and functional unit mix should be tried.

Once the array length is set, we look at all of the resources that are used by the application domain mapping. In essence, if only one of the netlists in an application domain uses any resource in the array, then that resource is part of the percent utilization for that application domain. We divide the sum of the area of all of the resources needed to support an application domain by the total area of the RaPiD II array to arrive at the value of the percent utilization for an application on a particular array of fixed tiles. It should be noted that the percent utilization is entirely dependent on the particular fixed tile that is used, and if the fixed tile were to change then the percent utilization for a set of applications could vary wildly. It is conceivable that application domains could move from very high percent utilization to very low percent utilization and vice versa, based solely on the choice of fixed tiles.

In essence, the percent utilization metric is a measure of how well a fixed tile is tuned to a particular application domain. If the percent utilization of an application domain is very high, then the resource mix of the fixed tile is well suited for that application

domain. The converse is also true, with a low percent utilization indicating that a fixed tile has many resources that are not needed by a particular application domain. This metric can help designers to tune fixed tiles to particular application domains. However, at the time when the composition of the RaPiD II fixed tile was determined, we did not have this metric available to us. Therefore, the only design consideration that we pursued was the ability of the RaPiD II tile to support as many application domains as possible. Therefore, we did not optimize the RaPiD II tile for one application domain over another, or even try to make sure that median value of all application domains had a high percent utilization.

Finally, the percent utilization metric enables us to evaluate how all three methods perform compared with each other and with the full custom fixed tile. By comparing each method to the fixed tile, we are then able to relate all three methods to each other, using percent utilization. It should be noted that the percent utilization metric is biased towards the Template Reduction Method. This is because to find the percent utilization we use the reduction lists that were created by the Template Reduction Method for each application set. The reduction lists were evaluated to find the total area removed from the template. In essence, the percent utilization metric is a measure of the Template Reduction Method with perfect compaction. While the percent utilization metric is not perfect, we feel that it is adequate for our purposes.

5.1.2 RaPiD II Tile

The RaPiD I tile, shown in Figure 2-10, was used as a starting point for the creation of the RaPiD II tile. While the RaPiD I tile is a template in its own right, the new RaPiD II template was required, since we found that the original RaPiD-I tile does not have enough interconnect resources to handle some of the benchmarks intended for the architecture [38].

The RaPiD II tile, shown in Figure 5-2 and Figure 5-3, was created to support as many application domains as possible. Various application domains, including those detailed above, were run on the RaPiD I architecture, with some of the domains failing. Resources were then slowly added to the RaPiD I tile until most of the domains could be placed and routed. The following is a comparison of the RaPiD I and RaPiD II tiles:

- RaPiD I has 14 busses, RaPiD II has 24 busses
- In RaPiD I, all BC, pipeline registers, and data registers have up to three delays, in RaPiD II all of these units only have one delay
- In RaPiD I, each functional unit has up to three delays at the output of the unit, in RaPiD II there are none
- In one RaPiD I cell, there are 3 ALUs, 1 Mult, 3 RAMs, and 6 data registers, along with 10 long lines with one BC
- RaPiD II has 5 ALUs, 3 Mults, 2 RAMs, and 6 data registers, along with 8 long lines with four BCs, and 8 long lines with 2 BCs

By using a large number of domains to create the RaPiD II tile, it was hoped that the final template would be able to support similar, but unknown domains in the future. This method of producing a tile that is capable of handling a wide range of application

domains is not unique to this work. Designers of fixed reconfigurable hardware must go through this same process when finalizing a tile design.

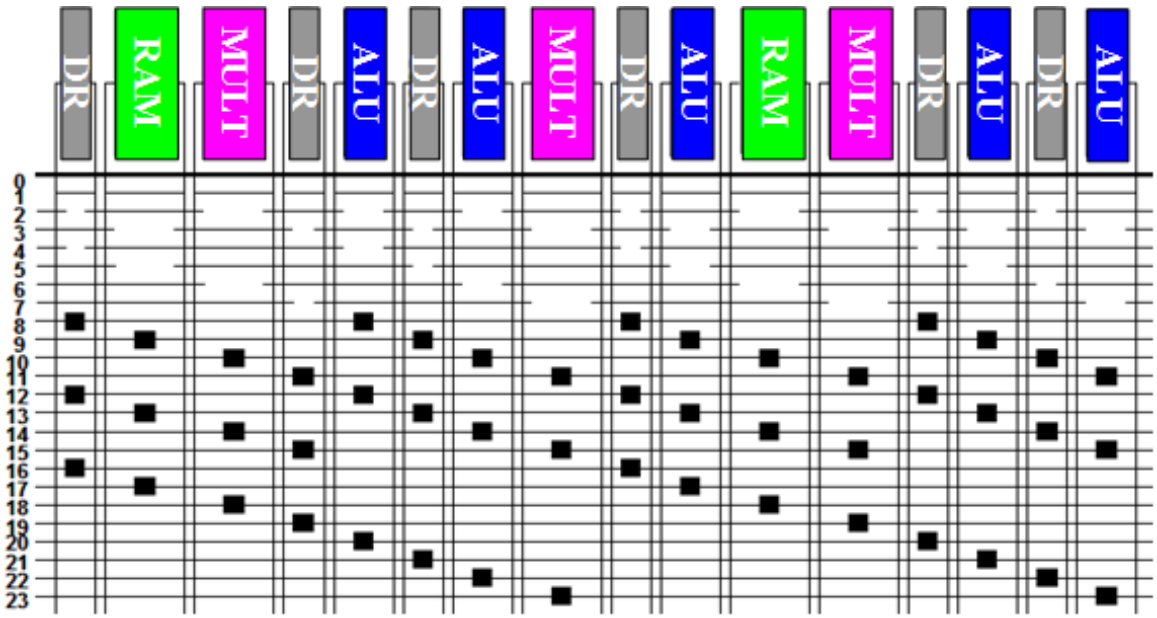


Figure 5-2: Block diagram of one feature rich RaPiD II cell. Up to thirty of these cells are placed along the horizontal axis.

While every effort was made to create a tile that supports a wide range of application domains, there are still some drawbacks associated with the RaPiD II tile. The first major drawback is the fact that the bus connectors, pipeline registers, and data registers only contain one register, while these same resources in the RaPiD I tile contain three registers. In addition, the functional units in the RaPiD II tile do not have registers on their outputs, while functional units in RaPiD I have three registers on their outputs. This severely limits the ability of the place and route tool to produce retimed mappings. Therefore, none of the mappings was retimed, leading to mappings with extremely long critical paths. The reduction of the number of registers was motivated by the fact that, at

the time when the RaPiD II tile was finalized, the Architecture Generator and the Place-and-Route Tool Generator were not able to handle more than one register.

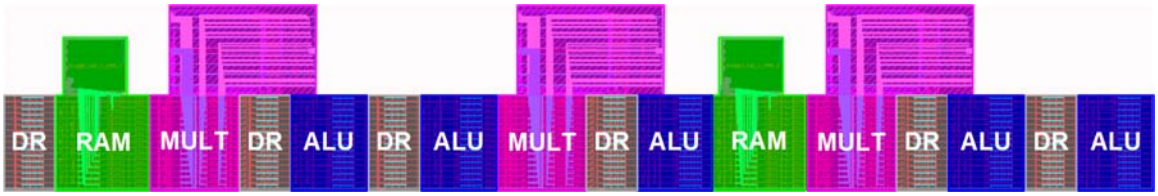


Figure 5-3: The floorplan of one feature-rich RaPiD II cell. Up to thirty of these cells are placed along the horizontal axis. Notice how the memory and the multiplier are place above the datapath.

The second drawback associated with the RaPiD II tile is a lack of routing resources. While the RaPiD II tile has 24 tracks, where the RaPiD I tile has 14 tracks, this is still not enough routing resources for certain application domains. This is borne out by the fact that the Camera and the Transforms applications still fail on the RaPiD II template. We chose 24 routing tracks for the RaPiD II tile, since this was the maximum number of routing tracks we were able to fit without increasing the height of the array when compared to the RaPiD I tile. This limitation was due to the fact that the functional units in both the RaPiD I and RaPiD II tiles were unchanged, therefore setting the height of both arrays to the same value.

5.1.3 Area and Performance Evaluation

To evaluate the three methods, we are concerned with two metrics, namely the overall area of the generated circuits, and the performance of the circuit when each of the application domains are mapped, as evaluated by the static timing analyzer contained in the place and route tool.

The area of the generated circuits is evaluated by measuring the area of the layout that is generated by each of the methods. This is a straightforward process, since all three methods generate circuits using the NCSU CDK [60] for the TSMC .18μm process. The benefit of having all three methods use the same layout process is the removal of any differences that can occur from the use of a different layout process, including differences in transistor size and layout rules.

The performance of each circuit is evaluated by using the Totem place and route tool to map, or bind, each of the netlists in the application domain onto the generated circuit. The place and route tool is then able to determine the performance of the mapped netlists on the circuit by performing static timing analysis of the critical path. The place and route tool is aware of the critical path of the netlist since it places and routes all of components and the signals that constitute a netlist. The types of delays present on the critical path can be broken into two different components, one type consisting of the various functional units and routing resources and one type consisting of the interconnect. It should be noted that the place and route tool is unable to retime signals, as mentioned in the previous section. Therefore, any performance numbers generated by the place and route tool should only be used for relative comparisons of the three methods.

To generate the performance models for all of the functional units and routing resources generated by the Standard Cell Method, we created a placement of each individual unit. We then performed spice [61] simulations on the small units like the mux, demux, and

the data and pipeline registers and Pathmill [44] simulations on the larger units like the multiplier, the ALU, and the embedded memory.

The functional units used in the Template Reduction and Circuit Generator Methods are the same, while the functional units used in the Standard Cell Method are different. It should be pointed out that we were unable to simulate the full custom layouts of the multiplier and the memory unit that were created by Carl Ebeling's RaPiD group, even though both units passed the design rule checks. In addition, the ALU also had many problems, though we were able to simulate a limited number of operations with it. We believe that these issues arose when we ported the original RaPiD I layouts from the HP .50μm process to the NCSU TSMC .18μm process. Therefore, to generate the models for these units, spice simulations were performed on the schematics, instead of the actual layouts.

The muxes and the demuxes used by both the Template Reduction and the Circuit Generator Methods are different, as is the width of each of the functional units (This is due to the differences in how each method provides input and output connections to the functional units, and this difference is accounted for in the interconnect delays). To find the mux and the demux models we ran spice simulations of these units from the extracted view of the layouts.

The interconnect delay was modeled by running spice simulations on various lengths of metal wires, with varying numbers of drivers and loads. The place and route tool is aware of how many drivers there are on a wire and how much load. It also knows the

length of each of wire, since it knows the length of each of the functional units, and it knows the path the wire takes. Therefore, the place and route tool is able to accurately model the interconnect delay of each segment between functional units on the critical path. Modeling the interconnect delay in this fashion is made much easier because we only have a one-dimensional interconnect.

Chapter 6

Template Reduction

The idea behind template reduction is to start with a full-custom layout that provides a superset of the required resources, and remove those resources that are not needed by a given domain, as shown in Figure 6-1. This is done by actually editing the layout in an automated fashion to eliminate the transistors and wires that form the unused resources, as well as replacing programmable connections with fixed connections or breaks for flexibility that is not needed. In this way, we can get most of the advantage of a full custom layout, while still optimizing towards the actual intended usage of the array. By using these techniques, the Template Reduction Method stands to obtain the benefits that full custom designs afford, while retaining the ability to remove unneeded flexibility to create further gains in both area and performance.

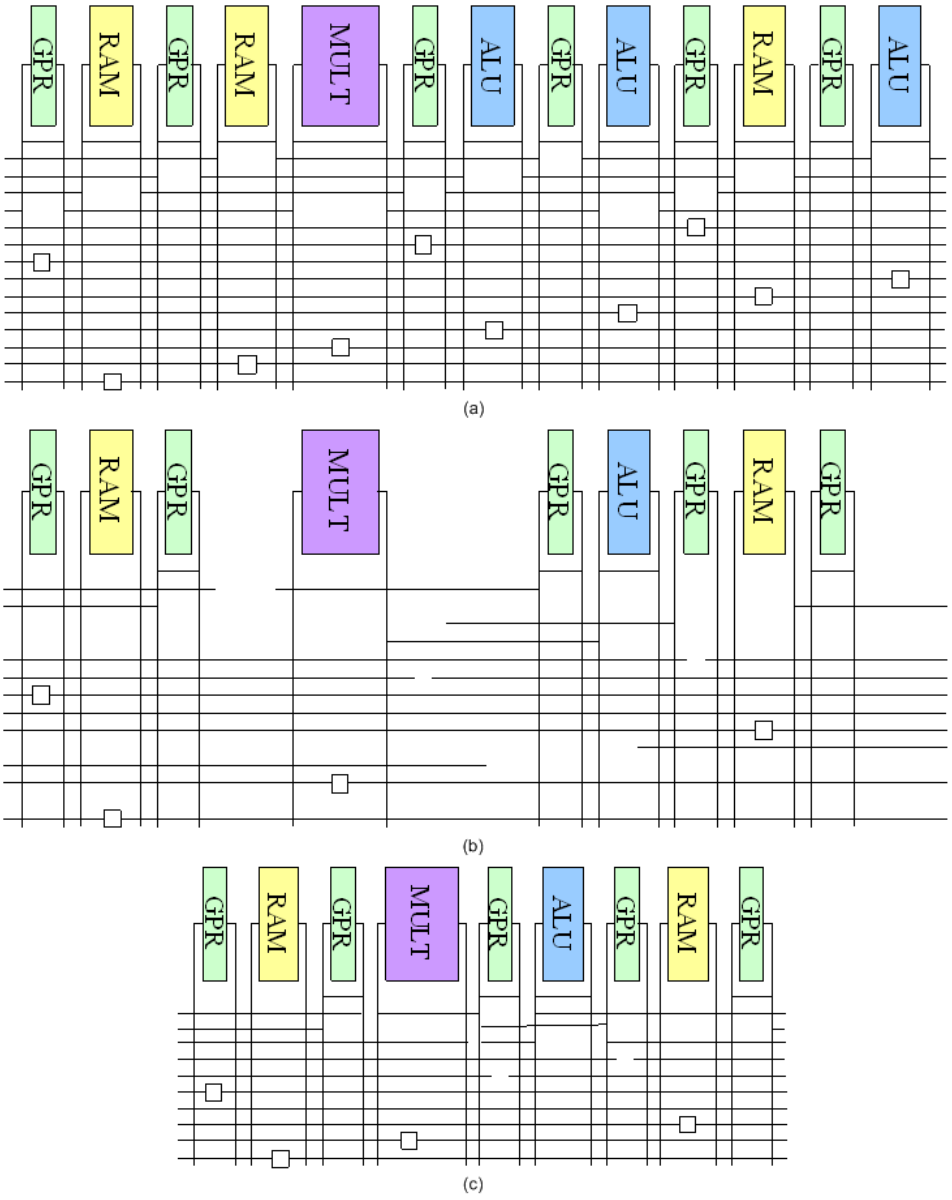


Figure 6-1: Template reduction in action. The block diagram of a feature rich macro cell is shown in (a). In figure (b), the macro cell has been reduced by the removal of routing resources and functional units that are not needed to support the application domain. Figure (c) is the final compacted cell.

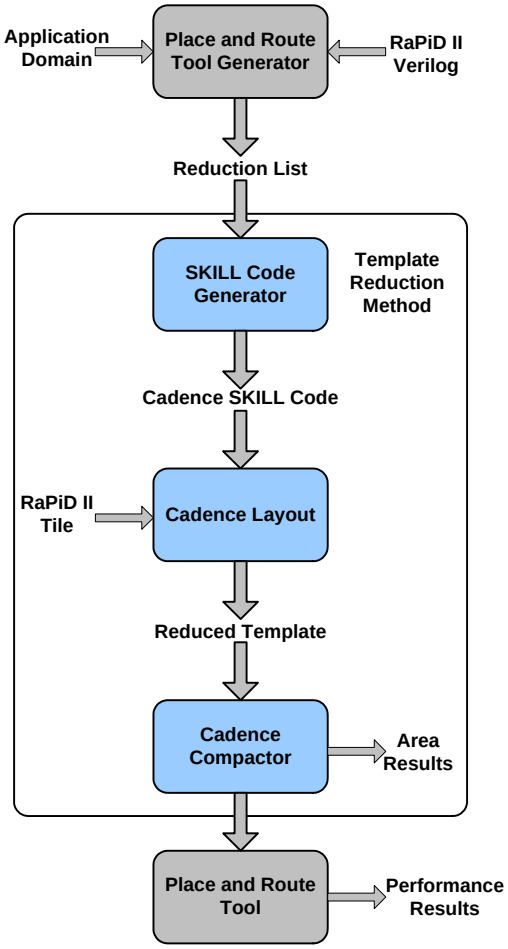


Figure 6-2: The tool flow of the Template Reduction Method. In the first step, the Place-and-Route Generator receives as input the application domain, in the form of netlists, and a verilog representation of the RaPiD II array. The P&R Tool then generates the reduction list, which is sent to the SKILL code generator. The SKILL code, generated by the SKILL code generator, is then sent to the Cadence Layout tool. The Cadence Layout tool automatically runs the SKILL code, which instructs it to perform all of the reductions on the RaPiD II tile. Next, Cadence Layout sends the reduced template to the Cadence Compactor, where the design is compacted and area results are collected. Finally, the P&R tool generates the performance numbers for the newly compacted template.

Template reduction has been broken into three main tasks. The first is the creation of a feature rich macro cell, which is used as an initial template that will be reduced and compacted to form the final circuit. The second is the creation of the reduction list that identifies the resources that should be removed. This is generated by Akshay Sharma's

place and route tool, which seeks to increase the commonality of resource usage between all of the mappings to the reconfigurable logic, and thus increase the amount of resources that can be eliminated. The final task is the implementation of the reductions on the template, followed by the compaction of the resultant circuit. This involves automated layout restructurings to edit the actual design files based upon the reduction list. Each of these tasks will be outlined in the following sections of this chapter. See Figure 6-2 for a detailed tool-flow.

6.1 Feature Rich Template

The creation of the feature rich template is the most critical aspect related to the Template Reduction Method. A poor template will not be able to support a wide range of applications, which in turn weakens the effectiveness of the method. Therefore, as discussed in Chapter 5, extensive profiling has been performed to create the RaPiD II tile, which was the feature rich template used by this iteration of the Template Reduction Method.

As discussed in Chapter 5, the RaPiD II template is not perfect, and it fails on some application domains. The failure of the RaPiD II template on these application domains highlights the difficulty in generating a template, and reinforces the idea that there is no perfect template. It also leads us to conclude that if a designer needs support for a set of application domains that diverge from the template considerably, then the Template Reduction Method has a high chance of performing poorly or failing altogether. If this

situation arises, then the designer can use a different template, if one is available, or use the Standard Cell or Circuit Generator Methods instead.

6.2 Reduction List Generation

The next task in template reduction is the creation of the reduction list, which is produced by an algorithm developed by Akshay Sharma at the University of Washington [6]. The creation of the reduction list is performed by a subtractive scheme that eliminates as many functional units and routing resources (functional units and routing resources are collectively called "resources") as possible while placing and routing a set of netlists onto the template architecture. Individual netlists are placed using a simulated annealing approach [6], and routed using the Pathfinder algorithm [14]. Initially, all netlists in the set are individually placed and routed on the template architecture. At the end of this first run, the fraction of netlists that used each resource in the template is recorded, and a cost (referred to as usage_cost) is assigned to each resource based on the fraction of netlists that used the resource during the previous run. The usage_cost of a resource is inversely proportional to the fraction of netlists that used the resource. Thus, the usage_cost of a resource that was used by none of the netlists is highest, while the usage_cost of a resource that was used by all netlists in the set is zero.

After completion of the first run on all netlists, a second run is commenced during which the netlists in the set are individually placed and routed again on the template architecture. However, for any given netlist, the cost of using a resource during the

second run is influenced by the usage_cost of that resource. During placement, assigning a logic block to a functional unit penalizes the cost of the placement by a factor proportional to the usage_cost of the functional unit. The cost of assigning a logic block to a functional unit with high usage_cost is higher than the cost of assigning the logic block to a functional unit that has a relatively lower usage_cost. Similarly, while routing a netlist, the base cost of using a routing resource is proportional to the usage_cost of that resource. In general, if the usage_cost of a resource is high (i.e. the fraction of netlists that used this resource in the previous run was low), the place-and-route tool is influenced to select another resource with a lower usage_cost (i.e. a resource that was used by a large fraction of netlists during the previous run). Thus, during the second run, we try to direct the placement and routing of individual netlists toward using resources that were used heavily during the previous run. At the same time, we also attempt to drive down the fraction of netlists that use a resource to zero, so that we can eliminate that resource eventually. At the end of the second run, the usage_cost of each resource is again adjusted in a manner identical to that at the end of the first run, and a third run is begun. We are only performing three runs, because the third run only deviates slightly from the second run in increasing the amount of resources that can be eliminated. Thus, any gains from subsequent runs are negligible, and therefore are not performed. Once the three runs are completed, we have a list of the resources that can be eliminated from the template architecture.

Equation (1) describes the variation in the usage_cost of a functional unit with the fraction of netlists that used that functional unit during the previous run.

$$usage\_cost = k1*(1 - f)^2 \quad \ldots\ldots (1)$$

The fraction of netlists that used the functional unit in the previous run is given by f. The usage_cost of all functional units that were used by more than 20% of the netlists during the previous run decreases quadratically with f. The value of k1 is chosen in a manner that ensures that the total usage_cost of a placement never exceeds 20% of the total cost of a placement.

