Automatic Layout of Domain Specific Reconfigurable Subsystems for System-on-a-Chip (SOC)

Master's Thesis
July 2001

Shawn A. Phillips
Northwestern University

As submitted to
Northwestern University
Department of Electrical and Computer Engineering
Evanston, IL USA

Advisor
Scott A. Hauck
University of Washington
Seattle, WA USA

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF GRAPHS

TABLE OF TABLES

Abstract

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 FPGA architectures, 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 FPGA architecture would make it impossible to meet any reasonable design cycle, while adding considerably to design costs. Therefore, we present a system that automates the layout of reconfigurable subsystems for systems-on -a-chip.

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, 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. But in the current SOC design methodology framework, in which only 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 FPGAs onto the SOC. Unlike application specific integrated circuits (ASICs), by including

FPGAs, designers would gain the ability to alter the SOC to meet differing system requirements after the SOC has been fabricated. However, FPGAs are often several times slower and larger and less energy efficient than ASICs, making them a less ideal choice for high performance, low power designs. Domain-specific FPGAs can be utilized to bridge the gap that exists between flexible FPGAs and high performance ASICs.

A domain-specific FPGA is a reconfigurable array that is targeted at specific application domains, instead of the multiple domains a reconfigurable processor targets. Creating custom domain-specific FPGAs 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. With this knowledge, 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 reconfigurable processors within this space. This ability to utilize custom arrays instead of ASICs in high performance SOC designs will provide the post-fabrication flexibility that FPGAs provide, while also meeting stringent performance requirements that until now could only be met by ASICs.

Possible application domains could include signal processing, cryptography, image analysis, or any other computationally intensive area. In essence, the more that is known about the target applications, the more inflexible and ASIC-like the custom array will be. On the other end of the spectrum, if the domain space is only vaguely known, then the custom array 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 FPGA.

Since all of the components in an SOC need to be fabricated after integration, this provides designers with a unique opportunity to insert application specific FPGAs into their devices. Unfortunately, if designers were forced to create custom logic for each domain-specific subsystem, it would be impossible to meet any reasonable design cycle. However, by automating the generation of the application specific FPGAs, designers would avoid this increased time to market and would also decrease the overall design cost.

These factors have led us to start the Totem project, which has the ultimate goal of automatically generating custom reconfigurable architectures based upon the perceived application domain in which the device will be used. Since the custom array will be optimized for a particular application domain, we expect that it will have a smaller area and perform better then a standard FPGA, while retaining all of the benefits of reconfigurability.

First we present a short background on reconfigurable computing, including RaPiD and Totem. Next, we provide a look at other work whose goal is to provide reconfigurability on SOC devices. We then examine the approach and experimental setup that I have taken to automate the layout of a domain-specific reconfigurable subsystem. Finally, I will show how my approach was able to create circuits that perform within Totem's design specifications, paving the way for future work in providing custom reconfigurable subsystems in SOCs.

2. Background

The processing devices that the general public is most familiar with fall into the general-purpose processor category. General-purpose processors are able to run a wide range of applications since they are based upon a fixed, well-defined instruction set. This computational model provides general-purpose processors with the ability to support basic computations in hardware, while running anything else the processor may encounter in software. Unfortunately, this great flexibility comes at the price of reduced performance, since software solutions perform poorer than hardware solutions.

When application requirements demand higher performance than a general-purpose processor can provide designers are forced to turn to hardware. An ASIC is a circuit that is designed to perform one application in hardware and to perform it well. ASICs are typically able to outperform general-purpose processors by orders of magnitude. But all of this computational power comes at the price of inflexibility, greatly increased design costs, and a very long design cycle. This has created a market atmosphere such that, unless a design will be mass-produced, an ASIC solution is just too costly to implement.

Reconfigurable computing is intended to fill the gap between the flexible and ubiquitous general-purpose processor and the expensive yet high performing ASIC. To retain its flexibility, the FPGA must sacrifice area and performance, losing to ASICs in both cases. Yet, FPGAs are still capable of performance that is several orders of magnitude better than general-purpose processors on computationally intensive applications. In addition, due to the fact that most commercial FPGAs are massed produced, it may be cheaper to implement an FPGA in place of an ASIC in certain design situations, such as a limited production design run.