The usage_cost of routing resources also has a negative quadratic dependence on f (Equation (2)). If none of the netlists in the set used a routing resource during the previous run (f = 0), then its usage_cost is maximum and is equal to k2. On the other hand, if all the netlists in the set used a routing resource during the previous run (f = 1), then its usage_cost is zero. The value of k2 is selected so as to ensure that the usage_cost of a routing resource never exceeds 10% of the base cost of the routing resource.

$$usage\_cost = k2*(1 - f)^2 \quad \ldots\ldots...\ldots.(2)$$

6.3 Reduction and Compaction

Once the reduction list is generated, the final task is to actually edit the template in an automated fashion, followed by a compaction step to reduce the template size. To reduce the template, the layouts were automatically edited within the Cadence CAD tools. To

achieve the required automation, Cadence SKILL code [39] is created by a SKILL code generator written in Perl. The SKILL code generator parses the reduction list and automatically creates a list of SKILL code reductions. Cadence SKILL Code enables interaction with the Cadence tools at a very low level. Therefore, each reduction that the subtractive method is able to perform has a corresponding SKILL routine that will implement the reduction on the template. In Figure 6-3 we see SKILL code that deletes all of the visible shapes contained within a specified bounding box. This type of SKILL procedure, when coupled with methods that toggle the visibility of layers, enable the Template Reduction Method to delete any shape in any of the layouts that represent the template.

```
; You need to pass in the Bbox and the windowId
procedure(TRdelete_Bbox(Bbox windowId)
   printf(
      "     TRdelete_Bbox(Bbox: %B)\n"
      Bbox
   );end printf

   unless(geGetInstHier(windowId);makes sure inst exists
      println(geGetInstHier(windowId))
      error("Something is wrong in TRdelete_Bbox!\n")
   );end unless

   if(geSelectArea(windowId Bbox)
      then
         geDeleteSelSet(windowId); performs the deletion
      else
         warn("In TRdelete_Bbox, geSelectArea is empty!\n")
   );end if
);procedure TRdelete_Bbox
```

Figure 6-3: This figure is SKILL code that instructs Cadence to delete all of the visible shapes or polygons that are within a particular bounding box (Bbox). This function is used extensively in Template Reduction to delete polygons, and is used in conjunction with functions that enable you to toggle the visibility of layers.

To remove as much overhead as possible we have implemented a wide range of reductions. First among them is the elimination of any unused cells (that is, complete RaPiD II tiles). The next reduction is the elimination of any functional units in any cell

that are not needed. Next, we remove any of the bidirectional bus-connectors that are not needed in the interconnect. The final reduction is the removal of any unused wires. When an unused wire is removed, the corresponding transistors and programming bits in any muxes and drivers that the wire interacts with are also removed. The removal of wires and their corresponding transistors in muxes and drivers may seem unnecessary, especially since the current compactor is unable to take advantage of these area gains (see below). We do this reduction with the hope that a better compactor will be able to utilize this reduction and make additional area gains. Once all of the reductions have been preformed on the RaPiD II array, the final design is compacted.

The arrays were compacted by the Cadence compactor along the horizontal axis only. The reason why the compactor was limited to only compacting along the horizontal axis was that we had difficulty in compacting along the vertical axis. In addition, the functional units themselves were not modified by the compactor. In essence, the compactor only reclaimed empty space in the array by sliding functional units together along the horizontal axis. Finally, the height of the array does not change from application group to application group during compaction. This is because we did not vary the bit width of the functional units, and because RaPiD utilizes a one-dimensional routing interconnect. It also does not vary because we only compact in the horizontal, and not the vertical direction.

6.4 Results

We used five application domains to evaluate the Template Reduction Method. We did not use all of the thirteen application domains discussed in Chapter 5. When this work was performed the additional eight application domains, which are reductions or combinations of the original five domains, did not exist. These eight domains were later created to better understand how individual netlists could dominate within an application domain. Unfortunately, after the eight additional domains were created, the Cadence Compactor was updated and was unable to work within our tool-flow. Therefore, results obtained from the additional application domains could not be produced.

All of the netlists in each application domain were compiled using the RaPiD compiler [40]. The five application domains are:

- • Radar – used to observe the atmosphere using FM signals
- • Image Processing – a minimal image processing library
- • FIR – six different FIR filters, two of which time-multiplex use of multipliers
- • Matrix Multiply – five different matrix multipliers
- • Sorters – two 1D sorting netlists and two 2D sorting netlists

Refer to Chapter 5 for a more detailed discussion of these application domains.

6.4.1 Reduction List Generation

To generate the reduction list, a subtractive method was used, as explained above. To maximize the number of possible reductions, netlists were forced to share resources as much as possible. The results of the forced sharing after each of the three runs are shown

in Figure 6-4 and Figure 6-5. Figure 6-4 shows functional unit sharing, while Figure 6-5 shows routing resource sharing. During the initial run, resource sharing is not considered since we do not yet know which resources the individual mappings will prefer to use. During the 2nd and 3rd runs, the cost to use sparsely utilized resources is made more expensive. This has the effect of forcing netlists to share units as much as possible.

As can be seen, the subtractive approach is effective in increasing the proportion of resources that can be eliminated. While on average 38% of the functional units and 60% of the routing resources are unused in the initial run, this increases to 43% of functional units and 73% of routing resources by the third run. Note that the changes from the second to third run are relatively small, indicating that further iterations are unlikely to make much improvement. Also note, the number of functional units that are shared by 2 or 3 netlists seems to stay static as we perform successive runs. Two things explain this effect. First, it appears that the initial placement seems to capture most of these units. Secondly, since the pie graph is only showing 0, 1, 2, or 3 or more netlists, we are not seeing the increase in sharing by 4, 5 etc netlists. The important thing to note is not necessarily the increase of sharing by 2, 3, 4, 5, etc. netlists, but the increase in the functional units that have 0 sharing, since these functional units are the only ones that will be eliminated by the Template Reduction Method and should be maximized as much as possible.

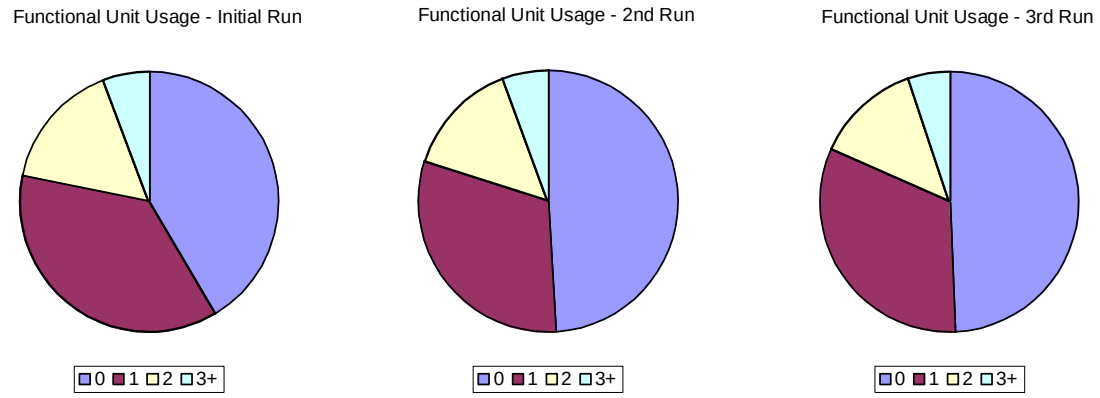


Figure 6-4: A comparison of the number of functional units used by zero, one, two, and three or more netlists.

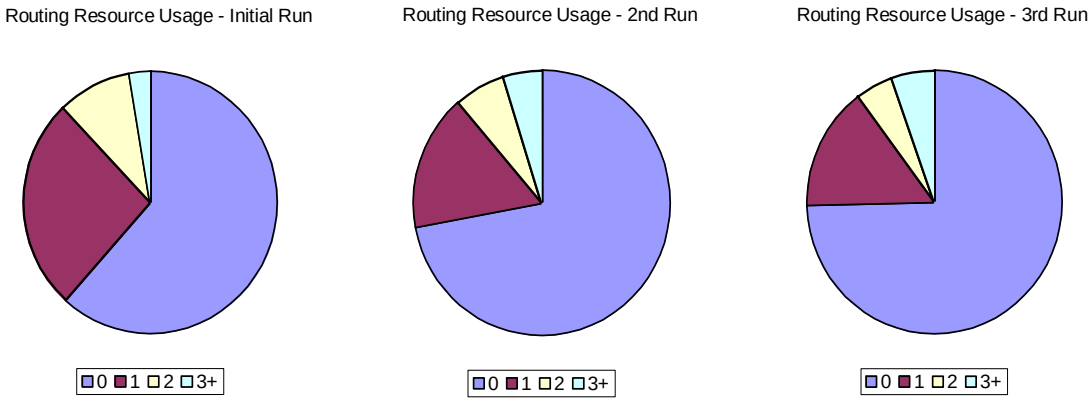


Figure 6-5: A comparison of the number of routing resources used by zero, one, two, and three or more netlists.

6.4.2 Area

In Table 6-1 we see the 5 application groups, along with the number of RaPiD II cells needed to support them. We also see the RaPiD II array length. This is followed by the percent remaining of the functional units and routing resources. The functional units include the ALUs, data registers, embedded memories, and multipliers present in the array. The routing resources include all of the long and short lines, as well as the bus-connectors that are in the array. The muxes and demuxes are not included in the table,

because they are deleted if their associated functional unit is deleted. The length of the array after compaction is then shown for each application group is shown. As stated above, this is only compaction along the horizontal axis. Finally, from Table 6-1 we see that the average percent of the template that remains is 47.6%.

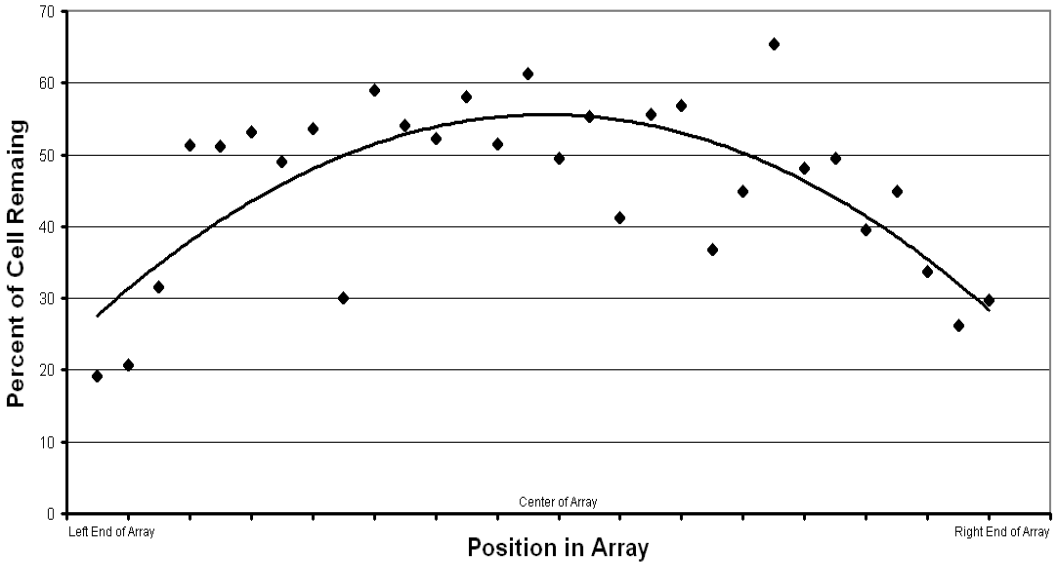
Table 6-1: This table shows the number of cells used by each application group, followed by the RaPiD Array length before compaction, the percent functional units and routing resources remaining, the RaPiD Array length after compaction, and the percent of the array that remains.

Application Group	Number of Cells	Array Length	Percent FUs Remaining	Percent Routing Remaining	Compacted Array Length	Percent of Array Remaining
FIR	20	50804.0	38	28	26663.7	52.5
Image Processing	30	76206.0	56	23	30207.9	39.6
Matrix Multiply	30	76206.0	51	28	34987.0	45.9
RADAR	7	17781.4	31	24	9968.8	56.1
Sorters	15	38103.0	46	29	16926.6	44.4
Average						47.6

While Table 6-1 shows overall trends, Graph 6-1, which was generated from the data presented in Table 6-2, shows the average reduction of each cell in the array based upon its position along the horizontal axis. The general trend that can be observed is the fact that the majority of the reduction takes place at the ends of the array. This is to be expected, since the middle of the array needs to support more communication than the ends of the array. This also leads us to the possibility of creating specific templates for the end of the arrays that do not have as much routing resources as the templates used for the middle of the array. Having different templates that more accurately reflect application domains will reduce any inefficiencies that occur when performing template reduction.

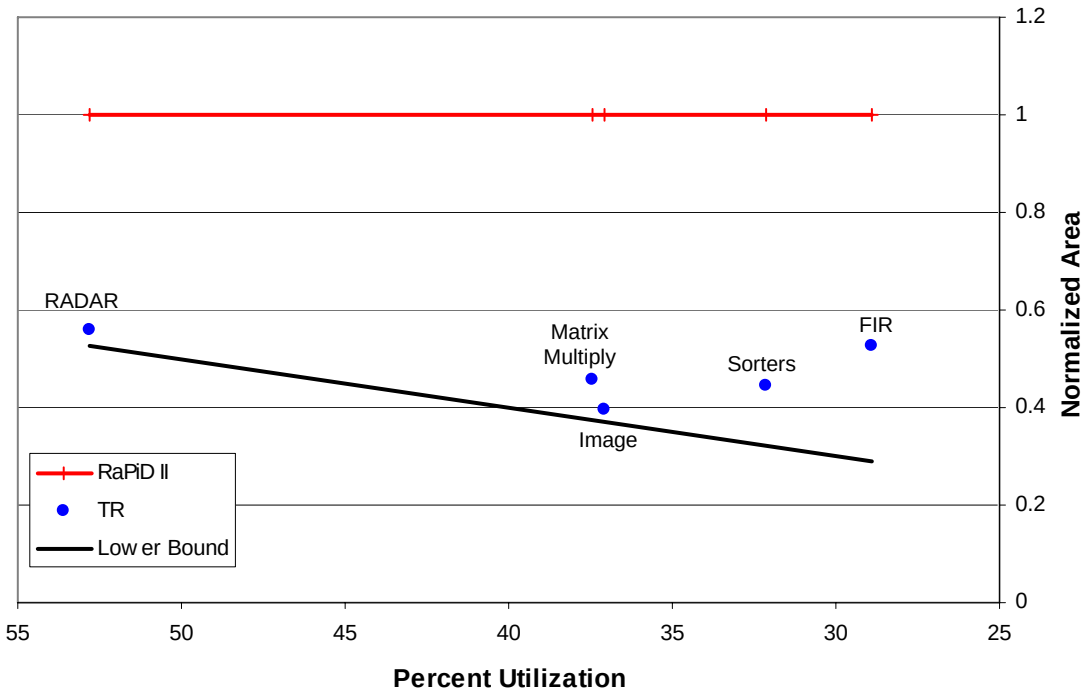
Table 6-2: This table shows the percentage of the area of each cell that remains after compaction. The final column shows the average across application domains. The application domains do not all require the same number of cells, hence the empty cells in the table.

Position in Array	Percent Remaining					
	Matrix Multiply	RADAR	Sorters	FIR	Image	Average
Left End of Array	16.5	37.2	4.66	3.50	33.9	19.2
	10.9			12.6	38.6	20.7
	44.7		4.66		45.2	31.5
	44.6			59.4	50.2	51.4
	43.7		31.1	60.9	69.2	51.2
	35.2	75.7			48.6	53.1
	47.6		55.3	81.4	11.9	49.0
	56.2			65.0	39.7	53.6
	47.9		32.0		10.3	30.1
	61.2			78.8	37.0	59.0
	52.4	40.8	54.2	73.4	49.7	54.1
	57.5				47.1	52.3
	64.7		55.9	69.3	42.3	58.1
	50.4			48.5	55.3	51.4
Middle of Array	44.6				77.9	61.2
	44.7	71.4	52.8	52.7	26.1	49.5
	44.3			74.3	47.5	55.4
	35.5		70.3		17.7	41.1
	50.7			71.0	45.2	55.6
	60.8	39.7	64.3	83.1	36.4	56.9
	36.6				37.0	36.8
	19.4		58.5	53.8	47.8	44.9
	67.8			70.5	58.1	65.5
	34.9		63.1		46.2	48.1
	50.9	81.4		27.8	38.0	49.5
	55.3		52.0	38.1	12.6	39.5
	46.2				43.4	44.8
	37.0		40.4		23.9	33.7
	40.8			21.2	16.8	26.3
Right End of Array	35.0	46.3	27.1	4.53	35.9	29.8



Graph 6-1: This graph shows the average percent of each cell remaining for each application group. The x-axis is the position in the array, with origin set at the left end of the array. The y-axis is the percent of the cell remaining.

Graph 6-2 shows the area of each benchmark set normalized to the area of the unaltered template. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile, with lower values representing smaller, and therefore more desirable, circuits. The x-axis represents the percent utilization, which is the percentage of the fixed tile resources needed to support the application domain. The RaPiD II fixed array is represented by the red line. We have decided to represent this array as a line because it reinforces the idea that regardless of the percent utilization needed to support an application domain, if this array is chosen, all of the resources are available whether or not they are used. We have also included a line representing the lower bound for the Template Reduction Method. It represents the best possible results, assuming perfect reduction and compaction.



Graph 6-2: This graph shows the normalized area of each benchmark set. The x-axis is the percentage of the resources of the fixed tile needed to support the benchmark set. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile. The lower bound on the best possible reductions. Three points on the line are (100, 1), (50, .5), and (0, 0).

From Graph 6-2 we see that the five benchmark sets fall in a narrow 25-point range that starts at approximately 25 percent and ends at approximately 55 percent. Various synthetic netlists have been created with the intention of breaking out of this range, but we have not been successful in this endeavor. All of the synthetic netlists that have been created thus far seem to fall into the low end of percent utilization, with numbers in the range of 35%. While it is not entirely clear why we have yet to encounter any netlists that are outside of this range, this effect could likely be caused by the method we are using to generate the template reduction list.

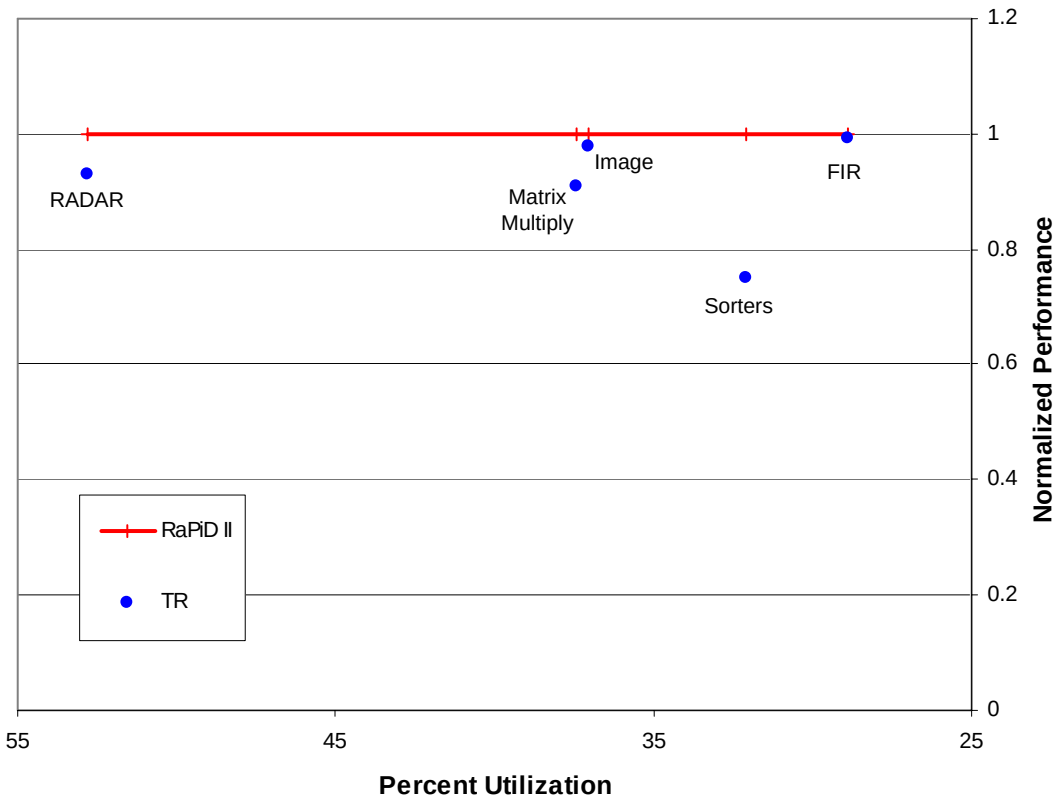
Table 6-3: This table shows the number of cells used by each application group, followed by the performance of each application group during the initial run, during pre-reduction, and during post-reduction. The performance increase is measured from the initial runs and the post-reduction runs.