2.1. FPGA

An FPGA is a multi-level logic device that can be reconfigured. Conceptually, an island-style FPGA, which is the most common type of FPGA manufactured today, is an array of configurable logic blocks (CLBs) that are embedded within horizontal and vertical channels of routing [5]. The routing channels also have the capability to be personalized to interconnect any one CLB with any other CLB within the array by the use of switchboxes. 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 programming of the FPGA is entirely electrical, providing the FPGA with 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 how many CLBs are in the array and by the richness of the routing fabric. Figure 2-1 shows an island-style FPGA, which was first developed by Xilinx in 1984 [6].

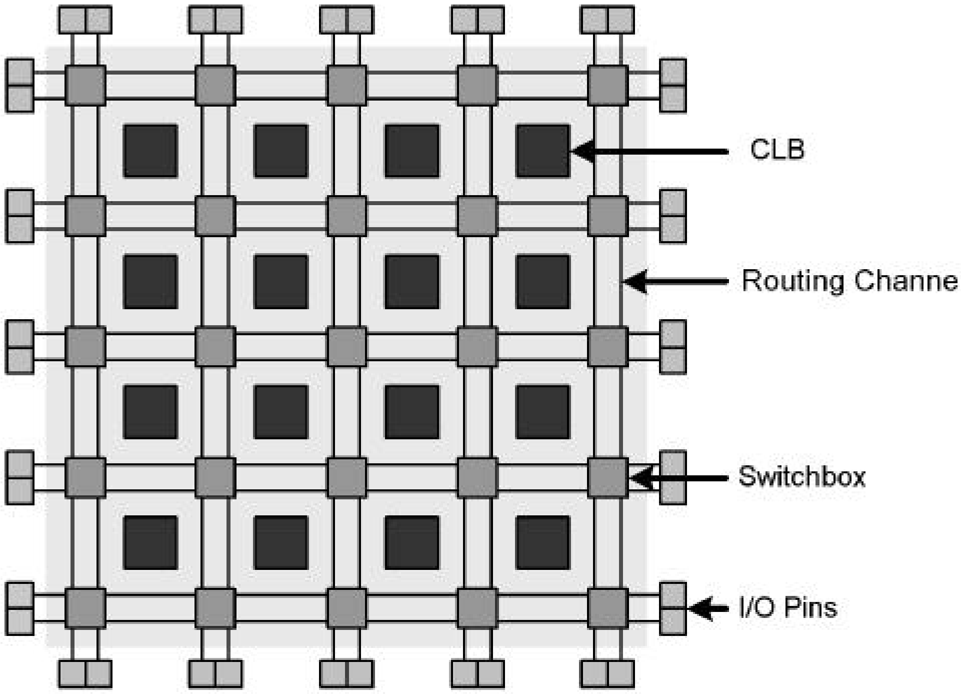


Figure 2-1: Xilinx style FPGA architecture. It contains an array of CLBs, switchboxes, and vertical and horizontal routing channels.

In an island-style FPGA, logic functions are typically implemented by means of multiplexers that have their control and possibly their data inputs connected to programming bits. By controlling these muxes with the programming bits, different functional blocks within the CLB can be chosen, depending on the application that is being mapped onto the FPGA. Logic functions can also be computed in a CLB by lookup-tables (LUTs), which are small memories that can be written to when the FPGA is configured. Through the use of muxes and LUTs, FPGAs are capable of implementing arbitrary logic functions, which have usually no more then 3 or 4 inputs.

2.2. RaPiD

We are using the reconfigurable-pipelined datapath (RaPiD) architecture as a starting point for the circuits that we will be generating [2]. RaPiD is positioned between standard FPGAs and ASICs. Its goal is to provide the performance of an ASIC while maintaining reconfigurability. RaPiD, like an FPGA, achieves reconfigurability through the use of block components such as memories, adders, multipliers, and pipeline registers. But, unlike a commercial FPGA, RaPiD is not targeted at random logic, but at coarse-grained, computationally intensive functions like arithmetic.

Also, RaPiD utilizes a one-dimensional structure to take advantage of the fact that all of its functional components are word-width computational devices. 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, leveraging all of the research into the compilation of algorithms onto systolic arrays. Finally, while a two dimensional RaPiD is possible, most two-dimensional algorithms can be mapped onto a one-dimensional array through the use of memory elements.

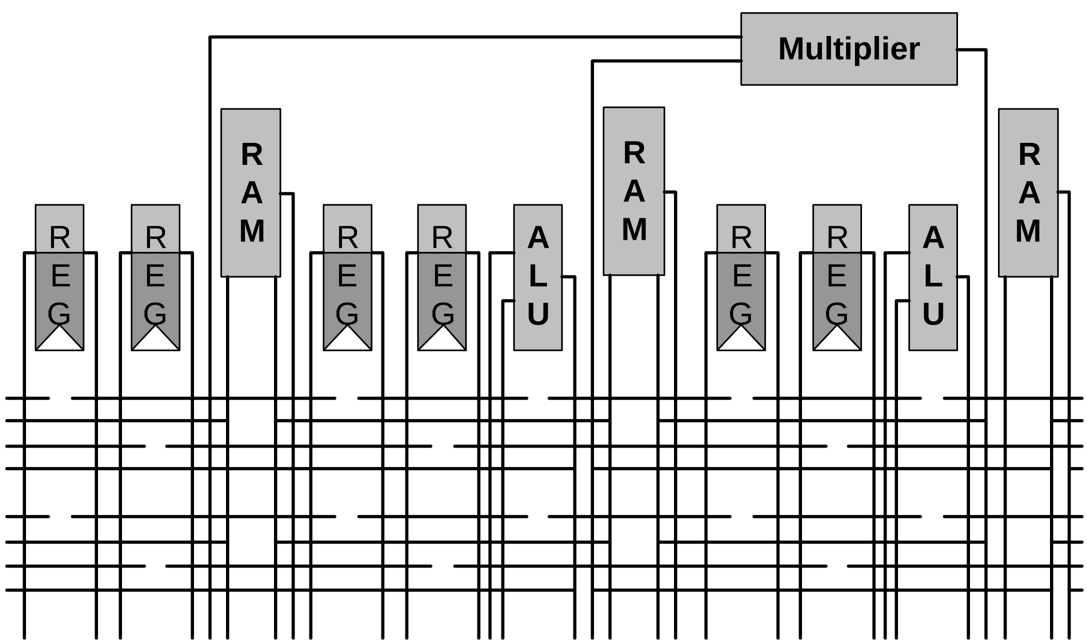


Figure 2-2: A block diagram of a basic RaPiD cell. Multiple cells are tiled along a one-dimensional horizontal axis.

The version of RaPiD that we are benchmarking against, RaPiD I, consists of memories, ALUs, and multipliers that are all 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. To create different versions of RaPiD that target different application domains, the following changes need to be made to the array: modify existing or add new functional units, change the width of the buses or the number of buses present, and modify the routing fabric. Figure 2-2 shows an example of one possible RaPiD cell. Multiple cells are tiled along the horizontal axis.

2.3. Totem

Reconfigurable hardware is a very efficient bridge that fills the gap between software implementations running on general-purpose processors and ASIC implementations. But, 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 each and 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. Figure 2-3 shows the dependencies between the major parts of Totem.