Application Group	Number of Cells	Initial Run (ns)	Pre-Reduction (ns)	Post-Reduction (ns)	Performance Increase (%)
FIR	20	48.76	55.87	48.45	0.64
Image Processing	30	76.52	91.82	74.81	2.23
Matrix Multiply	30	78.82	87.07	71.67	9.07
RADAR	7	28.69	30.47	26.72	6.87
Sorters	15	37.93	35.43	28.47	24.94
Average					8.75

6.4.3 Performance

The performance of the mapped netlists was evaluated at three stages by the place-and-route tool, as shown in Table 6-3 and Graph 6-3. The first stage was the initial run. The initial run is an average of the performance of each netlist in the application domain, with the condition that each netlist is unaware of the other netlists in the application domain. Therefore, each netlist in the application group is able to use all of the resources without any constraints. The pre-reduction performance numbers come from the third run, and are an average of the performance of each netlist in the application group with constraints placed upon which functional units and routing resources are available to a particular netlist. As compared to the initial run, the netlists are aware of each other, and thus are unable to use all resources unfettered. Finally, the post-reduction performance numbers are an average of the performance of each netlist in the application group after template reduction has been performed on the layout.

While the gains in area are substantial, the performance results are not. This can be attributed to the fact that the only real gains in performance generated by the Template Reduction Method are in the interconnect. In essence, this method is moving functional units closer together without changing the functional units themselves. The overall performance of the circuit appears to be dominated by the functional unit delays and not the interconnect delays. Therefore, this method has a minimal impact on performance. It should be noted that the place and route tool is unable to retime signals, as mentioned in Chapter 5. Therefore, any performance numbers generated by the place and route tool should only be used for relative comparisons of the three methods.



Graph 6-3: This graph shows the normalized performance of each benchmark set. The x-axis is the percentage of the resources of the fixed tile needed to support the benchmark set. The y-axis is the performance of each benchmark set normalized to the performance of the benchmark set running on the RaPiD II fixed tile.

6.5 Summary

In this chapter, we have presented the Template Reduction Method for automating the layout of custom domain specific reconfigurable devices for SOC. The Template Reduction Method is able to leverage full custom designs, while still removing any unneeded resources that are not required to support the targeted application domain. This method is able to create circuits that perform at or better than that of the initial full custom template. If the template is a superset of the application domain, then this method produces circuits that are approximately 47.6% smaller and 8.75% faster than the original feature rich template. However, if the initial template is not a superset of the targeted application domain, then the Template Reduction Method will not be able to produce a solution. If this occurs, the designer would be forced to either use a different template that contains a different resource mix, or the Circuit Generator Method or the Standard Cell Method.

It should be noted that the generation of a quality template tile is not only a critical aspect of the Template Reduction Method, but to any method that seeks to create reconfigurable hardware that is targeted at a wide range of application domains. When creating such a tile, many tradeoffs occur. For instance, should the template be able to handle the widest range of domains possible, or should it be targeted at a couple of highly used domains with minimal support for others? Deciding one over the other has a big impact on the overall area of the circuits and on how, and if, mappings of different application domains perform on the circuit.

Chapter 7

Circuit Generator Method

The Standard Cell Method, which will be discussed in Chapter 8, is very flexible, and it gives designers the ability to implement almost any circuit. However, one drawback associated with the flexibility of this design method is its inability to leverage the regularity that exists in FPGAs. By taking advantage of this regularity, a method of automatically generating circuits may produce designs that are of higher quality than standard cell based designs, without the need for feature-rich template cells.

One way of creating very regular circuits is by using generators. Circuit generators are used to great effect when creating memory arrays, and since FPGA components, and RaPiD components by extension, have well-known constrained structures like memory arrays, they are well positioned as viable candidates for circuit generators. The Circuit Generator Method will be able to handle a wider variety of possible architectures than the Template Reduction Method, since the Circuit Generator Method will not be limited to a particular set of templates. Thus, circuit generators are positioned to fill the gap between the inflexible, but powerful Template Reduction Method and the very flexible, but less efficient, Standard Cell Method.

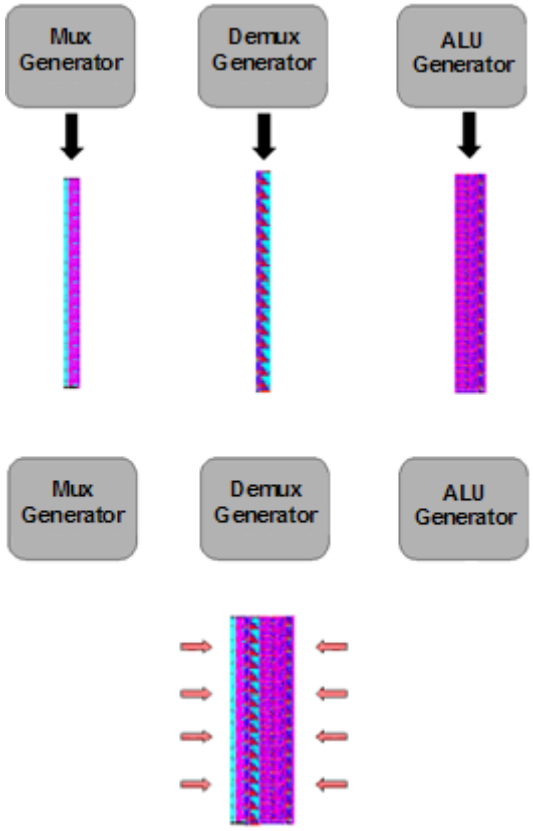


Figure 7-1: The top figure shows the initial generation of circuits by three generators. Once the circuits have been generated, they are abutted together to create the functioning reconfigurable array, which is shown in the bottom figure.

We have implemented circuit generators to automatically create the parts of the domain-specific reconfigurable device that inherently have regularity. This includes generators for the routing channels, DFFs, functional units, muxes, demuxes, and bus connectors. We have also implemented generators to create the parts of the reconfigurable device that do not have regularity, like the ALU, the multiplier, and the embedded memories. However, these generators are not able to vary the structures they create in any significant way. To create an entire reconfigurable subsystem out of blocks of logic that the circuit generators have generated, one would only need to abut the blocks together,

which is shown is Figure 7-1. Therefore, all of these generators have been combined to create a method that is capable of generating a complete reconfigurable subsystem.

7.1 Approach

One reason why memory generators are so efficient at creating memory arrays is their flexibility in tailoring the array to meet the design specifications, while at the same time minimizing area and maximizing performance. The Circuit Generator Method performs in much the same way (see Figure 7-1). This method does have a weakness, which is the availability and flexibility of a wide range of generators. Therefore, to truly take advantage of the Circuit Generator Method, the creation of generators that enable the designer to modify all or most of the components in RaPiD II is an important goal of this method.

The current approach for the Circuit Generator Method is a mix of two types of generators. One type of generator enables the designer to modify certain parameters for units like the mux, demux, pipeline register, and bus connector. The other type of generator does not allow the designer to modify any parameters for units like memory blocks, the ALU, and the multiplier. This last type of generator is just placing the original full custom circuits into the array, with modified interconnect for the inputs and outputs of the units that enable the unit to be tied into the overall array. This mix of approaches is necessary, since it is not currently feasible to create true generators that extract regularity from all units.

In the current implementation of the method, it was necessary to fix some of the degrees of freedom. This enabled the generation of preliminary results, which can be used as a guide to relax some of the fixed variables to further improve the quality of the generated circuits in future versions of this method. With this in mind, the generated circuits are loosely fixed in the vertical direction based upon the number of buses and the number of bits on each bus needed to support the specified architecture. In essence, the minimum number of tracks is ascertained from the architecture description, which in turn enables us to, with the TSMC .18μm rules as a guide, establish the height of the array using the width of each track and the minimum metal spacing rules. Once the height of the array is set, the other generators use this as an initial input parameter.

7.1.1 Silicon Compilers

The idea of using generators to create circuits is not unique. As mentioned, memory generators are quite popular and are capable of automatically generating very efficient memory arrays based upon a set of design parameters. However, memory generators are not the only type of parameter-based generators. Silicon compilers of the early 1980s were created with the notion of directly producing a mask layout from a behavioral or functional description of a circuit. This was, and still is an ambitious design goal, and with the advent of logic synthesis technology and industry-standard hardware description languages, silicon compilers have fallen out of favor.

The Bristle Block design system was the first version of a silicon compiler [64]. The goal of the Bristle Block system was the production of an entire LSI mask set from a single page high-level description of an integrated circuit. Bristle Blocks utilizes a three-pass compiler, consisting of a core pass, a control pass, and a pad pass to create a layout targeting one specific type of chip architecture. The circuits created by the Bristle Block design system were approximately +/- 10% of the area of comparable circuits created by hand.

The OASIS silicon compiler was introduced in 1990 as an open architectural system for semi-custom design [65]. It consists of a variety of public domain tools in addition to tools written specifically for OASIS. The full system consisted of four major parts, namely the compiler and logic synthesizer, the simulator, the automatic test pattern generator, and the automatic layout generator. A design control language (DECOL) was created to manage and tie together all of the tools within OASIS. Since OASIS is not confined to one specific type of chip architecture with limited control, it is a more capable system than Bristle Blocks.

Bristle Blocks and OASIS are just two of the many different versions of silicon compilers that have been created, but they both highlight the common approaches used in silicon compilation. Silicon compilers were successful at creating efficient circuits when the circuit size remained small and the targeted domain was limited and well known in advance. However, after the advent of flexible HDLs like VHDL and powerful synthesis tools such as Synopsys [41], silicon compliers fell out of favor. The Circuit Generator

Method will be using a similar approach to the original Bristle Blocks silicon compiler, since the circuits that will be created by the Circuit Generator Method are very regular and have very specific design constraints. In addition, the Circuit Generator Method is also similar in function to OASIS, because it uses Perl to parse the Verilog input and to create the SKILL code that drives the Cadence tool to create the mask layouts.

7.1.2 Tool Flow

The entire tool flow for the Circuit Generator Method is shown in Figure 7-2. The first step in the generation of circuits is to receive, as input from the AML, AMO, or GH versions of the Architecture Generator [7], a Verilog representation of the custom reconfigurable architecture. The next step in the circuit generation involves parsing the Verilog that was generated by the Architecture Generator. The Verilog is parsed into separate generator calls, including any required parameters. For example, the following Verilog code:

```
bus_mux16_28 data_reg_0_In(.In0(ZERO),….,Out(WIRE_data_reg_0_In));
```

would be parsed so that the MUX generator would create a structure that contains sixteen 28-to-1 muxes that are stacked on top of each other with their control tied together.

After the Verilog has been parsed, the next step involves generating the Cadence SKILL [39] code needed to implement the specified circuit. This is done by using Perl Cadence SKILL code generators, which are similar in scope to the Cadence SKILL code generator used in the Template Reduction Method. However, unlike in the Template Reduction

Method, the Circuit Generator Method creates circuits in an additive, not a subtractive fashion. The code generators call primitive Cadence SKILL functions that are able to do simple tasks in Cadence, including opening, saving and closing files, drawing polygons in the layout, and instantiating cells. The code generators create circuits for all of the units needed to create the reconfigurable circuits, including muxes, demuxes, pipelined registers, bus-connectors, ALUs, multipliers, and SRAM blocks.

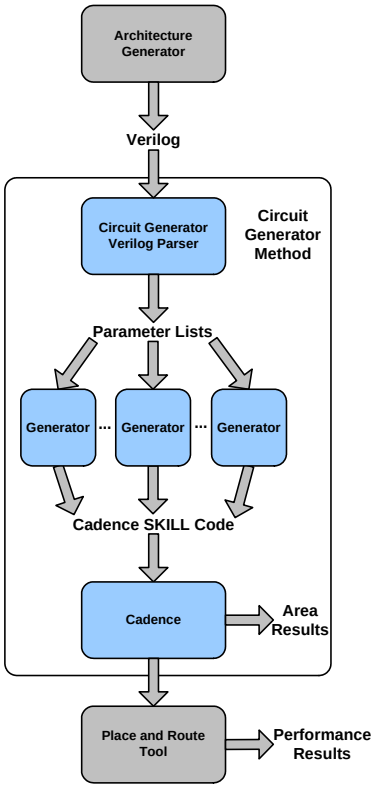


Figure 7-2: The tool flow of the Circuit Generator Method. The first step is receiving the Verilog representation of the reconfigurable circuit from the Architecture Generator [7]. The next step is to parse the Verilog. The parsed Verilog is then sent to the various generators, which create Cadence SKILL code [39] that will generate the circuits. The Cadence SKILL code is then sent into Cadence, which will do the actual circuit creation. The final step involves using the P&R tool [6] to generate performance numbers.

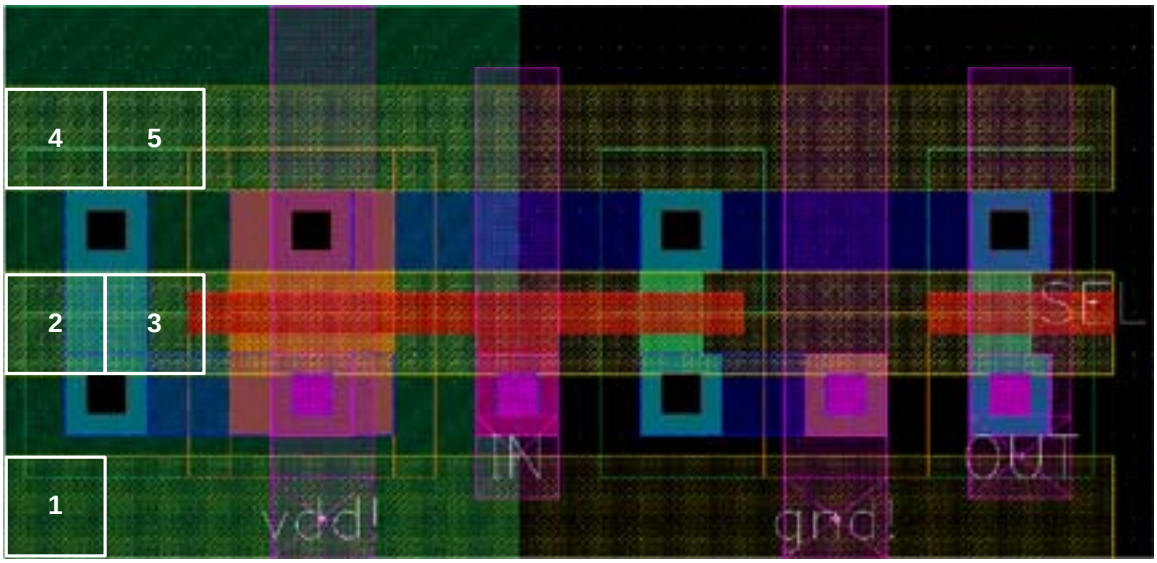


Figure 7-3: One tristate inverter laid out in a horizontal fashion, which is the smallest building block of both the muxes and demuxes, has enough length in the vertical direction to support up to three horizontal routing tracks. Three routing tracks are able to support up to five bits. The metal lines pictured in the figure are on the fourth and sixth metal layers, of the six metal layer TSMC .18µm process. The fifth metal layer is reserved for vertical jogs between metal layers four and six.

The generated circuits are targeted at the TSMC .18µm process. The height of the generated circuits is set by the number of routing tracks needed to support the number of bits per bus specified by the architectural description. In the TSMC .18µ process, a minimum size tristate inverter, laid out in a horizontal fashion, is equivalent in height to three routing tracks, as shown in Figure 7-3. In essence, using metal four and metal six for routing, three routing tracks are able to support a maximum of five bits, which is also shown in Figure 7-3.

Table 7-1 lists the different metal layers and their purpose. We were restricted in our use of metal layers, since the bulk of the layouts that we were working with were laid out in an HP .50µm 3 metal layer process. We were thus only able to route over cells with metal 4, metal 5, and metal 6.

Table 7-1: The metal layers and their corresponding purpose.

Metal Layer	Purpose
Metal 1	Local Cell Routing, Control
Metal 2	Power and Ground
Metal 3	Local Cell Routing
Metal 4	Horizontal Routing
Metal 5	Vertical Routing
Metal 6	Horizontal Routing

Once the circuits have been produced, the next phase of circuit generation involves the creation of SKILL code that will abut the generated circuits together. This glue logic involves ensuring that the routing between the circuits is maintained, as well as ensuring that a minimum amount of space is wasted between generated circuits. Careful planning of any "edge effects", which includes problems with wells, active-area, or implant, is critical to help mitigate any problems that can lead to wasted space.

The last step is to run Cadence to create the circuits and to place the generated circuits together along the horizontal axis with glue logic establishing connections between the various generated circuits. The runtime of the code is on the order of ten minutes, unlike the Template Reduction Method, which is on the order of hours. The reason why the runtimes varies so much between the methods is the fact that the Circuit Generator Method is essentially creating flat circuits with only a few layers of hierarchy, while the Template Reduction Method reduces circuits that contain many levels of unique hierarchy. Dealing with a large structure of unique cells takes hours due to opening and closing of so many different files. This fact lead us to make sure that the Circuit Generator Method operated on flat circuits with only a few levels of hierarchy.

Once the circuits are generated, wire lengths are extracted which are used by the Place and Route tool generator to determine performance characteristics for the specified architecture. The Place and Route tool maps the various netlists from the application domains onto the architecture to determine the performance numbers, using its detailed wire and functional unit models to attain these numbers. The next sections will go over the various generators in more detail.

7.2 Generators

We have created a generator for each of the components present in the RaPiD II template. The generators can be grouped according to the similarity of their generated structures, or according to the similarity of the circuit generation methodology. Therefore, the generators are grouped as follows: the mux and demux generators, the pipeline register and bus connector generators, and the ALU, multiplier, and embedded memory generators. The following sections discuss how each group of generators automatically generates circuits.

7.2.1 MUX and Demux Generators

The mux and demux generators are used to set the initial height of the reconfigurable arrays that the Circuit Generator Method creates. One goal of the circuit generators is to ensure that the capacitance and the delay of the muxes and the demuxes that are generated are as small as possible. Towards this end, we use the full custom muxes and demuxes used in the Template Reduction Method (Figure 7-4) as a starting point.

Therefore, all muxes and demuxes that are generated use the same full custom tristate inverters that are used in the Template Reduction Method, as can be seen in Figure 7-5, but in a horizontal instead of a vertical arrangement to help reduce the complexity of the generation of the control lines.

Another difference between the RaPiD I mux and the generated mux is the use of polyslicon lines instead of metal one to route the control wires. The move to use metal one wires instead of polysilicon wires was based upon the fact that under the TSMC .18µ design rules, a tremendous amount of polysilicon wire jogs would be needed to fit contacts into the control routing channels. In addition, since the number of buses that the Circuit Generator Method needs to support can be quite large, the length of the control wires has the potential to grow quite long. The long potential length of control wires makes metal one a better potential candidate than polysilicon. Metal 2 and metal 3 wires were considered, but the cells make use of both of these metals, as shown in Table 7-1, and they would not provide any additional benefit over metal 1.

One final point that should be made about the creation of these structures is the fact that the tristate inverters used for both the muxes and the demuxes are sized the same. This can compromise performance, since the demux drivers may need to drive much more capacitance than the mux drivers do. We have chosen not to size the demux tristate inverters in this version of the Circuit Generator Method to reduce the complexity of the problem. In future versions, the demux generators should be able to create tristate inverters that are sized more appropriately, which will address this shortcoming.

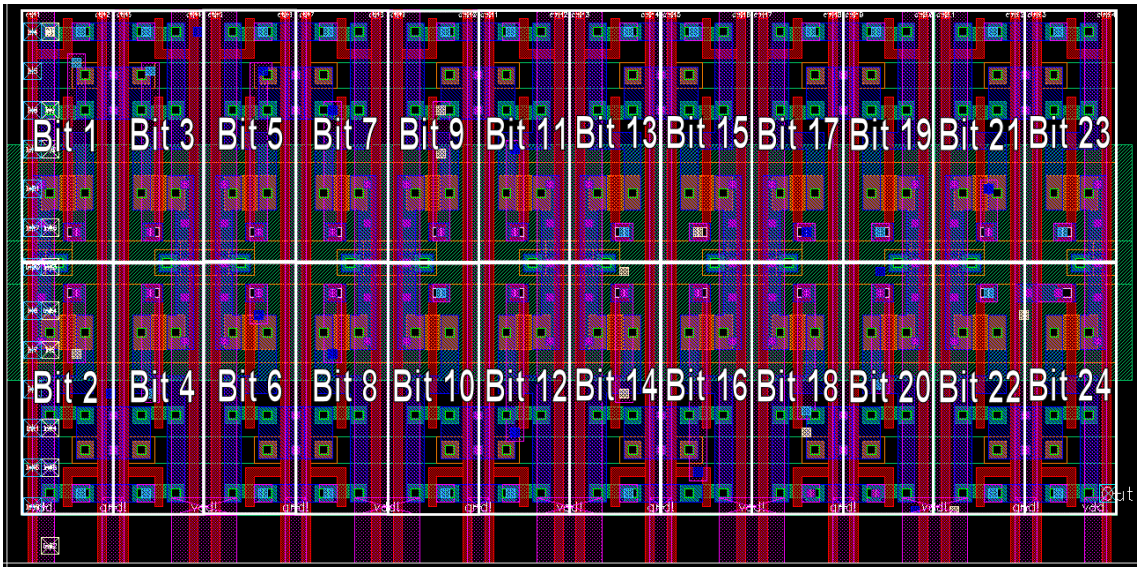


Figure 7-4: Full Custom 24-to-1 mux used in the Template Reduction Method.

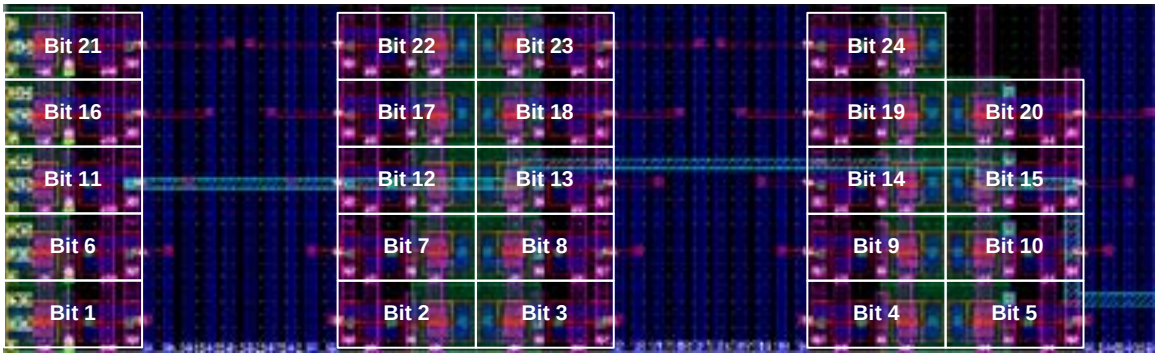


Figure 7-5: Generated 24-to-1 mux used in the Circuit Generater Method.

The process used in the generation of muxes and demuxes is fashioned after the process used to create the full custom muxes and demuxes in the full custom RaPiD II tile, only our approach is automated. The decision to create a new row of tristate inverters is based upon the number of metal tracks that can fit in the vertical area of one horizontally placed tristate inverter, which happens to be three. In essence, if six tracks are needed, you would use two rows or tristate inverters, but if nine tracks are needed you would use three tracks. The formula to determine the number of tristate inverter rows is floor((n+1)/5),

which is borne out by Table 7-2. Figure 7-6 shows the configurations of muxes from 4 bits in size to 20 bits in size, in 4 bit increments. When minimizing wasted area, the most efficient structures are muxes or demuxes of bit size p, where p mod 5 is equal to zero, since each horizontal tristate inverter is three tracks, or 5 bits, high. Structures with size q, where q mod 5 is equal to one, are the most inefficient.

Table 7-2: Various configurations of muxes or demuxes based upon the number of bits. The formula for establishing the number of rows is floor((n+1)/5), where n is the number of bits in the mux or demux. The most efficient structures are of bit size p, where p mod 5 is equal to zero. The most inefficient structures are of bit size q, where q mod 5 is equal to one.

Number of Bits	Number of Tracks	Number of Rows	Number of Tristate Inverters Per Row
4	3	1	4
5	3	1	5
6	4	1	6
7	5	1	7
8	5	1	8
9	6	2	5, 4
10	6	2	5, 5
11	7	2	5, 6
12	8	2	6, 6
13	8	2	6, 7
14	9	3	5, 5, 4
15	9	3	5, 5, 5
16	10	3	5, 5, 6
17	11	3	5, 5, 6
18	11	3	6, 6, 6
19	12	4	5, 5, 5, 4
20	12	4	5, 5, 5, 5
21	13	4	5, 5, 5, 6
22	14	4	5, 5, 6, 6
23	14	4	5, 6, 6, 6
24	15	5	5, 5, 5, 5, 4
25	15	5	5, 5, 5, 5, 5
26	16	5	5, 5, 5, 5, 6
27	17	5	5, 5, 5, 6, 6
28	17	5	5, 5, 6, 6, 6

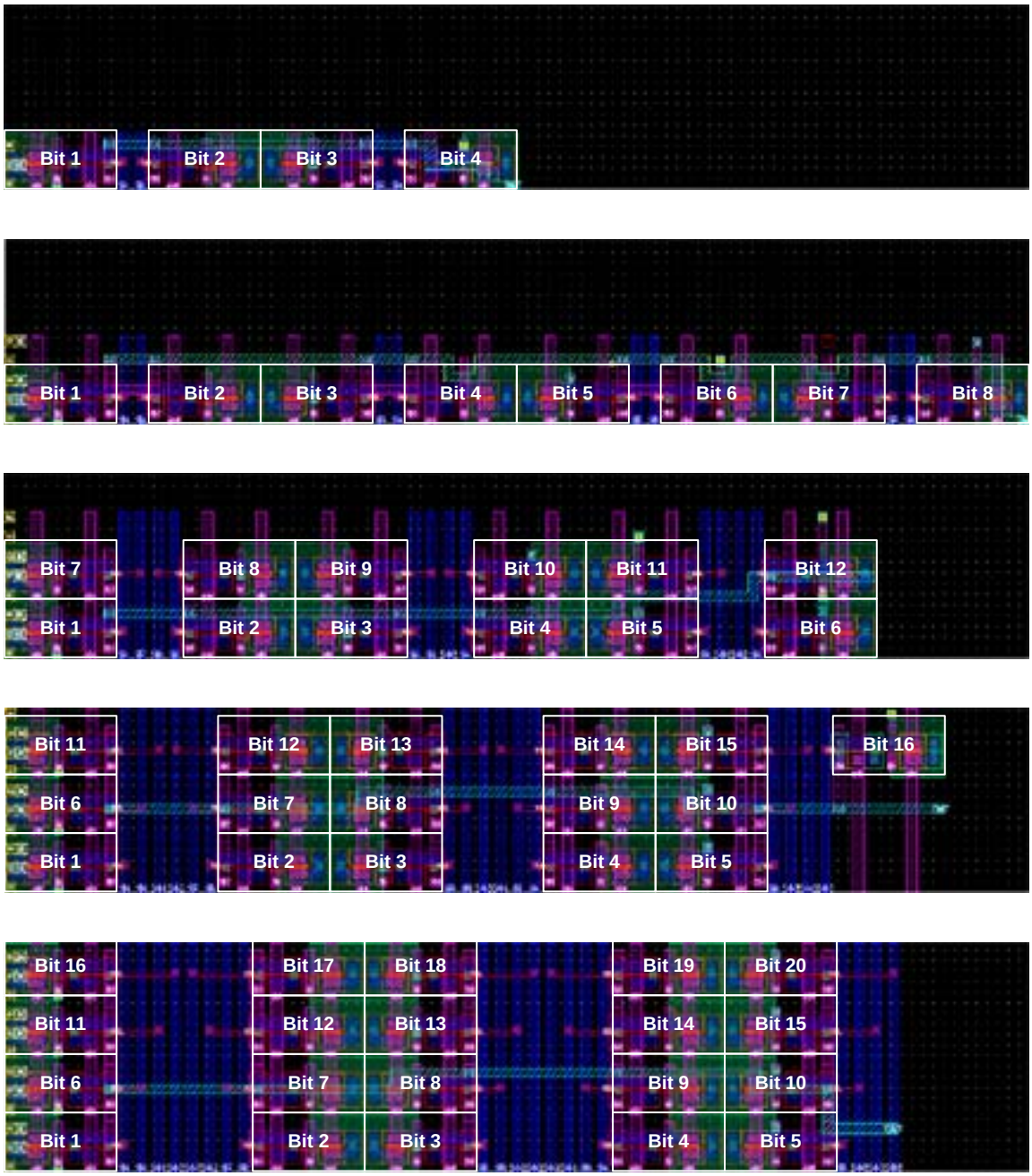


Figure 7-6: Various configurations of muxes based upon the number of bits, and the number of routing tracks. The top figure is a 4 bit mux, followed by 8, 12, 16, and 20. All of the figures are to scale. Notice the increase in the width of the control routing channel as the number of tristate rows increase, and the wasted space in the 16 bit mux.

7.2.2 Pipeline Register and Bus Connector Generators

The approach to the generation of bus connectors and pipeline registers is more constrained than the approach used to generate muxes and demuxes. Only two types of bus connectors (BC) or pipeline registers (PR) are generated, either one or three delay versions. Figure 7-7 and Figure 7-8 are the mask layouts for both types of BC's and PR's. It can also be seen in Figures 7-7 and 7-8 that the BC and PR are very similar in size and structure. Therefore, based upon these similarities, the generated structure of BC's and PR's for a particular circuit will follow the same pattern, depending on whether the specified architecture requires units that have one or three delays. The full custom RaPiD II tile consists of only one delay PR's and BC's, while the original RaPiD I template consist of three delay PR's and BC's.

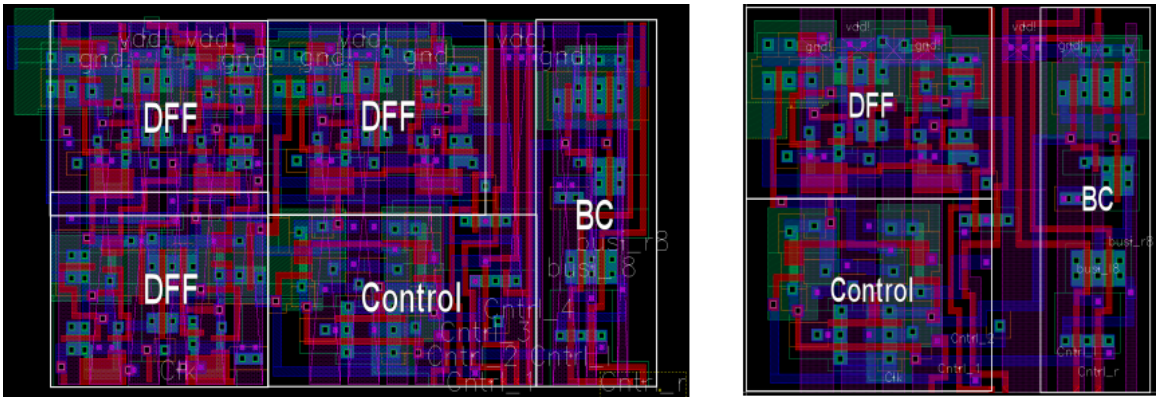


Figure 7-7: Bus Connectors. The BC on the left is capable of three delays, while the BC on the right is capable of one delay.

7.2.3 ALU, Multiplier, and SRAM Generators

True generation of ALUs, Multipliers, and memories is a more complicated proposition, when compared to the generation of muxes, demuxes, PRs, or BCs. That is why in this

version of circuit generation we do not generate these functional units, we only generate the input and output interconnect needed to tie them into the routing fabric.

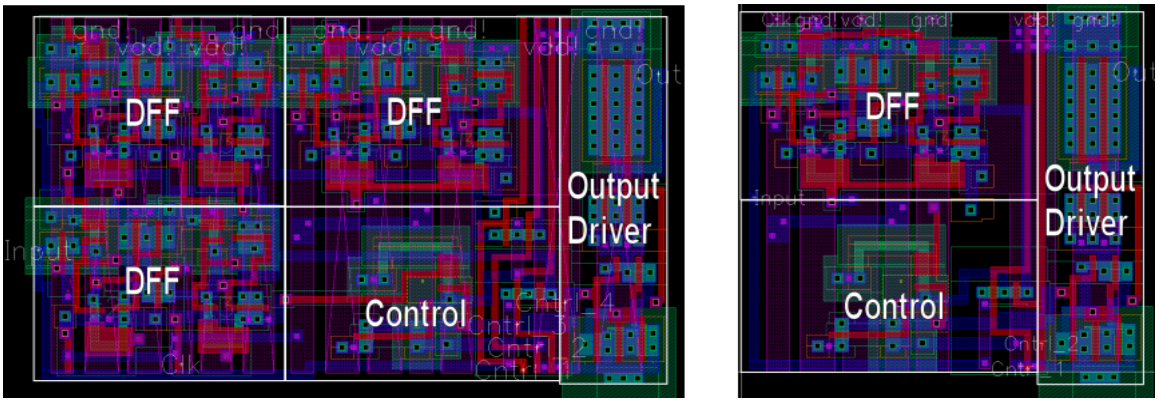


Figure 7-8: Pipeline Registers. The PR on the left is capable of three delays, while the PR on the right is capable or one delay.

The full-custom ALU used in RaPiD I, shown in Figure 7-9, is a carry-look-ahead adder design. The ALU was built in a very modular fashion, shown in Figure 7-7, but there are still very significant issues related to modularizing the carry-look-ahead portions. One drawback to the implementation of a CLA is the fact that to create an efficient structure, the circuit should be laid out in a linear fashion. This limitation fixes possible implementations of a CLA in either the horizontal or the vertical linear direction. In our case, the ALU is laid out in the vertical direction, so an increase or decrease in the bit width of the ALU would lead to an increase or a decrease, respectively, of the overall height of the RaPiD array. Because of these limitations and the fact that all of the application sets we have at our disposal currently only call for a 16 bit ALU, the only generation that occurs when an ALU is specified is the instantiation of the existing full custom ALU. The real work of the generator is ensuring that the I/Os of the ALU correctly match up with the routing fabric.

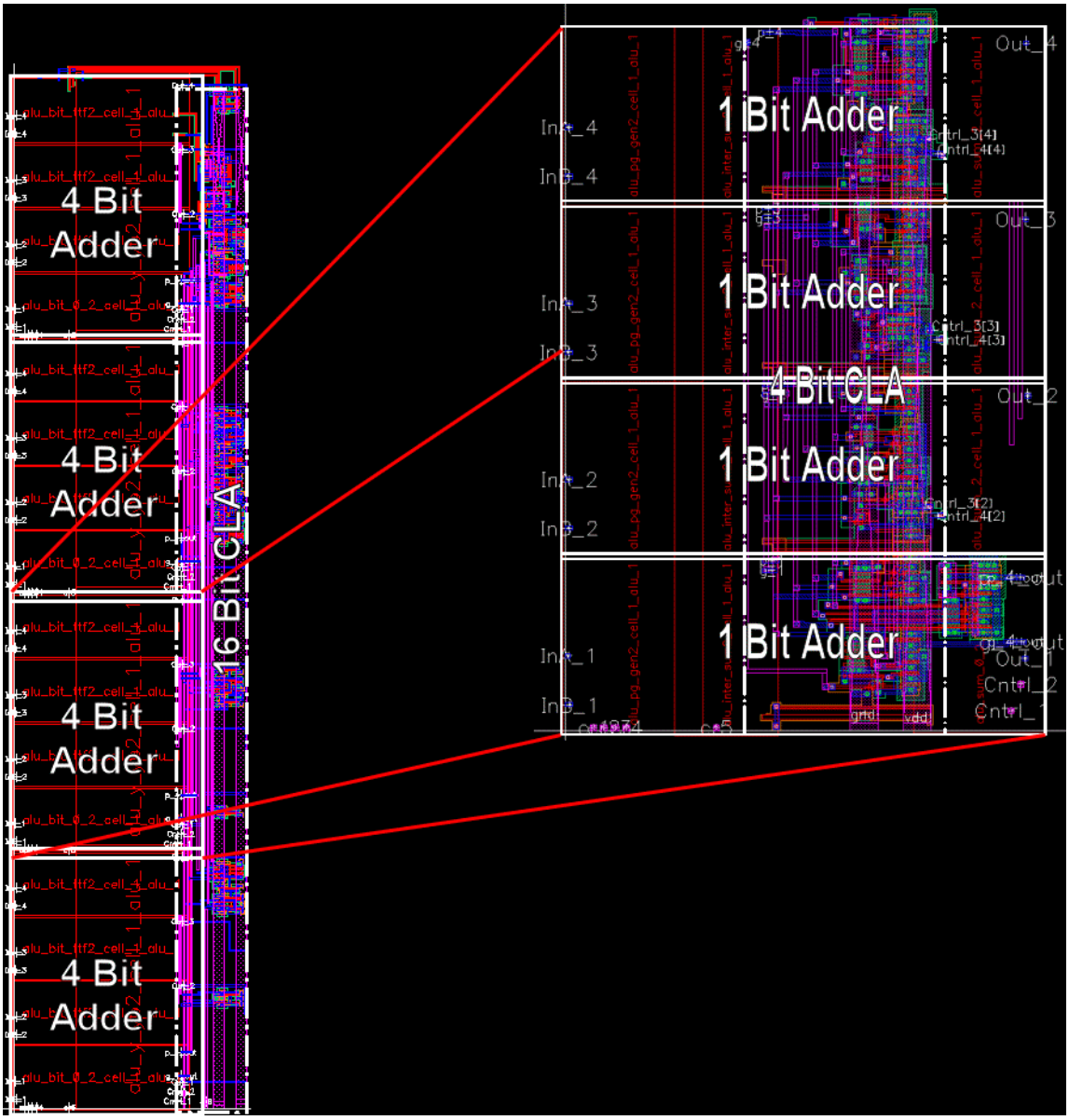


Figure 7-9: This shows a modular 16 bit carry look ahead ALU.

The multiplier and embedded memory elements are even less modular and more tightly coupled than the ALU. Therefore, a similar methodology to that used to generate the ALU is used when generating these units. Once again, the main work of the generators consists of coupling the I/Os of these units with the routing fabric.

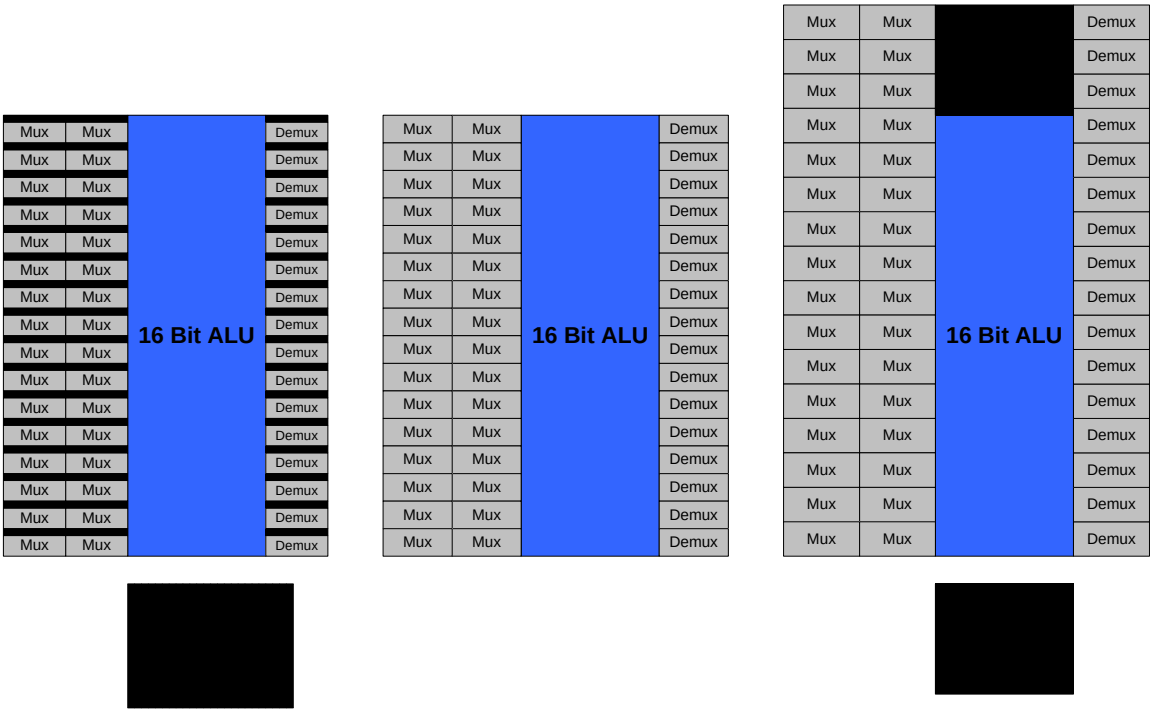


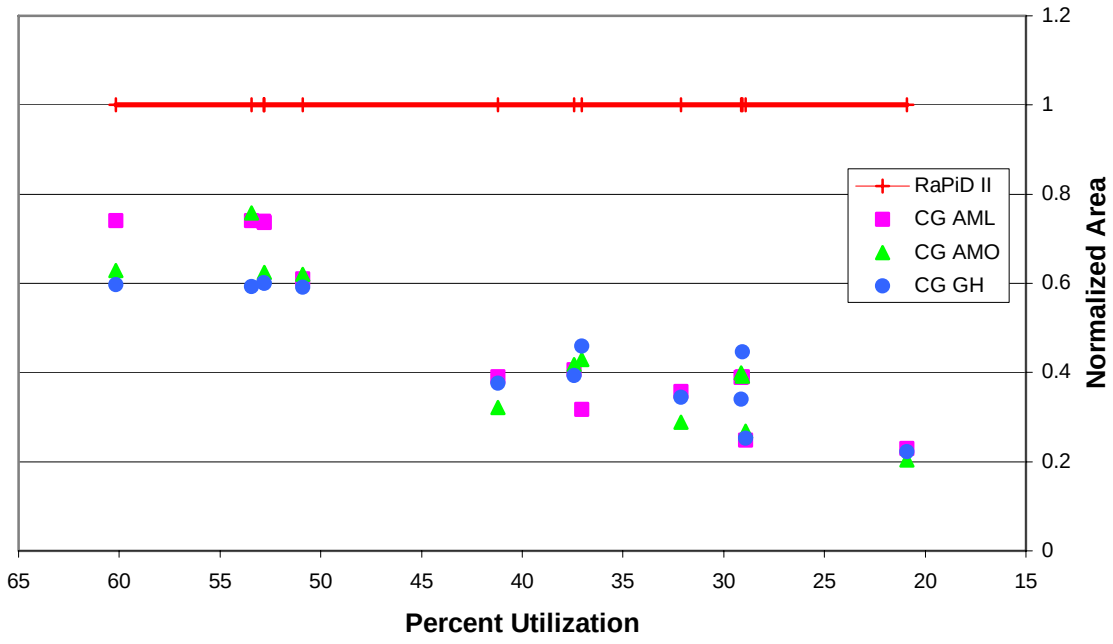
Figure 7-10: Three different generated ALUs. Black boxes represent wasted area. The version on the left has 20 tracks. The version in the middle has 24 tracks. The version on the right has 28 tracks. The most wasteful version is the ALU on the left with 20 tracks. This is verified when all of the wasted space is combined into a single block, as seen below the ALUs.