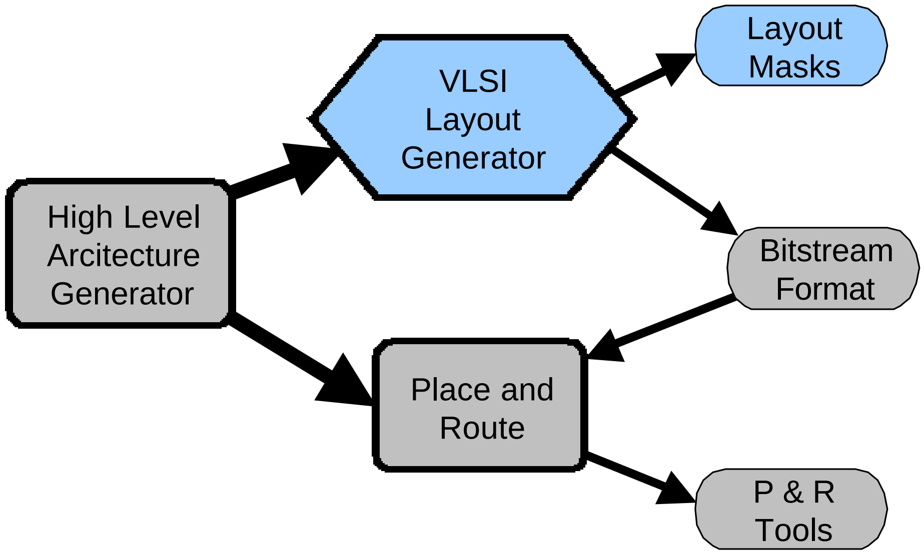


Figure 2-3: Totem dependency graph

The high-level architecture generator will receive, as input from the designer, information about the set of applications that the SOC will target. The architecture generator will then create a coarse-grained FPGA that consists of block components such as memories, adders, multipliers, and pipeline registers [7]. Routing resources will then connect these components to create a one-dimensional structure. Extending Totem to create two-dimensional arrays of functional units is possible, and will be explored in future research.

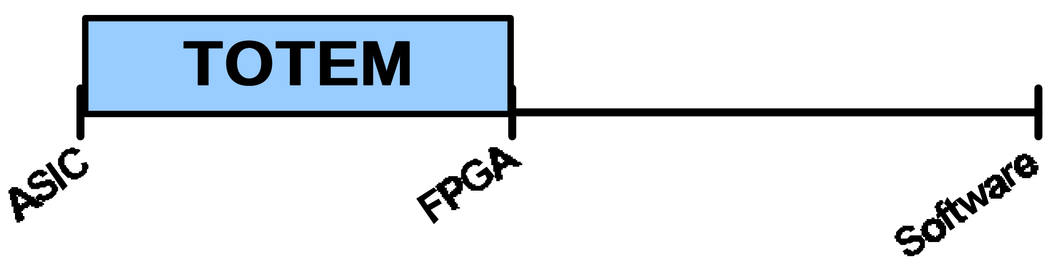


Figure 2-4: The Totem Project. Totem creates structures that fall between ASICs and FPGAs.

The final structure that is created will fall somewhere on the scale between ASICs and commercial FPGAs. Where on this scale the final device will fall depends on how much information the designer is able to provide in advance about the applications that will run on the reconfigurable hardware, and how similar those applications are in composition. If the designer can provide a lot of information, and the applications are similar in composition, then more hardware and connection points can be removed from the FPGA, thus generating a more compact and higher performing design. On the other side of the scale, if the designer does not know which applications will run on the SOC, or the applications that will run on the SOC are very different, then more reconfigurable components will be needed to support a wider range of logic, causing the final design to be more like a commercial FPGA. After the high-level architecture generator creates an architecture that is able to support the applications that will run on it, this information is disseminated to both the VLSI layout generator and the place and route tool generator.

The VLSI layout generator has the task of creating a fabrication-ready layout for the custom device by using the specifications that were provided by the high-level architecture generator. The layout generator will be able to create a layout for any possible architecture that the high-level architecture generator is capable of producing. The difficult task for the layout generator is creating efficient designs, so as not to squander any performance and area gains that the architecture generator was able to achieve over general-purpose FPGAs. The layout generator will also need to be flexible enough to change over time to take advantage of smaller device sizes as process technology scales down. In addition, the layout generator will need to produce a bit-stream format that the place and route tools will be able to use to configure the custom FPGA. Three different methods are being explored to automate the layout process: template reduction, standard cells, and FPGA-specific circuit generators. Each of these will be discussed in greater detail later in this thesis.