It is limiting and sometimes wasteful (as shown in Figure 7-10), not to be able to vary the bit width of these functional units, but for our purposes, it suffices. This is because the application sets that we have available to us only require functional units that are 16 bits wide. In the future, it would significantly increase the flexibility of this method if the bit width of all of the functional units could be varied. However, the creation of these types of generators at this time is beyond the scope of this work.

7.3 Results

We are using all of the 13 application sets of netlists, which were explained in detail in chapter 5, to evaluate the Circuit Generator Method. As described earlier, the first step in

the generation of circuits is to receive as input from the Architecture Generator a Verilog representation of the specified architecture. The Architecture Generator, as previously discussed in Chapter 4, uses three different methods to create the routing for the architectures that it generates. The three methods are Greedy Histogram (GH), Add Max Once (AMO), and Add Min Loop (AML). Therefore, all of the results in this section will reflect each of these three methods of architecture generation. When bypassing the Architecture Generator and using the Verilog representation of the RaPiD II tile, the layout of the circuit generated version of the RaPiD II tile is 6% larger than the full-custom RaPiD II tile layout.



Graph 7-1: This graph, which was generated from the data presented in Table 7-3, shows the normalized area of each benchmark set. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the benchmark set. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile.

7.3.1 Area

The first metric we can use to evaluate the quality of the circuits that are generated by the Circuit Generator Method is the area of each circuit. Graph 7-1 shows the area of each benchmark set, normalized to the area of the unaltered template. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile, with lower values representing smaller, and therefore more desirable, circuits. The x-axis represents the percentage of the fixed tile resources needed to support the benchmark set.

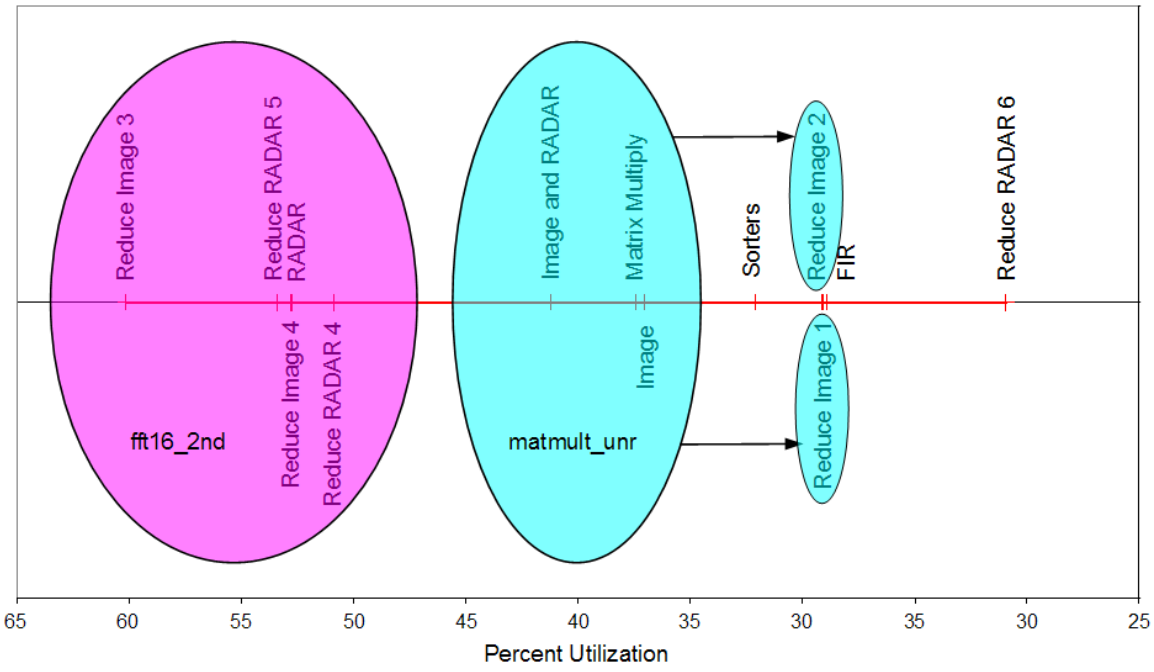


Figure 7-11: The thirteen application domains ordered along the horizontal axis by percent utilization. Application domains with higher percent utilizations are to the left, while application domains with lower percent utilizations are to the right. The fft16_2nd and the matmult_unr netlists dominate ten of the thirteen application domains, which is indicated by the red and blue circles.

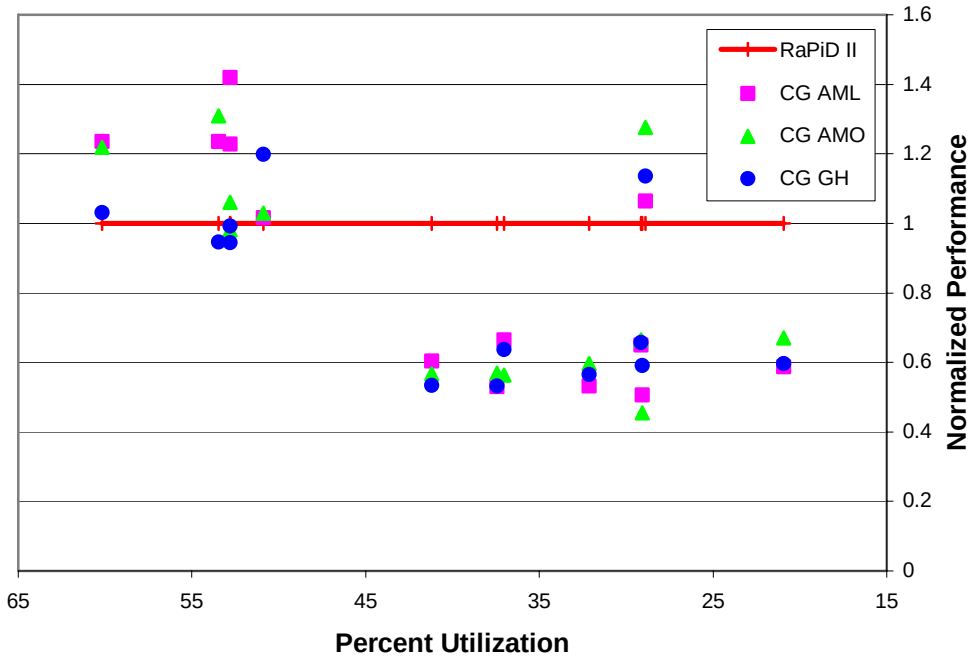
From Graph 7-1, the most noticeable feature is the fact that the Circuit Generator Method is consistently able to create circuits that have a smaller area than the fixed RaPiD II tile

throughout the 60% to 20% utilization range. Another feature of Graph 7-1 is the fact that the benchmarks seem to cluster into two groups, one group that has a high percent utilization, and another that has a low percent utilization. This is due to the domination of certain netlists in each application group, which can be seen more clearly in Figure 7-11, and can also be seen in Table 5-4. The first cluster is dominated by the fft16_2nd netlist, and the second cluster is dominated by the matmult_unr netlist. Finally, when comparing the three types of Architecture Generation algorithms, it is apparent that GH outperforms AML and AMO when percent utilization is high, but as the application domain narrows and the percent utilization is reduced, the three algorithms perform similarly.

Table 7-3: The application domains and their respective percent utilization are shown. The area normalized to the RaPiD II tile of the circuit generated architectures are shown. The architectures were generated from Verilog that was generated by the AMO, AML, and the GH Architecture Generators.

Application Domain	Percent Utilization	Normalized Area			
		AML	AMO	GH	Average
Reduced RADAR 6	20.92	0.229	0.204	0.223	0.219
FIR	28.90	0.248	0.269	0.253	0.257
Reduced IMAGE 1	29.07	0.391	0.392	0.446	0.410
Reduced IMAGE 2	29.15	0.389	0.400	0.340	0.377
Sorters	32.12	0.357	0.288	0.344	0.330
Image	37.05	0.317	0.429	0.459	0.402
Matrix Multiply	37.43	0.407	0.418	0.394	0.406
Image and RADAR	41.21	0.391	0.322	0.376	0.363
Reduced RADAR 4	50.88	0.610	0.620	0.592	0.607
RADAR	52.79	0.737	0.625	0.601	0.654
Reduced IMAGE 4	52.82	0.739	0.616	0.601	0.652
Reduced RADAR 5	53.45	0.741	0.757	0.593	0.697
Reduced IMAGE 3	60.18	0.741	0.629	0.597	0.656
Average	---	0.484	0.459	0.448	0.464

7.3.2 Performance



Graph 7-2: This graph, which was generated from the data presented in Table 7-4, shows the normalized performance of each benchmark set. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the benchmark set. The y-axis is the performance of each benchmark set normalized to the RaPiD II fixed tile.

Graph 7-2 shows the performance of each benchmark set after it has been normalized to the performance of the fixed RaPiD II tile, where lower performance indicates a higher quality circuit. Once again, as seen on Graph 7-1, the benchmarks are clustered into two groups, depending upon which netlists are dominating within the application domains. Another feature that can be seen on the graph is the fact that the performance of the benchmarks increases, represented by lower values on the graph, as percent utilization decreases. This is an overall trend with some outliers, and these results are highly dependent on the efficiency of the place and route tool. The most noticeable outlier is the

FIR application domain. As can be seen in Table 7-4, two netlists dominate the performance of this application group, namely the firsm3_unr and the firsymenven_unr, causing it to perform poorly. It should be noted that the place and route tool is unable to retime signals, as mentioned in Chapter 5. Therefore, any performance numbers generated by the place and route tool should only be used for relative comparisons of the three methods.

Table 7-4: The normalized average performance of the circuits generated using the AML, AMO, and GH architecture generators for each application group.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Reduce RADAR 6	AML_decnsr	9.5	11.6	19.8	0.59	0.62
	AML_psd	13.7				
	AMO_decnsr	10.2	13.3	19.8	0.67	
	AMO_psd	16.3				
	GH_decnsr	8.7	11.8	19.8	0.60	
	GH_psd	14.9				
FIR	AML_firsm2_unr	23.3	35.5	33.3	1.06	1.16
	AML_firsm3_unr	52.4				
	AML_firsm4_unr	23.3				
	AML_firsymeven_unr	42.8				
	AMO_firsm2_unr	24.3	42.5	33.3	1.28	
	AMO_firsm3_unr	67.5				
	AMO_firsm4_unr	24.3				
	AMO_firsymeven_unr	54.1				
	GH_firsm2_unr	24.3	37.9	33.3	1.14	
	GH_firsm3_unr	54.3				
	GH_firsm4_unr	24.3				
	GH_firsymeven_unr	48.7				

Table 7-4: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Reduce Image 1	AML_firtm_2nd	22.5	34.1	67.4	0.51	0.52
	AML_matmult_unr	45.7				
	AMO_firtm_2nd	21.4	30.6	67.4	0.45	
	AMO_matmult_unr	39.8				
	GH_firtm_2nd	21.4	39.9	67.4	0.59	
	GH_matmult_unr	58.3				
Reduce Image 2	AML_1d_dct40	58.8	57.2	88.0	0.65	0.66
	AML_fft16_2nd	73.0				
	AML_matmult_unr	39.8				
	AMO_1d_dct40	53.2	58.5	88.0	0.67	
	AMO_fft16_2nd	81.9				
	AMO_matmult_unr	40.4				
	GH_1d_dct40	44.6	57.9	88.0	0.66	
	GH_fft16_2nd	82.9				
	GH_matmult_unr	46.0				
Sorters	AML_sort_2d_g	18.9	18.2	34.2	0.53	0.56
	AML_sort_2d_rb	17.5				
	AML_sort_g	18.0				
	AML_sort_rb	18.3				
	AMO_sort_2d_g	23.4	20.4	34.2	0.60	
	AMO_sort_2d_rb	18.7				
	AMO_sort_g	20.5				
	AMO_sort_rb	18.8				
	GH_sort_2d_g	21.8	19.4	34.2	0.57	
	GH_sort_2d_rb	19.1				
	GH_sort_g	18.6				
	GH_sort_rb	18.0				

Table 7-4: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Image	AML_1d_dct40	68.6	53.1	79.9	0.66	0.62
	AML_fft16_2nd	85.4				
	AML_firtm_2nd	16.3				
	AML_matmult_unr	42.0				
	AMO_1d_dct40	46.5	45.0	79.9	0.56	
	AMO_fft16_2nd	68.5				
	AMO_firtm_2nd	21.2				
	AMO_matmult_unr	43.7				
	GH_1d_dct40	47.0	51.0	79.9	0.64	
	GH_fft16_2nd	79.7				
	GH_firtm_2nd	22.2				
	GH_matmult_unr	55.0				
Matmult	AML_limited_unr	41.9	41.7	78.6	0.53	0.54
	AML_matmult4_unr	41.8				
	AML_matmult_unr	42.8				
	AML_vector_unr	40.5				
	AMO_limited_unr	43.4	44.7	78.6	0.57	
	AMO_matmult4_unr	46.4				
	AMO_matmult_unr	46.8				
	AMO_vector_unr	42.2				
	GH_limited_unr	42.5	41.8	78.6	0.53	
	GH_matmult4_unr	43.9				
	GH_matmult_unr	40.7				
	GH_vector_unr	39.9				

Table 7-4: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Image and RADAR	AML_decnsr	46.5	48.4	80.1	0.60	0.57
	AML_psd	52.9				
	AML_1d_dct40	52.9				
	AML_fft16_2nd	79.7				
	AML_firtm_2nd	15.8				
	AML_matmult_unr	42.3				
	AMO_decnsr	37.8	45.4	80.1	0.57	
	AMO_psd	44.9				
	AMO_1d_dct40	44.1				
	AMO_fft16_2nd	86.2				
	AMO_firtm_2nd	20.0				
	AMO_matmult_unr	39.5				
	GH_decnsr	38.8	42.8	80.1	0.53	
	GH_psd	46.1				
	GH_1d_dct40	46.3				
	GH_fft16_2nd	61.3				
	GH_firtm_2nd	23.8				
	GH_matmult_unr	40.5				
Reduce RADAR 4	AML_decnsr	19.5	30.7	30.2	1.02	1.08
	AML_fft16_2nd	42.0				
	AMO_decnsr	18.8	31.1	30.2	1.03	
	AMO_fft16_2nd	43.4				
	GH_1d_dct40	23.6	36.3	30.2	1.20	
	GH_fft16_2nd	48.9				
RADAR	AML_decnsr	25.3	35.6	29.0	1.23	1.08
	AML_fft16_2nd	56.1				
	AML_psd	25.5				
	AMO_decnsr	19.2	30.8	29.0	1.06	
	AMO_fft16_2nd	47.9				
	AMO_psd	25.2				
	GH_decnsr	18.0	27.4	29.0	0.95	
	GH_fft16_2nd	39.5				
	GH_psd	24.8				

Table 7-4: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Reduce Image 4	AML_1d_dct40	34.1	47.0	33.1	1.42	1.13
	AML_fft16_2nd	59.8				
	AMO_1d_dct40	25.1	32.4	33.1	0.98	
	AMO_fft16_2nd	39.8				
	GH_decnsr	18.0	32.8	33.1	0.99	
	GH_fft16_2nd	47.7				
Reduce RADAR 5	AML_fft16_2nd	56.4	40.7	32.9	1.24	1.16
	AML_psd	25.0				
	AMO_fft16_2nd	60.6	43.1	32.9	1.31	
	AMO_psd	25.6				
	GH_fft16_2nd	39.5	31.2	32.9	0.95	
	GH_psd	22.9				
Reduce Image 3	AML_1d_dct40	27.0	33.1	26.8	1.24	1.16
	AML_fft16_2nd	58.9				
	AML_firtm_2nd	13.4				
	AMO_1d_dct40	31.2	32.7	26.8	1.22	
	AMO_fft16_2nd	55.0				
	AMO_firtm_2nd	11.8				
	GH_1d_dct40	25.9	27.7	26.8	1.03	
	GH_fft16_2nd	41.1				
	GH_firtm_2nd	16.0				
Average						0.836

7.4 Summary

In this chapter, we have presented the Circuit Generator Method for automating the layout of custom domain specific reconfigurable devices for SOC. The Circuit Generator Method is able to leverage the regularity that exists in FPGA designs in a method very similar to the creation of memory arrays by memory generators. This method is able to

create circuits that perform better than that of the RaPiD II full custom fixed tile, as long as the specified architecture does not require functional units or routing resources that do not have a corresponding generator. If this condition is met, then the Circuit Generator Method produces circuits that are approximately 46% smaller and 16% faster than the full custom RaPiD II fixed tile, as shown in Tables 7-3 and 7-4, respectively. However, if the specified architectures require units or routing resources for which a generator does not exists, then the Circuit Generator Method will not be able to produce a solution, forcing the designer to try either the Template Reduction Method or the Standard Cell Method.

Chapter 8

Standard Cell Method

The Template Reduction Method produces very efficient implementations, but it only functions if the proposed architecture does not deviate significantly from the provided macro cells. The Circuit Generator Method does not rely on a set macro cell like the Template Reduction Method, but it does rely upon the existence of a wide range of flexible generators, which must exist in order to create viable circuits. Therefore, to fill the gaps that exist between the templates, and to alleviate the need for a wide range of generators, we have implemented a Standard Cell Method of layout generation. This method will provide Totem with the ability to create a reconfigurable subsystem for any application domain, even for application domains in which the Template Reduction and Circuit Generator Methods have failed.

The use of standard cells provides an opportunity to more aggressively optimize the architecture than if templates or generators were used. This opportunity arises because the circuits created by this method are built from the ground up, and thus do not inherit any design tradeoffs that were implemented when the full custom template or circuit generators were created. For example, if a designer requires a low power design, then it is possible to use a low power library instead of a library that is optimized for area or

performance. Template Reduction and Circuit Generators can only compete with this level of flexibility by the creation of multiple templates or generators, each of which is targeted at different design goals, like lower power, smaller area, or higher performance. In addition, the use of the Standard Cell Method allows the designer to easily integrate the structures created by Totem into the typical SOC design flow, since standard cell flows are widely used throughout industry.

Unfortunately, the Standard Cell Method inherits all of the drawbacks introduced into a design when using standard cells instead of full custom layouts, including increased circuit size and reduced performance. Another drawback associated with the Standard Cell Method is the fact that generic standard cell libraries are not able to take advantage of the regularity that exists in FPGAs. In previous work [5], we explored the possibility of creating an FPGA optimized standard cell library. This was done by creating optimized cells, which included LUTs, SRAM bits, muxes, and demuxes, which are typical FPGA components. Since these types of resources are used extensively in FPGAs, an improvement of approximately 20% can be attained by the optimization of these cells, when compared to an unoptimized generic standard cell library [5]. In Figure 8-1, we show a full-custom layout of a version of RaPiD that does not include multipliers or memories. We also show the same RaPiD created by using our standard cell tool flow with both a generic standard cell library created by Tanner [42], and an FPGA optimized version of the Tanner library that had optimized muxes, demuxes, and flip-flops. The relative sizes of the layouts have been preserved and, as shown, the FPGA optimized

Tanner standard cell layout is approximately 20% smaller then the generic Tanner standard cell layout.

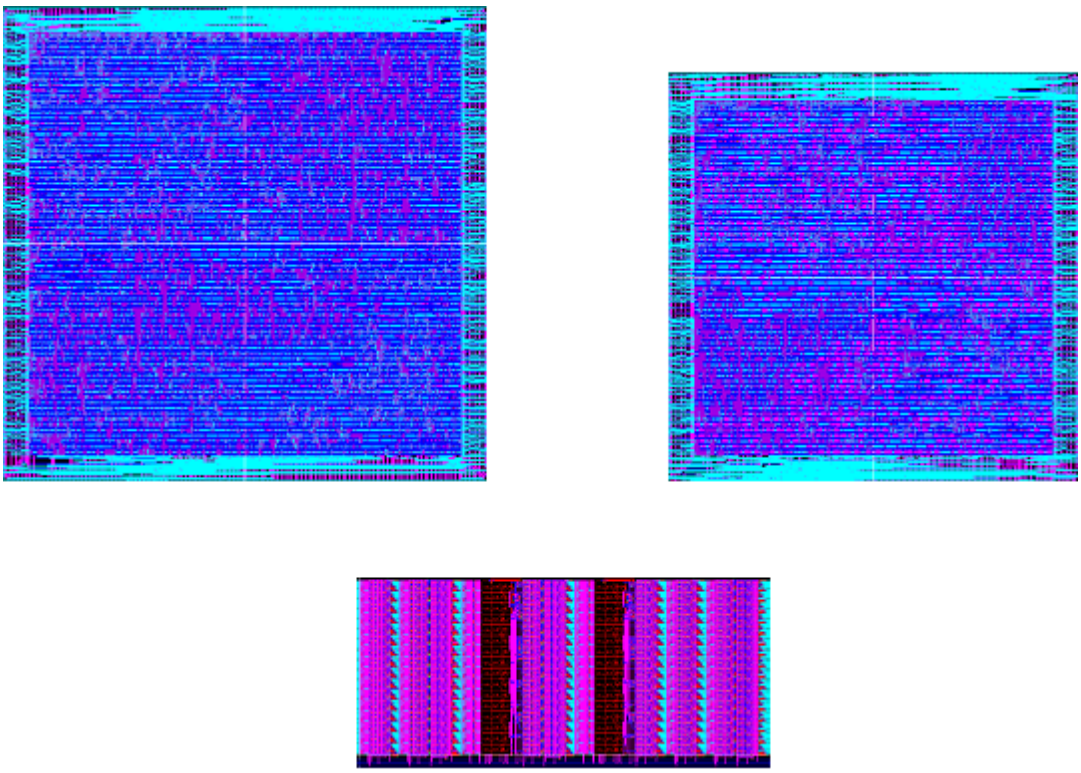


Figure 8-1: From previous work, the generic Tanner standard cell library (top left), FPGA optimized Tanner standard cell library (top right), and a full-custom RaPiD (bottom) that does not contain any multipliers or memories [5]. The relative size of the various layouts has been preserved.

8.1 Experimental Setup and Procedure

8.1.1 Setup

To retain as much flexibility as possible in our standard cell implementation, behavioral Verilog representations were created for all of the RaPiD components. The Architecture Generator used these behavioral components as leaf cells when it generated Verilog

versions of RaPiD that support a particular application domain. Synopsys was used to synthesize the behavioral Verilog to produce structural Verilog that has been mapped to our standard cell library [7]. This gives us the ability to swap out standard cell libraries, since we would only need to re-synthesize the behavioral Verilog with a new library file generated for the new standard cell library. The ability to easily and efficiently use different libraries is a very powerful feature of the Standard Cell Method. It enables designers to choose different libraries that provide different capabilities like lower power, smaller area, or higher performance.

Silicon Ensemble was used to place and route the cells. Silicon Ensemble is part of the Cadence Envisia Tool Suite, and is capable of routing multiple layers of metal, including routing over the cells. One powerful feature of Silicon Ensemble is its ability to run from macro files, minimizing the amount of user intervention.

Cadence was chosen as our schematic and layout editor because it is a very robust tool set that is widely used in industry [39]. Cadence also has tools for every aspect of the design flow. We are currently using the NCSU TSMC 0.18μm design rules for all layouts created in Cadence. As technology scales, we will be able to scale our layouts down with hopefully only minimal loss of quality in our results.

The full custom RaPiD components that were used in benchmarking were laid out by Carl Ebeling's group at the University of Washington for the RaPiD powertest, which is a version of RaPiD that does not contain any multipliers or memories. All circuits were laid out using the Magic Layout Editor for the HP 0.50μm process. The designs were

ported over to Cadence and the NCSU rules for the TSMC 0.25µm process, and then ported to the NCSU rules for the TSMC 0.18µm process. The port from the HP 0.50µm process did create many DRC errors that needed to be addressed, with the loss of some quality. Fortunately, the port from the NCSU rules for the TSMC 0.25µm process to the NCSU rules for the TSMC 0.18µm process required no changes to the layouts.

The choice of a standard cell library was based upon the need to find an industrial strength library that has been laid-out for the TSMC 0.18µm process. Unfortunately, we were not able to find a library targeted at the TSMC 0.18µm process, but we were able to find a library targeted at the TSMC 0.25µm process. This led us to the VTVT standard cell library, which was available from the Virginia Tech VLSI for Telecommunications group [58, 59]. This library has thirty-six basic blocks at its core. It also includes Synopsys synthesis files, VHDL simulation libraries, and LEF files for Silicon Ensemble.

8.1.2 Procedure

The tool flow shown in Figure 8-2 was used by the Standard Cell Method. The first step in the flow is the creation of behavioral Verilog architectures by the AML, AMO, and GH Architecture Generators that support the application domains. Synopsys was then used to synthesize this Verilog file to create a structural Verilog file that used the VTVT standard cells as modules. With this structural Verilog, Silicon Ensemble was able to then place and route the entire design. The utilization level of Silicon Ensemble, which is an indication of how dense cells are packed in the placement array, was increased until the design could not be routed. For most designs, this level was set to 90%. The aspect

ratio of the chip was also adjusted from 1, which is a square, to 2, which is a rectangle that is twice as long as it is high, to find the smallest layout. For all designs, an aspect ratio of 1 yielded the smallest layout. Once Silicon Ensemble was done creating the layout, the P&R tool was used to evaluate the quality of the circuits that were created, as discussed in Chapter 5.

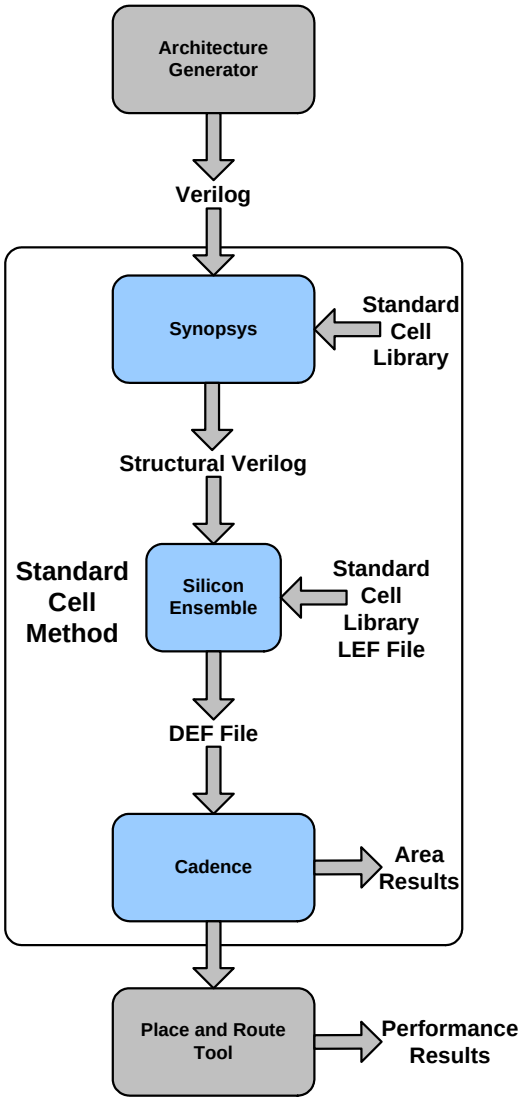


Figure 8-2: Tool flow for Standard Cell Method of architecture layout generation.

Table 8-1: The application domains and their respective percent utilization are shown. The area normalized to the RaPiD II tile of the VTVT standard cell architectures are shown. The architectures were created from Verilog that was generated by the AMO, AML, and the GH Architecture Generators.

Application Domain	Percent Utilization	Normalized Area			
		AML	AMO	GH	Average
Reduced RADAR 6	20.92	0.812	0.698	0.779	0.763
FIR	28.90	0.928	1.04	0.949	0.971
Reduced IMAGE 1	29.07	1.34	1.35	1.44	1.38
Reduced IMAGE 2	29.15	1.34	1.39	1.22	1.32
Sorters	32.12	1.26	1.15	1.20	1.20
Image	37.05	1.20	1.47	1.49	1.39
Matrix Multiply	37.43	1.40	1.46	1.34	1.40
Image and RADAR	41.21	1.34	1.23	1.27	1.28
Reduced RADAR 4	50.88	2.27	2.33	2.18	2.26
RADAR	52.79	2.52	2.36	2.23	2.37
Reduced IMAGE 4	52.82	2.53	2.31	2.23	2.36
Reduced RADAR 5	53.45	2.54	2.62	2.18	2.45
Reduced IMAGE 3	60.18	2.54	2.38	2.21	2.38
Average	---	1.70	1.68	1.59	1.65

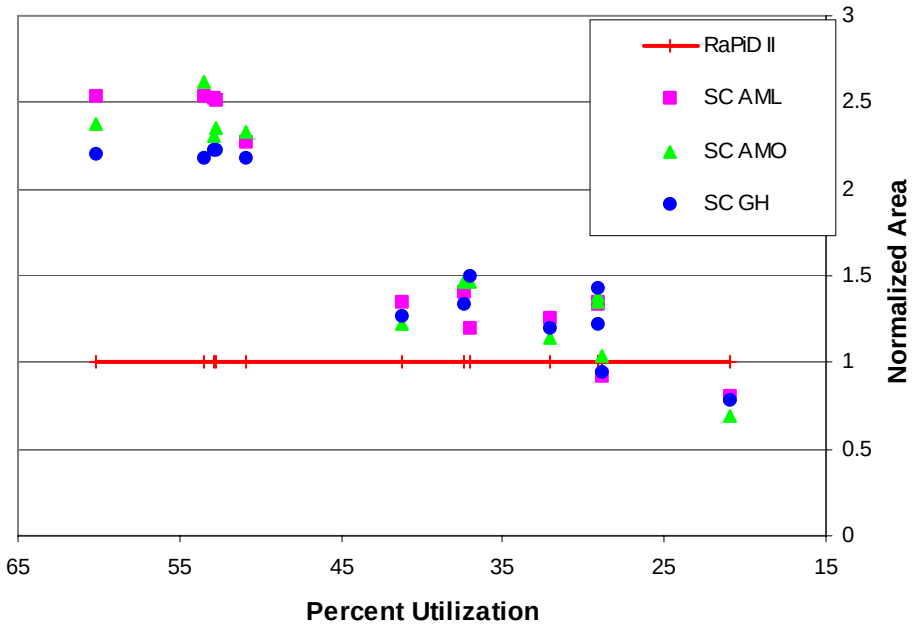
8.2 Results

The runtime of the entire tool flow to generate each template was on the order of hours to days. We generated results for the thirteen application domains, which were explained in detail in chapter 5, to evaluate the Standard Cell Method.

8.2.1 Area

The first metric we can use to evaluate the quality of the circuits that are generated by the Standard Cell Method is the area of each circuit. Table 8-1 and Graph 8-1 show the area of each benchmark set normalized to the area of the RaPiD II tile when using the generic VTVT standard cell library. In Graph 8-1, we also use data generated from all three

methods of the Architecture Generator. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile, with lower values representing smaller, and therefore more desirable, circuits. The x-axis represents the percentage of the fixed tile resources needed to support the benchmark set.



Graph 8-1: This graph, which was generated from the data presented in Table 8-1, shows the normalized area of each benchmark set when using the VTVT standard cell library. A lower value on the y-axis is preferable. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the application domain. The y-axis is the area of each application domain normalized to the RaPiD II fixed tile. The AMO, AML, and GH Architecture Generators were used to create the Verilog.

As mentioned above, in previous work [5], we have observed that when optimizing a standard cell library for FPGAs, gains of approximately 20% can be made when compared to a generic unaltered standard cell library. The optimizations included muxes, demuxes, and flip-flops. Therefore, the data in Table 8-1 has been reduced by 20%, and is presented in Table 8-2. Graph 8-2, was generated from the data in Table 8-2. This was

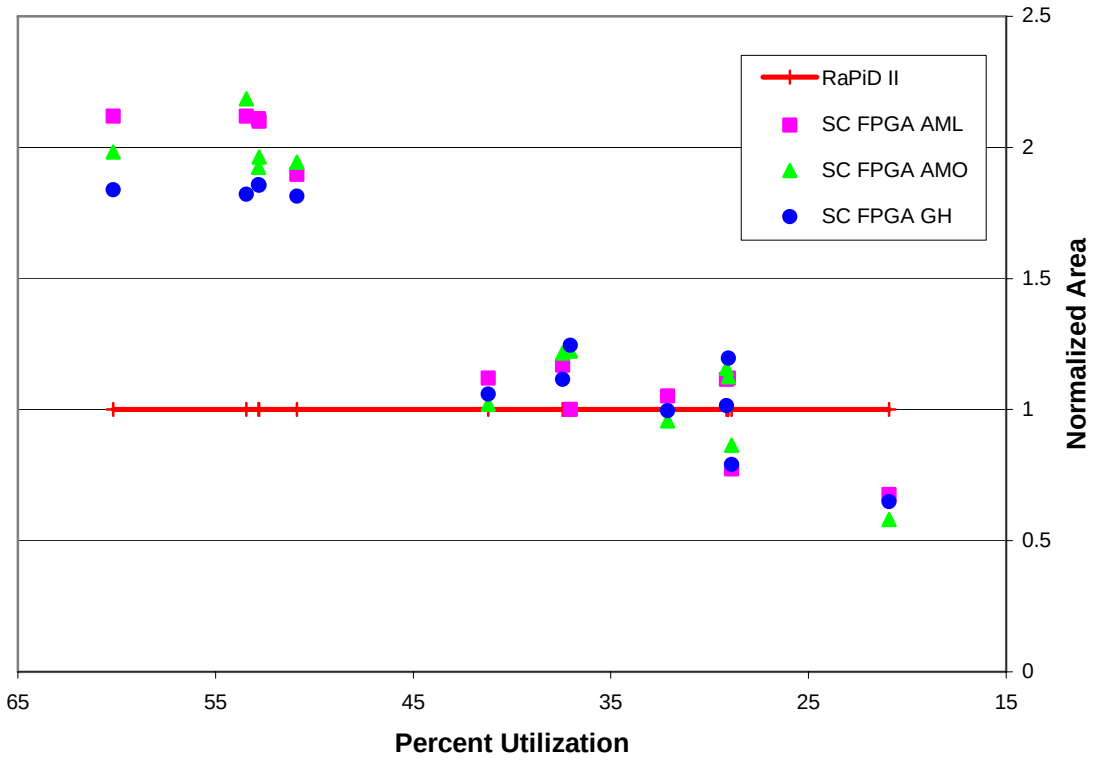
done in an attempt to estimate the possible area gains that may be made if the VTVT were optimized for use in generating FPGAs. Finally, in Graph 8-3, we see the average of the Standard Cell Methods using both the generic VTVT standard cell library and the estimated FPGA optimized VTVT standard cell library.

Table 8-2: The application domains and their respective percent utilization are shown. The area normalized to the RaPiD II tile of the FPGA optimized VTVT standard cell architectures are shown. The architectures were created from Verilog that was generated by the AMO, AML, and the GH Architecture Generators.

Application Domain	Percent Utilization	Normalized Area			
		AML	AMO	GH	Average
Reduced RADAR 6	20.92	0.677	0.582	0.650	0.636
FIR	28.90	0.773	0.864	0.791	0.809
Reduced IMAGE 1	29.07	1.12	1.13	1.20	1.15
Reduced IMAGE 2	29.15	1.12	1.16	1.02	1.10
Sorters	32.12	1.05	0.96	1.00	1.00
Image	37.05	1.00	1.22	1.25	1.16
Matrix Multiply	37.43	1.17	1.22	1.11	1.17
Image and RADAR	41.21	1.12	1.02	1.06	1.07
Reduced RADAR 4	50.88	1.90	1.94	1.81	1.88
RADAR	52.79	2.10	1.96	1.86	1.97
Reduced IMAGE 4	52.82	2.11	1.92	1.86	1.96
Reduced RADAR 5	53.45	2.12	2.19	1.82	2.04
Reduced IMAGE 3	60.18	2.12	1.98	1.84	1.98
Average	---	1.41	1.40	1.33	1.38

When comparing the three different methods used by the Architecture Generator, namely AML, AMO, and GH, we can see that the generated circuits are comparable, with almost identical results when percent utilization is low, as seen in Graph 8-1 and Graph 8-2. As percent utilization increase, it appears that GH performs better, followed by AMO and AML. These results do not reflect how much flexibility each of the different

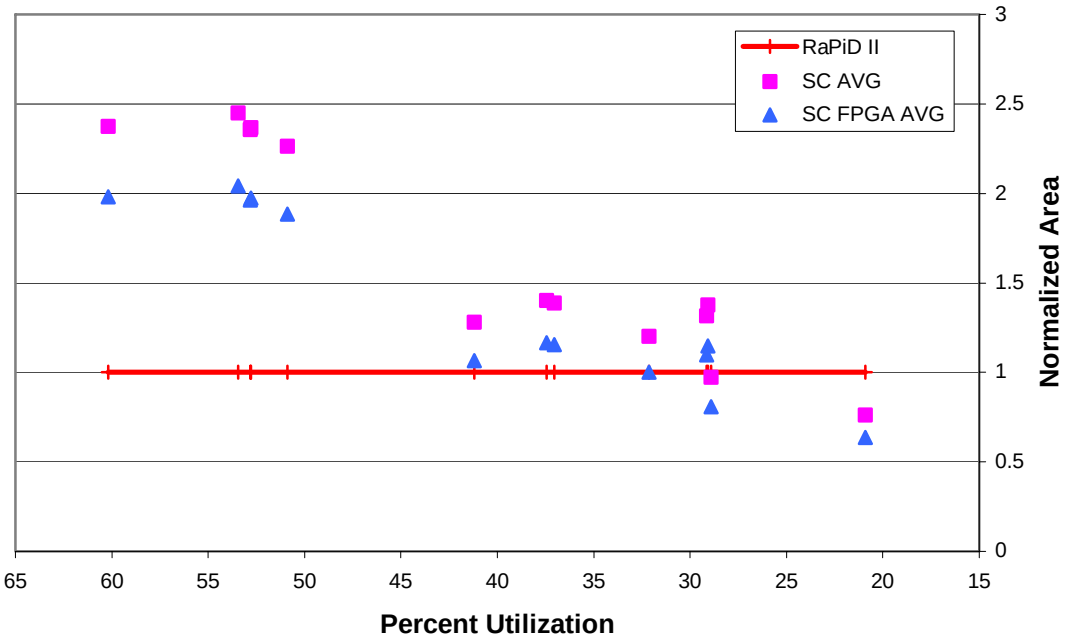
Architecture Generator methods produces. A flexibility analysis was performed in other work done for the Totem Project, and can be found at [8].



Graph 8-2: This graph, which was generated from the data presented in Table 8-2, shows the normalized area of each benchmark set when using the fitted lines representing the FPGA optimized VTVT standard cell library. A lower value on the y-axis is preferable. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the benchmark set. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile. The AMO, AML, and GH Architecture Generators were used to create the Verilog.

It can be observed from Graph 8-3 that circuits generated using the generic library are competitive with the full custom circuit when the percent utilization is at approximately 27%, and circuits generated using the FPGA optimized standard cell library when percent utilization is at approximately 33%. Therefore, the Standard Cell Method is able to

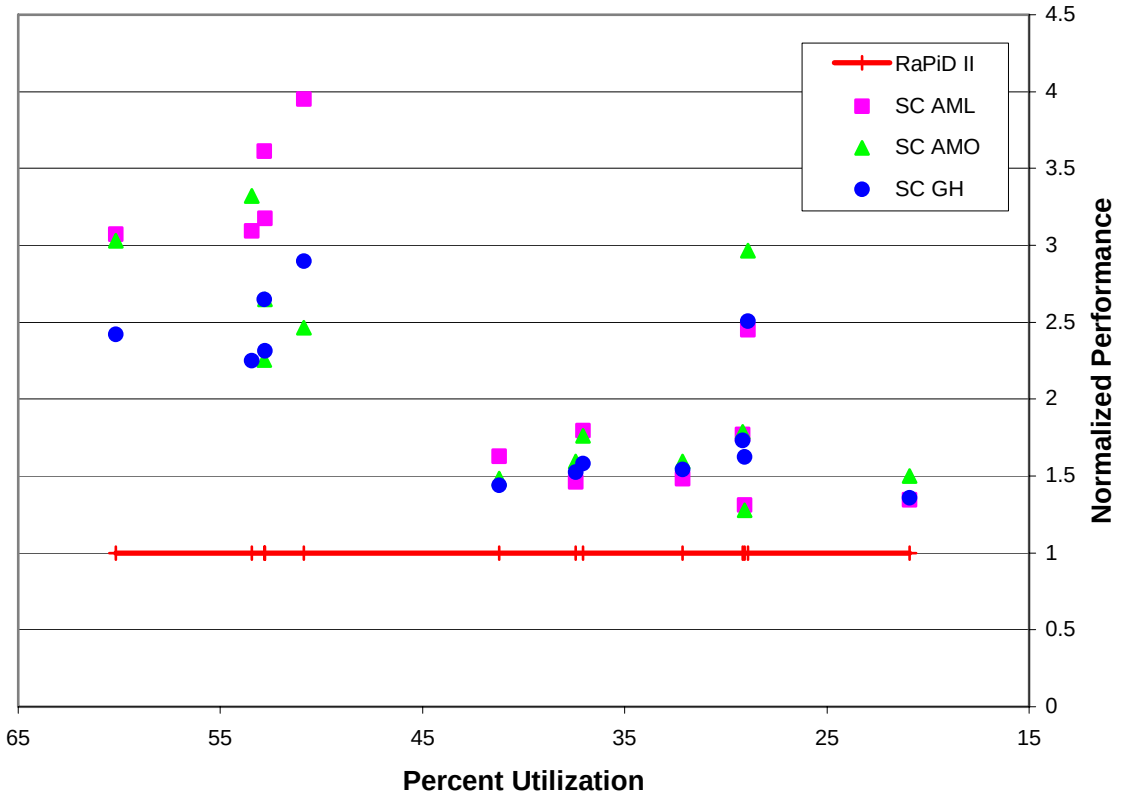
surpass the area of the full custom template when resources are reduced to about one third.



Graph 8-3: This graph shows the normalized area of each benchmark set, where a lower value on the y-axis is preferable. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the benchmark set. The y-axis is the area of each benchmark set normalized to the RaPiD II fixed tile. The SC AVG line represents the average of AMO, AML, and GH using the generic unaltered VTVT standard cell library, while SC FPGA AVG represents the average of AMO, AML, and GH using the fitted line representing the FPGA optimized VTVT standard cell library.

8.2.2 Performance

Graph 8-4 is very similar to Graph 8-1, but it shows the normalized performance of the various circuits that were generated by the Architecture Generator, as opposed to normalized area. All three methods of Architecture Generation are present in the graph, namely AML, AMO, and GH. The same clustering that occurs in Graph 8-1 is present, which is to be expected since the percent utilization metric is unchanged.