Once the custom architecture is created, the end user will then need a tool set that will automatically generate mappings that target the custom array. The place and route tool generator will create a physical mapping of a user application by using an architecture description that was created by the high-level architecture generator and the bit-stream format that was created by the VLSI layout generator. It does this task by the use of a placement tool, which is based on the simulated annealing algorithm, and a router that is an adaptation of the Pathfinder algorithm.

3. Prior Work

As the semiconductor industry moves to multi-million gate designs, the need to integrate reconfigurable logic onto SOCs is being addressed by more and more companies. Most of the approaches that have been proposed so far include embedding current "off-the-shelf" FPGA designs onto the SOC. A few companies, though, are creating application specific reconfigurable cores, but they still do not have a more flexible standard cell implementation to serve as a safety net in case an application domain does not map onto an available template. The following is a sampling of the projects on which industry and academia are currently working.

3.1. Xilinx

Xilinx is still sitting on the fence concerning the use of FPGA blocks in SOCs, but that stance is shifting. As of June 22nd, Xilinx is in talks with ASIC vendors about licensing reconfigurable blocks [8]. One obvious candidate to begin embedding Xilinx's FPGA architectures into SOCs is IBM Microelectronics, since a cross-licensing agreement between the two companies already exists. It appears that the FPGAs blocks that will be provided by Xilinx will be "off-the-shelf" Xilinx cores.

3.2. Actel

Actel has taken a much more aggressive attitude towards embedding FPGA blocks in SOCs, and is currently offering numerous cores to its ASIC partners through its VariCore™ program. Actel's drive into the embedded FPGA market began on June 5th, 2000 when it acquired intellectual property (IP) innovator, Prosys Technology. To further strengthen their presence in the SOC market, on May 21st, 2001 Actel created a partnership with LogicVision to help solve the increasingly complex challenges of SOC design and test [9].

VariCore™ logic contains highly efficient embedded FPGA cores (EPGA™). EPGA™ cores are fabrics of reconfigurable logic that are embedded throughout the SOC, upon which different applications are mapped. The use of EPGA™ cores is akin to the "off the shelf" FPGA approach, but with the ability to partition the EPGA™ where needed [10].

3.3. LSI & Adaptive Silicon

On June 12th, 2001, after two years of speculation, LSI Logic Corp, with partners Adaptive Silicon and Ericsson, finally began providing details about its LiquidLogic™ programmable core for use in SOCs [11]. LiquidLogic™ will enable designers to create a partition on an SOC that can be reprogrammed later to meet changing standards or customer specifications. Currently, LiquidLogic™ is only targeted at third-generation (3G) wireless applications, with consumer electronics, printers, and storage-area-networks as future targets. LSI has produced cores on its G12 0.18-micron process, with its Gflex 0.13-micron process due next year.

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 [12]. 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, CALU are tiled in 16x16 arrays called hex blocks, with the maximum size being a 4x4 array of hex blocks. Refer to Figure 3-1 for an example of a MSA. Once the reconfigurable logic is in place, various soft-cores can be downloaded to it. This approach, like the one proposed by Actel, consists of tiling existing FPGA cells onto an SOC.