Graph 8-4: This graph shows the normalized performance of each benchmark set, where a lower value on the y-axis is preferable. The x-axis is the percentage of the resources of the fixed RaPiD II tile needed to support the benchmark set. The y-axis is the performance of each benchmark set normalized to the RaPiD II fixed tile. All three methods of Architecture Generation are present, namely AML, AMO, and GH.

Table 8-3: The normalized average performance of the circuits generated using the AML, AMO, and GH architecture generators for each application group.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Reduce RADAR 6	AML_decnsr	22.9	26.6	19.8	1.35	1.40
	AML_psd	30.4				
	AMO_decnsr	24.7	29.7	19.8	1.50	
	AMO_psd	34.7				
	GH_decnsr	21.1	26.9	19.8	1.36	
	GH_psd	32.6				

Table 8-3: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
FIR	AML_firsm2_unr	59.4	81.7	33.3	2.45	2.64
	AML_firsm3_unr	114.6				
	AML_firsm4_unr	59.4				
	AML_firsymeven_unr	93.3				
	AMO_firsm2_unr	59.8	98.8	33.3	2.96	
	AMO_firsm3_unr	154.0				
	AMO_firsm4_unr	59.8				
	AMO_firsymeven_unr	121.8				
	GH_firsm2_unr	57.9	83.6	33.3	2.51	
	GH_firsm3_unr	119.9				
	GH_firsm4_unr	57.9				
	GH_firsymeven_unr	98.8				
Reduce Image 1	AML_firtm_2nd	58.6	88.6	67.4	1.31	1.40
	AML_matmult_unr	118.5				
	AMO_firtm_2nd	60.8	86.0	67.4	1.28	
	AMO_matmult_unr	111.2				
	GH_firtm_2nd	57.6	109.6	67.4	1.63	
	GH_matmult_unr	161.6				
Reduce Image 2	AML_1d_dct40	162.6	155.7	88.0	1.77	1.76
	AML_fft16_2nd	192.3				
	AML_matmult_unr	112.1				
	AMO_1d_dct40	141.6	157.4	88.0	1.79	
	AMO_fft16_2nd	216.3				
	AMO_matmult_unr	114.3				
	GH_1d_dct40	119.0	152.3	88.0	1.73	
	GH_fft16_2nd	222.5				
	GH_matmult_unr	115.5				
Sorters	AML_sort_2d_g	52.9	50.7	34.2	1.48	1.54
	AML_sort_2d_rb	51.8				
	AML_sort_g	48.4				
	AML_sort_rb	49.5				
	AMO_sort_2d_g	63.1	54.5	34.2	1.59	
	AMO_sort_2d_rb	53.3				
	AMO_sort_g	52.4				
	AMO_sort_rb	49.3				
	GH_sort_2d_g	59.3	52.7	34.2	1.54	
	GH_sort_2d_rb	52.6				
	GH_sort_g	49.4				
	GH_sort_rb	49.4				

Table 8-3: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Image	AML_1d_dct40	157.5	143.5	79.9	1.79	1.71
	AML_fft16_2nd	217.7				
	AML_firtm_2nd	68.6				
	AML_matmult_unr	130.2				
	AMO_1d_dct40	109.5	140.7	79.9	1.76	
	AMO_fft16_2nd	242.3				
	AMO_firtm_2nd	75.5				
	AMO_matmult_unr	135.4				
	GH_1d_dct40	135.4	126.3	79.9	1.58	
	GH_fft16_2nd	178.5				
	GH_firtm_2nd	73.6				
	GH_matmult_unr	117.8				
Matmult	AML_limited_unr	116.1	114.8	78.6	1.46	1.53
	AML_matmult4_unr	114.2				
	AML_matmult_unr	116.6				
	AML_vector_unr	112.2				
	AMO_limited_unr	121.9	125.4	78.6	1.60	
	AMO_matmult4_unr	131.5				
	AMO_matmult_unr	129.6				
	AMO_vector_unr	118.6				
	GH_limited_unr	121.9	119.8	78.6	1.53	
	GH_matmult4_unr	127.5				
	GH_matmult_unr	116.1				
	GH_vector_unr	113.8				

Table 8-3: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Image and RADAR	AML_decnsr	129.8	130.6	80.1	1.63	1.52
	AML_psd	144.0				
	AML_1d_dct40	146.4				
	AML_fft16_2nd	205.7				
	AML_firtm_2nd	42.6				
	AML_matmult_unr	115.4				
	AMO_decnsr	103.5	118.8	80.1	1.48	
	AMO_psd	114.7				
	AMO_1d_dct40	113.7				
	AMO_fft16_2nd	219.6				
	AMO_firtm_2nd	54.9				
	AMO_matmult_unr	106.1				
	GH_decnsr	106.9	115.3	80.1	1.44	
	GH_psd	124.8				
	GH_1d_dct40	123.5				
	GH_fft16_2nd	160.3				
	GH_firtm_2nd	62.8				
	GH_matmult_unr	113.3				
Reduce RADAR 4	AML_decnsr	86.9	119.5	30.2	3.95	3.10
	AML_fft16_2nd	152.1				
	AMO_decnsr	59.8	74.5	30.2	2.46	
	AMO_fft16_2nd	89.2				
	GH_1d_dct40	57.3	87.7	30.2	2.90	
	GH_fft16_2nd	118.0				
RADAR	AML_decnsr	69.0	92.2	29.0	3.18	2.71
	AML_fft16_2nd	142.3				
	AML_psd	65.2				
	AMO_decnsr	51.0	76.8	29.0	2.65	
	AMO_fft16_2nd	116.2				
	AMO_psd	63.2				
	GH_decnsr	47.0	67.1	29.0	2.31	
	GH_fft16_2nd	94.2				
	GH_psd	60.2				

Table 8-3: Continued.

Application	Netlists	Performance (ns)	Average (ns)	RaPiD II	Normalized	Normalized Average
Reduce Image 4	AML_1d_dct40	86.9	119.5	33.1	3.61	2.84
	AML_fft16_2nd	152.1				
	AMO_1d_dct40	59.8	74.5	33.1	2.25	
	AMO_fft16_2nd	89.2				
	GH_decnsr	57.3	87.7	33.1	2.65	
	GH_fft16_2nd	118.0				
Reduce RADAR 5	AML_fft16_2nd	141.7	101.9	32.9	3.10	2.89
	AML_psd	62.2				
	AMO_fft16_2nd	154.4	109.3	32.9	3.32	
	AMO_psd	64.2				
	GH_fft16_2nd	91.7	74.0	32.9	2.25	
	GH_psd	56.4				
Reduce Image 3	AML_1d_dct40	66.9	82.4	26.8	3.07	2.84
	AML_fft16_2nd	149.0				
	AML_firtm_2nd	31.3				
	AMO_1d_dct40	77.5	81.2	26.8	3.03	
	AMO_fft16_2nd	138.4				
	AMO_firtm_2nd	27.8				
	GH_1d_dct40	63.0	64.9	26.8	2.42	
	GH_fft16_2nd	95.3				
	GH_firtm_2nd	36.4				

The most obvious result that can be seen from Graph 8-4 is the fact that none of the netlists mapped onto circuits that are generated by the Standard Cell Method perform better than the same netlists mapped onto the RaPiD II template. Generating circuits using the Standard Cell Method introduces too much overhead, which cannot be overcome even when the percent utilization is at the lowest possible value for the benchmarks that we have. One possible solution to this problem is to try to use other types of standard cell libraries. A standard cell library that is targeted at high performance may produce better quality results. It should be noted that we did not it

performance numbers for the an FPGA optimized version of the VTVT standard cell library, since we have found in previous work [5] that there is no benefit to performance when using FPGA optimized standard cell libraries instead of generic standard cell libraries. It should be noted that the place and route tool is unable to retime signals, as mentioned in Chapter 5. Therefore, any performance numbers generated by the place and route tool should only be used for relative comparisons of the three methods.

8.3 Summary

In this chapter, we have shown that automation of layout generation for domain specific FPGAs is possible using a Standard Cell Method. We have further shown that as a target application domain narrows (requiring less resources), the savings gained from removing unused logic from a design enables the Standard Cell Method of layout generation to approach that of a full custom layout in area, and in some cases surpass it. With respect to performance, circuits created by the Standard Cell Method perform close to full custom circuits, but do not surpass them. The Standard Cell Method is capable of producing circuits with areas ranging from 2.45 times larger to 0.76 times smaller than comparable full custom circuits, as shown in Table 8-1. The Standard Cell Method can produce circuits with performances raging from 3.10 times to 1.40 times slower than comparable full custom circuits, as shown in Table 8-3. In addition, by adding to a standard cell library a few key cells that are used extensively in FPGAs, improvements of approximately 20% can be achieved with regard to area, as shown in Table 8-2, while the impact to performance is negligible. It should be noted that the greatest strength of this

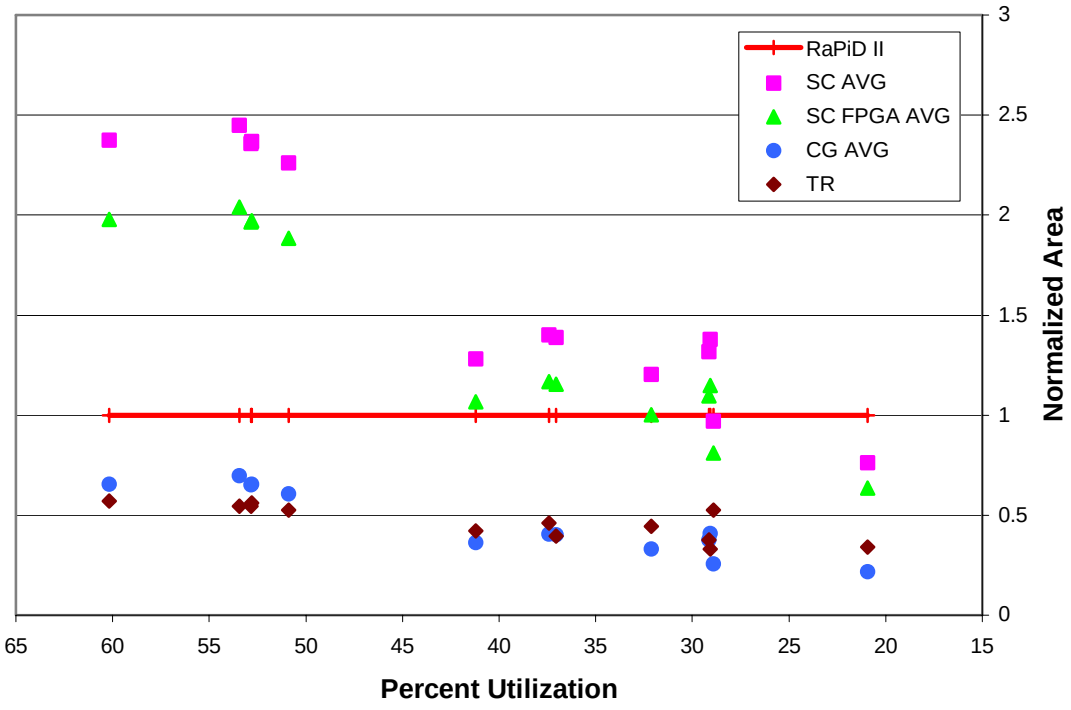
method is its high level of flexibility. This method is always capable of producing a result, even when the other methods fail. Finally, with a wider range of libraries, including libraries optimized for power, performance, and area, this method has a lot of potential for improvement.

Chapter 9

Comparison and Contrast of the Methods

It is evident that no single method is able to produce architectures that meet a designer's constraints in all cases. Template Reduction works well when a specified architecture is a subset of an existing full-custom template, and thus is able to leverage the benefits of full-custom design. However, the Template Reduction Method fails completely if the specified architecture is not a subset of the available templates. The Standard Cell Method, on the other hand, can support any arbitrary FPGA design, even though circuits created by this method have decreased performance and an area penalty when compared to full-custom designs. Thus, the Standard Cell Method is able to support architectures that require more resources than any available template, which can occur with fixed templates, as was shown in Chapter 6 with regard to the RaPiD-I architecture and the image processing application group. The Circuit Generator Method is able to complement the Template Reduction and Standard Cell Methods by creating more efficient circuits than the Standard Cell Method is capable of producing, while possessing more flexibility than the Template Reduction Method.

9.1 Area Comparison: FC, TR, SC, and CG



Graph 9-1: Area comparison of the circuits created to support the benchmark sets, using the Template Reduction Method, the Circuit Generator Method, and the Standard Cell Method. The y-axis is the area of the circuits normalized to the area of the full custom RaPiD II template, while the x-axis is the percent utilization. Lower values represent circuits that are more desirable. The SC AVG, SC FPGA AVG, and the CG AVG are lines generated by taking the average of the circuits generated by the AML, AMO, and GH versions of the Architecture Generator.

The area of the circuits created varies greatly depending on both the specified application domain, and the proposed method. The graph shown in Graph 9-1 presents the three methods along with the original full custom RaPiD II tile. As we have seen in Chapters 6, 7, and 8, the x-axis is percent utilization, which is an indication of the amount of resources that an application domain would require to run on the full custom RaPiD II template. The y-axis is the area normalized to the full custom RaPiD II template. The

points for the Circuit Generator and the Standard Cell Methods are an average of the AML, AMO, and GH Architecture Generators.

As mentioned in Chapter 5, the RaPiD II template does fail on the Camera, Transforms, and the OFDM application domains, which is a slight improvement over the RaPiD I template, which fails on the Camera, Image Processing, Transforms, and the OFDM application domains. Therefore, the Camera, Transforms, and OFDM application domains have been removed form the application domains used to compare the various methods. Both templates are failing on these application domains because they are routing constrained, which is a fundamental architectural limitation that was inherited with the full custom layouts that were provided by the RaPiD group. This inherited limitation was because the height of the RaPiD array was fixed by the functional units. When using the HP .50μm 3 metal layer process, this constrained the number of routing tracks at 14. Moving to the TSMC .18μm 6 metal layer process allowed us to change the number of routing tracks to 24. Unfortunately, 24 routing tracks is still not enough routing for the Camera, Transforms, and the OFDM application domains.

It is evident from Graph 9-1 that the Template Reduction and the Circuit Generator Methods create circuits that are comparable to each other in area. The Template Reduction Method is more efficient when the percent utilization is high, while the Circuit Generator Method is more efficient when the percent utilization is lower. This is a strong showing for the Circuit Generator Method, since it is creating circuits from scratch that can compete with reduced full custom circuits. These results may be an indication that

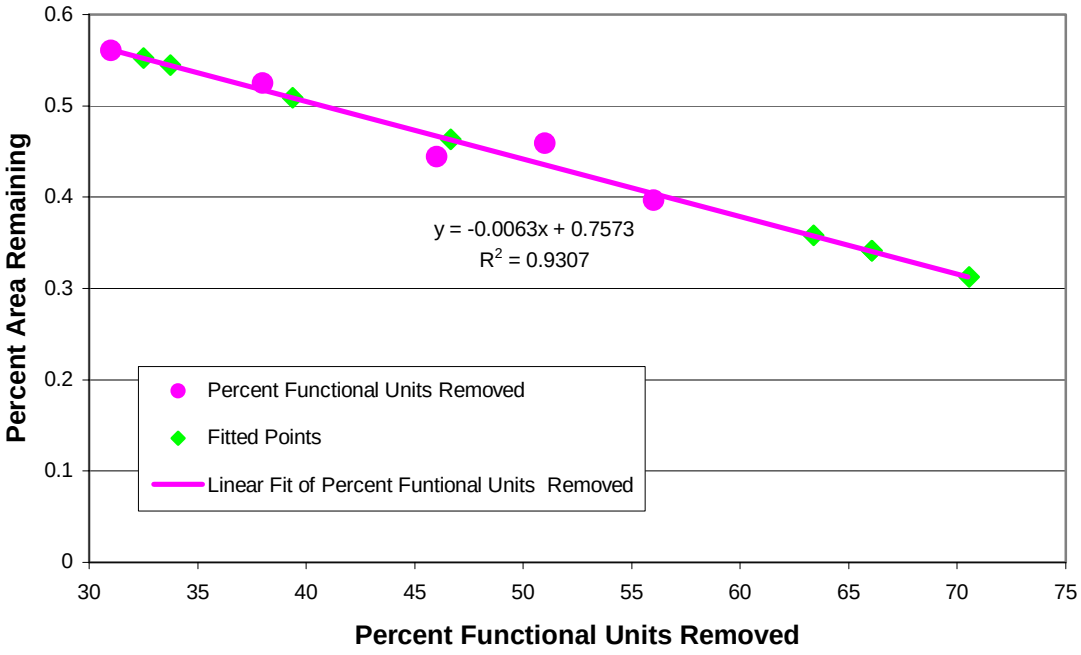
the compaction of circuits is less efficient as the percent utilization drops. This is because the circuits created by the Template Reduction Method are becoming less and less regular. The Circuit Generator Method is not affected by this, since it is creating circuits from the ground up, as opposed to reducing existing structures.

Table 9-1: This table shows the percent functional units removed for all thirteen application domains, which were found by the Template Reduction Method. For five of the application domains the area remaining was found by running the Cadence compactor on the reduced layouts. The area remaining was found for the other eight application groups, whose values are in the gray cells, by using a fitted linear line to the original five data points.

Application Domain	Percent Functional Units Removed	Area Remaining (y = -0.0063x + 0.7573)	Percent Utilization
Reduced RADAR 6	66.25	0.340	20.92
FIR	38.00	0.525	28.90
Reduced Image 1	67.50	0.332	29.07
Reduced Image 2	60.62	0.375	29.15
Sorters	46.00	0.444	32.12
Image	56.00	0.396	37.05
Matrix Multiply	51.00	0.459	37.43
Image and RADAR	53.33	0.421	41.21
Reduced RADAR 4	36.61	0.527	50.88
RADAR	31.00	0.561	52.79
Reduced Image 4	33.93	0.544	52.82
Reduced RADAR 5	33.93	0.544	53.45
Reduced Image 3	29.46	0.572	60.18

It should also be noted that, as mentioned in Chapter 6, only five of the thirteen application domains were run on the Template Reduction Method namely: FIR, image, matrix multiply, RADAR, and sorters. The other eight application domains were fit, by using the relation of remaining functional units to remaining area, which is strongly correlated, with a correlation constant of 0.931. This relationship is shown in Table 9-1 and Graph 9-2. The reason why we did not run the Template Reduction Method on the

additional eight application domains was that the new version of the Cadence compactor failed to run on the additional eight application domains. However, this did not preclude us from running the first stage of the Template Reduction Method on these application domains, which was able to obtain the percent of the functional units that were removed from the RaPiD II tile in support of these application domains. We then used this data and the fitted line, shown in Graph 9-2, to find the percent area remaining for the additional eight application domains.

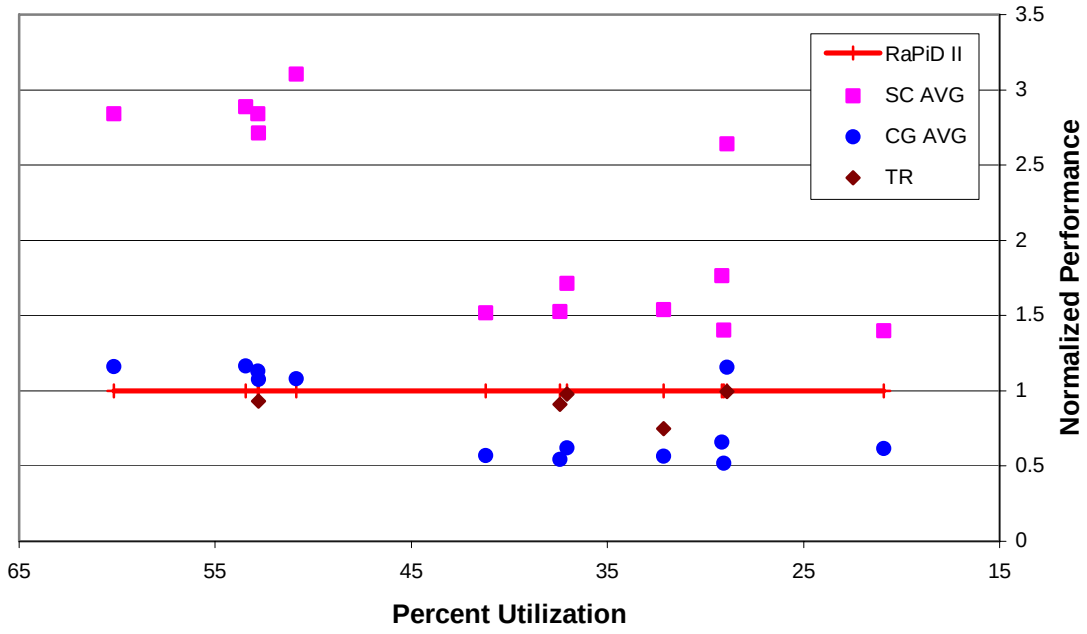


Graph 9-2: This graph shows the results of the Template Reduction Method on the original five application domains. The x-axis is the percent of the functional units removed, while the y-axis is the percent of the template area remaining after compaction. The fitted points are the other eight application domains, all of which were run through the Template Reduction Method, but not compacted. Performing the Template Reduction Method without compaction still yields the percentage of functional units removed, enabling these points to be fitted.

The Standard Cell Method suffers from a large amount of inherent overhead that leads to the creation of circuits that are approximately 2 times larger than those created by the Template Reduction and Circuit Generator Methods. These results do not show the flexibility that the Standard Cell Method has at its disposal. If either, or both, of the other methods fail, the Standard Cell Method is positioned to handle any of these circuits. It should also be noted that the current library we have is not an industrial strength library that is targeted at low area designs. By using a low area industrial strength library, circuits created by the Standard Cell Method would probably be of much better quality. In essence, the Standard Cell Method is poised for gains if industrial libraries are used.

9.2 Performance Comparison: FC, TR, SC, and CG

The performance of the application domains on the circuits created by the various methods, shown in Graph 9-3, are more scattered and do not show the same level of improvement as the area improvements. The Standard Cell Method cannot overcome the overhead associated with this method. Therefore, the circuits created by the Standard Cell Method never perform better than the full-custom RaPiD II tile, and are approximately 3.10 times to 1.40 times slower than the other three methods. The shortcomings of the Standard Cell Method are even more magnified when it is pointed out that the full-custom RaPiD II tile is unaltered, and therefore capable of handling application domains that require 100% utilization, while the circuits generated by the Standard Cell Method have been reduced, and are therefore less capable.



Graph 9-3: Performance comparison of the benchmarks run on the full-custom RaPiD II tile, and the Template Reduced, the Circuit Generator, and the Standard Cell Methods. The y-axis is the performance normalized to the RaPiD II cell, while the x-axis is the percent utilization. Lower y-values are preferable. The CG AVG and the SC AVG are lines generated by taking the average of the circuits generated by the AML, AMO, and GH versions of the Architecture Generator.

The Circuit Generator and the Template Reduction Methods produce circuits that have an average improvement of approximately 16% to 9% in performance over the benchmarks run on the full custom RaPiD II tile. When the percent utilization is high, the Template Reduction Method appears able to produce higher performing circuits than the Circuit Generator Method. When the percent utilization is low, the Circuit Generator Method is able to produce circuits that perform better than the Template Reduction Method. It should be noted that, as mentioned in Chapter 6, only five of the thirteen application domains are run on the circuits created by the Template Reduction Method. The application domains are the FIR, image, matrix multiply, RADAR, and sorters. This lack

of data points with regard to the Template Reduction Method makes these conclusions weak, which may change if more data is collected for the Template Reduction Method. It should be noted that the place and route tool is unable to retime signals, as mentioned in Chapter 5. Therefore, any performance numbers generated by the place and route tool should only be used for relative comparisons of the three methods.

Chapter 10

Conclusions and Future Work

When designing SOCs, a unique opportunity exists to add custom reconfigurable devices, which will provide an efficient compromise between the flexibility of software and the performance of hardware, while at the same time allowing for post-fabrication modification of circuits. Unfortunately, the high cost in both time and design effort of creating unique reconfigurable devices for each and every possible application domain would outweigh any benefit gained. Therefore, automation of the design flow is required if these new custom architectures are to be designed in a timely fashion. The goal of the Totem Project is to provide an automated design flow for the creation of application specific reconfigurable devices for SOCs.

10.1 Contributions

The focus of this work has been the automation of the layout portion of the Totem design flow. Towards this end, we have implemented the VLSI layout generator, which automates the creation of mask ready layouts from the circuit descriptions provided by the Architecture generator. The VLSI layout generator consists of three methods of automating the layout process: Template Reduction, Circuit Generators, and Standard Cell generation.

The Template Reduction Method was introduced in Chapter 6. This method is able to leverage full custom designs, while still removing any resources that are not needed to support the specified application domain. This enables the Template Reduction Method to create circuits that perform at or better than that of the initial full-custom template, with an average area decrease of approximately 48% and an average performance increase of approximately 9%.

One of the drawbacks associated with the Template Reduction Method is its reliance on the existence of a feature-rich macro cell that is a superset of the specified application domain. The Circuit Generator Method, detailed in Chapter 7, is able to produce efficient circuits in both area and performance in an additive fashion, while removing the need for feature-rich templates. Circuits created by the Circuit Generator Method are approximately 46% smaller and 16% faster than the full custom RaPiD II tile.

The Standard Cell Method, presented in Chapter 8, while extremely flexible, was able to produce competitive circuits with regard to area, only when the resources were reduced to approximately 25% of the full-custom template. Unfortunately, the Standard Cell Method was never able to produce a circuit that performed better than the full-custom RaPiD II template. The Standard Cell Method is capable of producing circuits with areas ranging from 2.45 times larger to 0.76 times smaller than comparable full-custom circuits. The Standard Cell Method can also produce circuits with performances ranging from 3.10 times to 1.40 times slower than comparable full-custom circuits. These numbers do not highlight the fact that the strength of the Standard Cell Method lies in its

ability to produce a circuit for any application domain, even when the Template Reduction and Circuit Generator Methods fail.

10.2 Conclusions and Future Work

Choosing an appropriate method is based on many factors. If a robust template along with suitable reductions exists, then the Template Reduction Method is quite capable of producing competitive circuits. While this suggests that the Template Reduction Method should be competitive with the other methods, we feel that it is the weakest method. The Template Reduction Method is too inflexible. Its reliance on templates is its biggest liability, since the generation of even one template is a costly endeavor. In future versions of the Totem Project, the designer will be able to specify certain design constraints, like low power, small area, and high performance. This implies that a single template will not be able to cover all of these design areas, forcing the Totem Project to have at its disposal multiple templates that have been laid out with different design goals in mind. This would entail considerable effort, thus defeating the purpose of the Totem Project to automatically provide custom reconfigurable circuits in a timely fashion.

Another problem with the Template Reduction Method is its propensity for causing errors in circuits. Of the three methods, the Template Reduction Method was the most error prone method, and was the most complicated method to implement and debug. In essence, the Template Reduction Method is manipulating full-custom circuits at the lowest level. This can lead to numerous DRC errors, including n-implant, p-implant, and well errors. In addition, when manipulating full-custom designs in this manner, the

Template Reduction Method is changing the dynamics of the circuits in potentially unforeseen ways. For example, the transistors in a full-custom circuit are sized to ensure that they are capable of driving their load in an efficient manner. By cutting out transistors, wires, etc, the Template Reduction Method is altering those loads, which can lead to a poorly performing circuit.

The Circuit Generator Method is able to leverage the regularity that exists in FPGAs when creating RaPiD-like structures. It can create structures that are more efficient than the Template Reduction Method, while not being bound to a particular template. In addition, the Circuit Generator Method is an additive method. Therefore, this method is less error prone than the Template Reduction Method since we are not cutting low-level components out of full-custom circuits.

However, the Circuit Generator Method has problems of its own. The creation of a wide range of generators can be as costly a proposition as creating a wide range of templates. But, to improve the Circuit Generator Method, providing a wide range of different types of generators is critical. To increase the quality of the circuits that the method creates, all of the generators should be able to handle a wide range of parameters. For example, in the current implementation of the Circuit Generator Method, the generators that create the functional units are unable to change the bit width of units that they create. However, it has been shown in previous work [5] that the largest impact on area is achieved through reducing the overall bit-width of the device that is created. Therefore, the creation of generators that are able to modify the bit width of the functional units could drastically

increase the ability of the Circuit Generator Method to create higher quality circuits. Finally, if the Circuit Generator Method needs to create circuits that are targeted at low power, high performance, or small area, even more types of generators will be needed.

This leads us to the Standard Cell Method. As anticipated, the Standard Cell Method has inherent inefficiencies that it must overcome to become competitive with the other two methods. However, it is extremely flexible and is able to create a circuit in any circumstance. This is important because the overall goal of the Totem Project is to support any designer defined application domain. To build upon this flexibility, the ability to utilize a wide range of industrial strength standard cell libraries is needed. With a wide range of libraries, the designer could select the library most suited to the specifications of their design. Specifications could include higher performance, lower power, or smaller area, and if there was a corresponding library, the Standard Cell Method has the potential to create high quality circuits with a minimal amount of effort.

Although the results presented in this work indicate that the Standard Cell Method performed poorly when compared to the other two methods, many gains can be made by optimizing this method. There are numerous ways that this method can be optimized. Not only is an industrial strength standard cell library essential, but, to ensure efficient designs, a high quality P&R tool is also a necessity. Additionally, macro blocks of functional units including the multiplier, ALU, bus interconnect, etc can be created with standard cells. To ensure high quality macro blocks, a synthesis tool can be used that has been tuned to the standard cell library. Finally, floorplanning and routing of the macro

blocks can be performed to yield high quality circuits. In essence, by tuning the Standard Cell Method, we feel that it is capable of producing circuits that perform at the level of the other two methods, while requiring less effort to implement. Therefore, this method should be pursued over the other two methods.

The initial version of the Totem Project creates reconfigurable devices that are based upon the RaPiD family of architectures. RaPiD arrays are coarse grained in nature and only utilize a one-dimensional routing interconnect. While there are benefits to these types of coarse-grained devices as detailed in the RaPiD section of Chapter 2, industry is dominated by medium-grained island-style devices that utilize a two-dimensional routing interconnect. Therefore, future versions of the Totem Project should be capable of creating medium-grained island-style devices that have a two-dimensional routing interconnect. This will enable accurate comparisons between structures created by the Totem Project and their more generic multipurpose FPGA counterparts present in industry. In addition, it will enable the Totem Project to create a wider range of devices that are capable of supporting a larger design space.

The automatic generation of medium-grained island-style devices is more difficult due to the two-dimensional routing interconnect. However, by using an optimized version of the Standard Cell Method, these difficulties should be alleviated. By creating two-dimensional standard cell macro blocks of all of the main components, and floorplanning and routing them, the Standard Cell Method is capable of producing high quality circuits. Therefore, the only difference between generating the current version of RaPiD-like

devices and two-dimensional island-style devices involves the creation of their constituent macro blocks. The floorplanner should be able to efficiently handle either type of circuit, once these blocks have been created. In effect, by optimizing the Standard Cell Method, including the synthesis of the verilog, the P&R tool, and by using macro blocks, the Standard Cell Method should be able to efficiently handle either type of architecture.

Bibliography

[1] Glökler, Tilman. “System-on-a-Chip Case Study: ADSL-Receiver.” <http://www.ert.rwth-aachen.de/Projekte/VLSI/soc.html> (30 January 2004).

[2] C. Ebeling, D. C. Cronquist, and P. Franklin. “RaPiD - reconfigurable pipelined datapath.” In *Proc. 6th Int. Workshop on Field Programmable Logic and Applications (FPL)*, volume 1142 of *Lecture Notes in Computer Science*, pages 126-135. Springer-Verlag, 1996.

[3] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. “PipeRench: A Reconfigurable Architecture and Compiler”, *IEEE Computer*, 33(4):70-77, April 2000.

[4] A. Abnous and J. Rabaey. “Ultra-low-power domain-specific multimedia processors.” In *VLSI Signal Processing, IX*, pages 461-470. IEEE Signal Processing Society, 1996.

[5] S. Phillips, “Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip.” M.S. Thesis, Northwestern University, Dept. of ECE, July 2001.

[6] A. Sharma, “Development of a Place and Route Tool for the RaPiD Architecture.” M.S. Thesis, University of Washington, Dept. of EE, 2001.

[7] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation." *IEEE Symposium on FPGAs for Custom Computing Machines Conference*, 2001.

[8] K. Compton, A. Sharma, S. Phillips, and S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field Programmable Logic and Applications*, pp. 59-68, 2002.

[9] S. Phillips and S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 165-173, 2002.

[10] A. Sharma, C. Ebeling, and S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 68-77, 2003.

[11] Neil H. E. Weste and Kamran Eshraghian, Principles of CMOS VLSI Design: A Systems Perspective, Addison-Wesely Publishing Company, 1993.

[12] C. Sechen, VLSI Placement and Global Routing Using Simulated Annealing, Kluwer Academic Publishers, Boston, MA: 1988.

[13] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Seventh International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997.

[14] Larry McMurchie and Carl Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995

[15] Thomas H.Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, Prim's algorithm, pp 505-510, 1990.

[16] Altera Corporation, “Corporate Overview”, <http://www.altera.com/corporate/overview/ovr-index.html> (30 January 2004).

[17] Lattice Semiconductor, “Lattice Corporate Information”, <http://www.latticesemi.com/corporate/index.cfm> (30 January 2004).

[18] QuickLogic, “Corporate Information”, <http://www.quicklogic.com/home.asp?PageID=191&sMenuID=88> (30 January 2004).

[19] Xilinx Inc., “About Xilinx”, <http://www.xilinx.com/company/about.htm> (30 January 2004).

[20] Atmel, “Corporate Overview”, <http://www.atmel.com/corporate/corporate_profile.asp> (30 January 2004).

[21] Elixent, “About Elixent”, <http://www.elixent.com/corporate/proposition.htm> (30 January 2004).

[23] Triscend, “Triscend Company Info Overview”, <http://www.triscend.com/companyinfo/> (30 January 2004).

[24] Xilinx, “DS110: Virtex-II Pro X Platform FPGAs Complete Data Sheet”, version 1.0, November 17, 2003.

[25] IBM, “PowerNP Npe405H Embedded Processor”, October 2002.

[26] Altera Corporation, “APEX II Programmable Logic Device Family Data Sheet”, version 3.0, August 2002.

[27] Altera Corporation, “HardCopy Device Handbook, Volume 1”, version 1.1, August 2003.

[28] Lattice Semiconductor, “ORCA ORLI10G Data Sheet”, November 2003.

[29] QuickLogic, “QuickMIPS: Enabling System Flexibility, Performance and Customization in a Single Piece of Silicon”, revision Rev. A, April 2003.

[30] Atmel, “AT94K Series FPSLIC Data Sheet”, Rev. 1138F-FPSLI-06/02, June 2002.

[31] Elixent, “Applications of D-Fabrix”, 2003.

[33] Actel Corporation, “VariCore™ Embedded Programmability - Flexible by Design”, <http://varicore.actel.com/cgi-bin/varicore.cgi?page=overview> (30 January 2004).

[34] Advanced Products: Introducing LiquidLogic Embedded Programmable Logic Core, <http://www.lsilogic.com/products/asic/advanced_products.html> (30 January 2004).

[35] P. Hallschmid, S.J.E. Wilton, “Detailed Routing Architectures for Embedded Programmable Logic IP Cores”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[36] Anthony Cataldo, “Startup stakes out ground between FPGAs and ASICs” <http://www.eetimes.com/story/OEG20010709S0075>, 2001. (30 January 2004).

[37] eASIC, <http://www.easic.com/products/easicore018.html> (30 January 2004).

[38] K. Compton, *Programming Architectures for Run-Time Reconfigurable Systems*, M.S. Thesis, Northwestern University, Dept. of ECE, December, 1999.

[39] Cadence Design Systems, Inc., “Openbook”, version 4.1, release IC 4.4.5, 1999.

[40] D. C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling, "Specifying and Compiling Applications for RaPiD", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1998.

[41] Synopsys, Inc., “Synopsys Online Documentation”, version 2000.05, 2000.

[42] Tanner Research, Inc., “Tanner CES Products”, <http://www.tanner.com/CES/products/files_now/dit_std_cell.htm> (30 January 2004).

[43] Synopsys, Inc, “Epic Tools User Manual”

[44] Synopsys, Inc., “Pathmill User Guide”, release 5.4, 2000.

[45] Xilinx Virtex-4 Family Overview, DS112 (v1.1) September 10, 2004

[46] Altera Inc., San Jose California. *Cyclone™II Device Handbook, Volume 1*, June 2004.

[47] Kerry A. Dolan. Forbes. New York: Jan 12, 2004. Vol. 173, Issue. 1; p. 156a

[48] D. C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling, “Specifying and Compiling Applications for RaPiD”, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[49] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, “Architecture Design of Reconfigurable Pipelined Datapaths”, *Twentieth Anniversary Conference on Advanced Re-search in VLSI*, 1999.

[50] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Presentation at Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.

[51] Kathrine Compton, “*Architecture Generation of Customized Reconfigurable Hardware*”, Ph.D. Thesis, Northwestern University, Dept. of ECE, 2003.

[52] C. Sechen, VLSI Placement and Global Routing Using Simulated Annealing. Kluwer Academic Publishers, Boston, MA, 1988.

[53] “IBM, Xilinx shake up art of chip design with new custom product”, <http://www-3.ibm.com/chips/news/2002/0624_xilinx.html>, (15 August 2004).

[54] Company Fact Sheet, <http://www.xilinx.com/company/press/fctsheet.htm>, (15 August 2004)

[55] Triscend Corporation, Triscend A7S Configurable System-on-Chip Platform: Product Description. Triscend Corporation, Mountain View, CA, 2002.

[56] Triscend Corporation, Triscend E5 Customizable Microcontroller Platform: Product Description. Triscend Corporation, Mountain View, CA, 2002.

[57] David Maliniak, “Basics of FPGA Design”, Mentor Graphics White Paper, April 1, 2004

[58] J. B. Sulistyo, J. Perry, and D. S. Ha, "Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules", Department of Electrical and Computer Engineering, Virginia Tech, Technical Report VISC-2003-01, November 2003.

[59] Jos. B. Sulistyo and Dong S. Ha, "A New Characterization Method for Delay and Power Dissipation of Standard Library Cells", VLSI Design **15** (3), pp. 667-678, 2002.

[60] Shaffer, Stanaski, Glaser, and Franzon, “The NCSU Design Kit for IC Fabrication through MOSIS", 1998 International Cadence User Group Conference in Austin, Texas.

[61] Avant! Star-Hspice Manual, Release 1998.2, July 1998

[62] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. ACM Computing Surveys, 34(2):171–210, June 2002.

[63] Shawn Phillips, Akshay Sharma, and Scott Hauck. “Automating the Layout of Reconfigurable Subsystems Via Template Reduction”. I*nternational Symposium on Field-Programmable Logic and Applications*, pp. 857-861, 2004.

[64] D. Johannsen, “Bristle Blocks: A silicon compiler”, *Proc. 16th Design Automation Conf.*, 1979.

[65] G. Kedem, F. Brglez, K. Kozminski, “OASIS: A silicon compiler for semi-custom design”, *Euro ASIC ’90*, pp. 118-123, 1990.

Vita

Education

Ph.D., Electrical Engineering, University of Washington, Seattle, WA. 2004.

Advisor: Scott Hauck, GPA 3.96.

Thesis: "Automatic Layout of Reconfigurable Subsystems for Systems-on-a Chip"

M.S., Electrical and Computer Engineering, Northwestern University, Evanston, IL, 2000.

Advisor: Scott Hauck, GPA 3.63.

Thesis: "Automatic Layout of Domain-Specific Reconfigurable Subsystem for Systems-on-Chip"

B.S., Magna Cum Laude, Physics and Math, Youngstown State University (YSU), Youngstown, OH, 1998.

Minors: Computer Science and Chemistry, GPA 3.51.

Research Interests/Skills

FPGA Architectures, Applications, and Tools; Computer Architecture; VLSI Design; Reconfigurable Computing; Operating System Integration of Reconfigurable Computing.

Use of Cadence Tool Suite; fluent in various programming languages (Cadence SKILL, Perl, Perl-Tk, C and Unix Script); extensive use of Synopsys Design Analyzer; use of Silicon Ensemble (including the ability to create custom automation scripts).

Awards

MIT Lincoln Laboratories Fellowship, University of Washington.

Walter P. Murphy Fellowship, Northwestern University.

Myron C. Wick Jr. Scholarship for academic excellence in the physical sciences, YSU.

Albert Gurrieri Scholarship for academic achievement in Physics, YSU.

Vice President of the Society of Physics Students, YSU Chapter.

Employment

Annapolis Micro Systems, Inc., Annapolis, MD.

10/2004 – present DSP Applications Design Engineer.

University of Washington, Seattle, WA.

09/1999 – 09/2004 Research Assistant. Developed systems for the automatic layout of reconfigurable subsystems for SOC as part of ACME Labs Totem Project. System administrator for ACME Labs during the critical transition from Illinois to Washington.

Northwestern University, Evanston, IL.

09/1998 – 09/1999 Research Assistant. Floating Point Arithmetic on Field Programmable Gate Arrays (FPGAs). Developed a system for the automatic layout of reconfigurable subsystems for SOC as part of ACME Labs Totem Project.

Picker International, Cleveland, OH.

05/1996 – 09/1998 Software Integration Tester/Field Service Engineer. Regression testing and programming for the PQ2000 CT Scan System. Repaired and maintained x-ray processors in the medical industry.

Youngstown State University, Youngstown, OH.

09/1993 – 06/1998 Research Assistant. Project: Fast beam studies of Rydberg H2. Built a high energy CO2 laser. Built a high voltage power supply for an ion source. Machined and built a vacuum system for an ion beam source.

Publications/Presentations

S. Phillips, A. Sharma, S. Hauck. “Automating the Layout of Reconfigurable Subsystems Via Template Reduction”. International Symposium on Field-Programmable Logic and Applications, pp. 857-861, 2004.

S. Phillips, A. Sharma, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Via Template Reduction", poster at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA; April 2004.

S. Phillips, A. Sharma, K. Compton, S. Hauck, “Automatic Layout for the Totem Project,” invited oral presentation, MIT Lincoln Laboratory; Boston, MA; July 2003.

K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", International Symposium on Field Programmable Logic and Applications, pp. 59-68, Montpellier, France; September 2002.

S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, pp. 165-173, Monterey, CA; February 2002.

S. Phillips, Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip, M.S. Thesis, Northwestern University, Dept. of ECE, Chicago, IL; July 2001.

Personal

Born January 31, 1975. US citizen.

Actively train and compete in Brazilian Ju Jitsu form of martial arts, May 2001-present.

Volunteer as a science fair judge, seasonal.

Volunteered as a counselor and lifeguard for orthopedically handicapped children, seasonal.