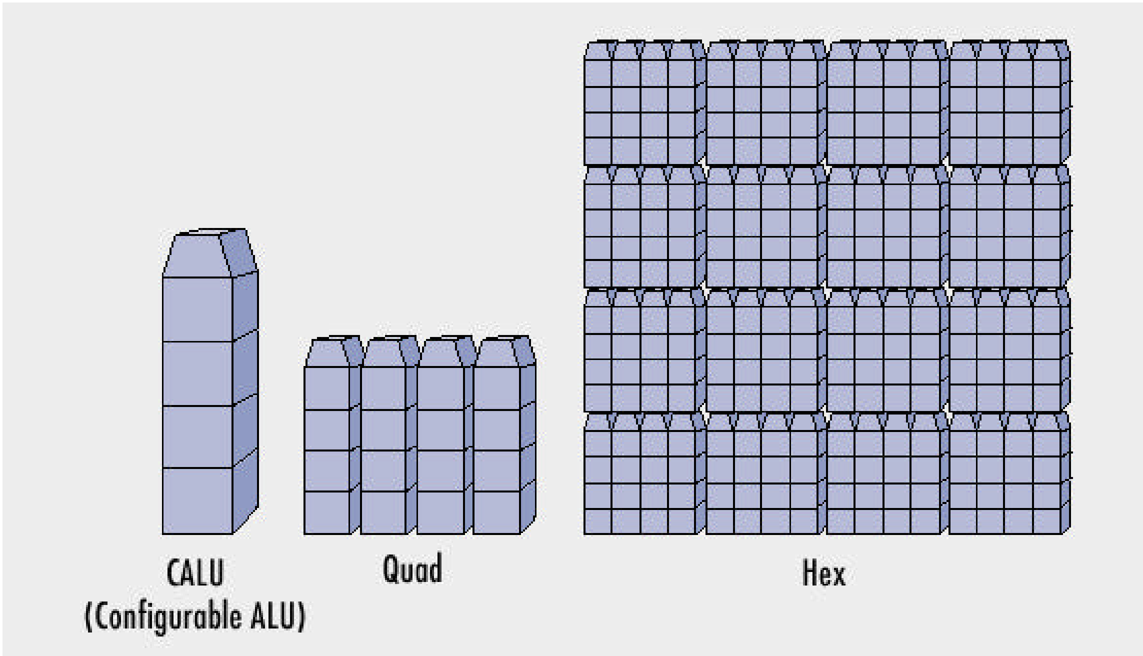


Figure 3-1: Configurable ALU (CALU) [12].

3.4. eASIC

eASIC has an interesting approach to provide limited reconfigurability in an SOC. Through the use of an ultra-dense cell architecture that is based on commercial FPGAs, eASIC is capable of producing circuits that have a gate density that is as high as twenty times that of a conventional FPGA [13][14]. To achieve this density, eASIC has shifted all routing form 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, and at the loss of logic density, some interconnect reconfigurability can be retained.

3.5. U. B. C. SOC Research Group

The University of British Columbia's SOC Research Group is currently exploring how to construct these massive and complex chips. One problem that the UBC SOC Research Group has focused on is how to create reconfigurable subsystems on SOCs that have a variety of shapes and sizes [15]. Through the use of a new switch box, the area penalty for a design with an aspect ratio of 2:1 has been reduced to 1.6% and the performance penalty has been reduce to only 1.1%, when compared to a design that has a square core. This relatively minor area and performance penalty pave the way for cores that are radically different then the typical square cores of today's FPGAs. This will help ease the integration of reconfigurable subsystems into the design flow of SOCs.

4. Approach

Current design methodologies for the layout of circuits typically fall under either full-custom design or standard-cells, with both of these approaches having associated pros and cons. Producing a full-custom circuit is a labor-intensive task, which requires a very long and expensive design cycle. However, the resulting circuit that is created is usually the fastest and the smallest that is possible at that time. Generating a standard cell library can be a difficult endeavor, and therefore, companies that have extensive libraries vigorously guard them from competitors. However, once the library is created, the ability for indefinite reuse and design automation justifies both the time and expense involved. Unfortunately, circuits that are created using standard cells can be, and often are, larger and slower than full-custom designs [5]. One of the goals of the Totem project is to automate the generation of FPGAs that begin to approach the level of performance that full-custom layouts currently enjoy.

The Totem project has decided to investigate three different approaches to automate the layout process: standard cells, template reduction, and FPGA-specific circuit generators. A goal of the Totem project is to decide which of the three approaches should be used in a particular situation. We may find that one approach is the best for all situations, or that each approach has compelling characteristics that make it the best choice in a particular instance. The goal of this thesis is to investigate the standard cell approach. Towards this end, I present the creation of a standard cell tool flow and the investigation of how removal of logic from the standard and FPGA specific cell templates compares to a full custom RaPiD array.

The approach that I am currently investigating is the use of standard cells. I am further refining the standard cell approach by adding cells to the library that are specific to or used extensively in FPGA designs, thus creating an optimized library. These include muxes, demuxes, and SRAM bits among others. I predict that by adding a few critical cells, the results obtained can be enhanced. The standard cell approach is flexible enough that various cell libraries can be swapped in and out, as circumstances dictate.

The next implementation that will be evaluated is the use of a feature rich template that will be a superset of the required logic. Once the actual application domain that the circuit is expected to operate in is known, logic that is not needed will be eliminated from the template. This approach has the potential to deliver circuits that are very efficient, as long as the application domain is supported by the available templates.

The final method that will be considered is the creation of FPGA-specific circuit generators, which will be akin to memory generators. A custom circuit generator program will be created that will generate logic within the constraints of the overall architecture. These three techniques will be explained in detail in the next sections.

4.1. Standard Cell Implementation

The use of template reduction produces very efficient implementations, but it only works well if the proposed architecture does not deviate significantly from the provided macro cells. To fill the gaps that exist between templates' domains, I 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.

Using standard cells also creates an opportunity to more aggressively optimize logic than if templates were used, since the circuit can be built from the ground up. It will also allow the designer to easily integrate this method into the normal SOC design flow. In addition, the structures created will retain their reconfigurability since the CLBs and routing interconnect will be programmable. This is achieved by creating a structural Verilog representation of the FPGA, and then generating a standard cell layout based upon that Verilog. Since the Verilog design includes SRAM bits and programmability, the result is a reconfigurable ASIC.

Unfortunately, this method also inherits all of the drawbacks introduced into a design by the use of standard cells, including increased circuit size and reduced performance. To overcome these failings, we will create standard cells that are specific to or often used in FPGAs. These cells will include LUTs, SRAM bits, muxes, demuxes, and other typical FPGA components. Since these cells are used extensively in FPGAs, significant improvement could be attained. In this thesis we compare a standard cell library and a more comprehensive optimized library.

4.2. Template Reduction

The template reduction method will leverage the performance edge that full-custom layouts provide, in an automated fashion. This is achieved by using feature rich macro cells as templates that are reduced and compacted to form the final circuit. Providing quality macro cells that can cover a wide range of applications is critical to the success of this method. Therefore, extensive profiling of application domains will be required to establish what resource mixes are needed for each template.

Figure 4-1 is an example of how a template is reduced. The initial template has numerous muxes, along with a 2-input LUT and a DFF in each cell. The high-level architecture generator finds two mappings that must be supported by the circuit to cover the application domain. All resources that are not needed to support these two mappings are pruned from the initial template. Therefore, any unused or under-utilized functional units are removed, like the DFFs, or modified, like the LUT, into an OR gate in cell two. Also, muxes become wires or have their number of inputs reduced, and wires that are not needed are removed. The final optimized template is then compacted to further increase performance.

The potential exists to create designs that achieve a performance level that is at parity with that of ASICs. But this is only possible if two conditions are met: the applications that the template needs to support are similar in composition, and the domain that contains the applications is a subset of the initial template. The first condition implies that if a template is required to support a wide range of applications, then it will have to retain most of its reconfigurability. This means that the performance of the final template will be near that of an FPGA, which, as noted earlier, is usually not optimal. While the first condition may mean the final template may not perform well, the second condition, if not met, may mean that the applications specified cannot be mapped onto the initial template. This could occur, for example, if the initial template does not contain a multiplier, but the applications that need to run on the template require one.

With these conditions in mind, if the design specified by the architecture generator does not deviate significantly from the available macro cells, we can use the template reduction method to automate the layout generation in a fast and efficient manner without sacrificing performance. Since our initial focus is on a one-dimensional array as our target FPGA, we will be using variations of the RaPiD architecture as a basis for our feature rich macro template.

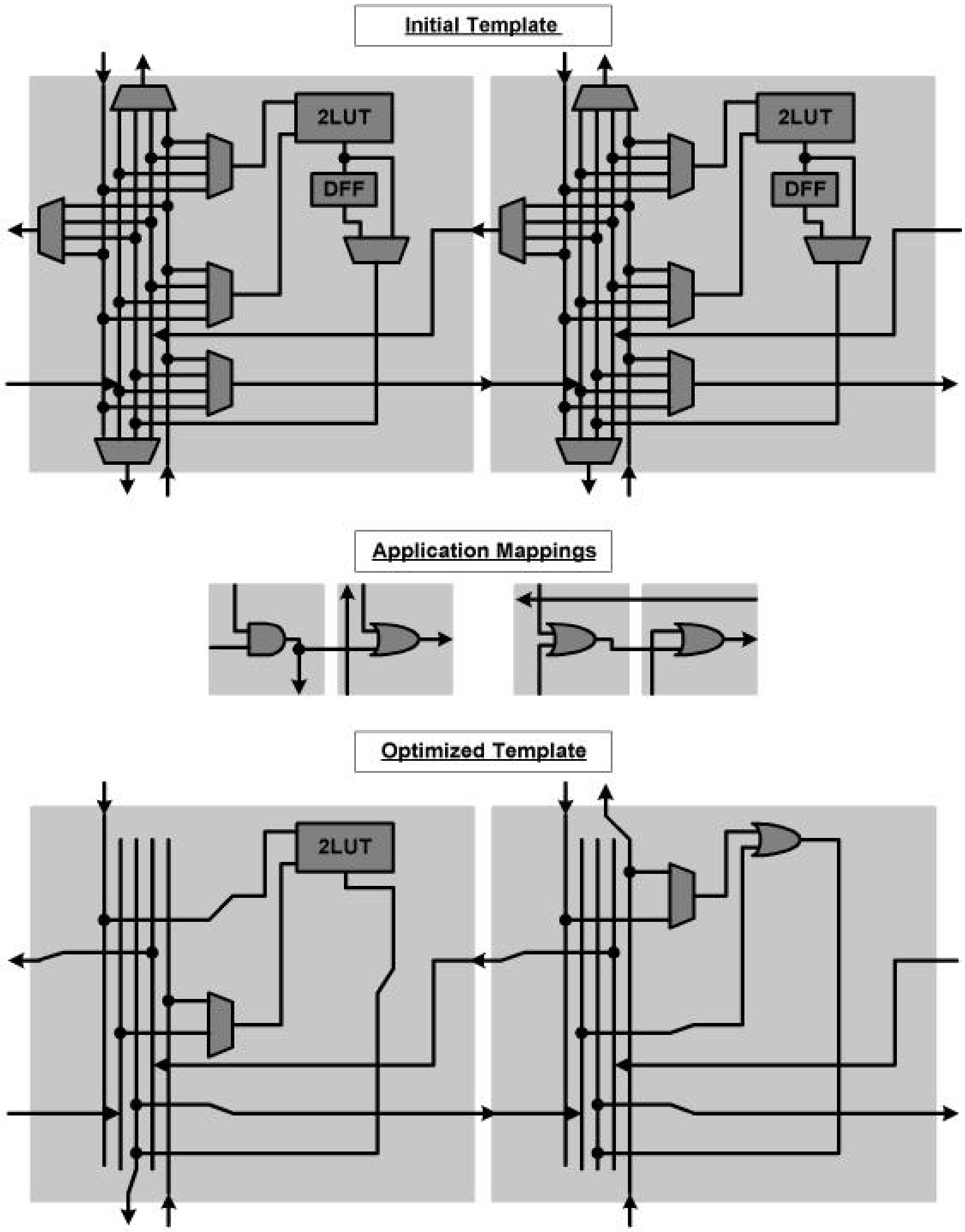


Figure 4-1: Simplified example of template reduction. The initial template (top) is a modified version of the Xilinx XC6200. The high-level architecture generator has found two target applications (middle). Logic resources that are not needed, including routing resources, are removed from the initial template to create the optimized template (bottom). Notice how both DFFs have been removed, and how the 2LUT in the right cell is reduced to an OR gate.

4.3. FPGA-specific Circuit Generators

The standard cell design method is very flexible, and it gives the designer the ability to implement almost any circuit. But 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 may produce designs that are of higher quality than standard cell based designs.

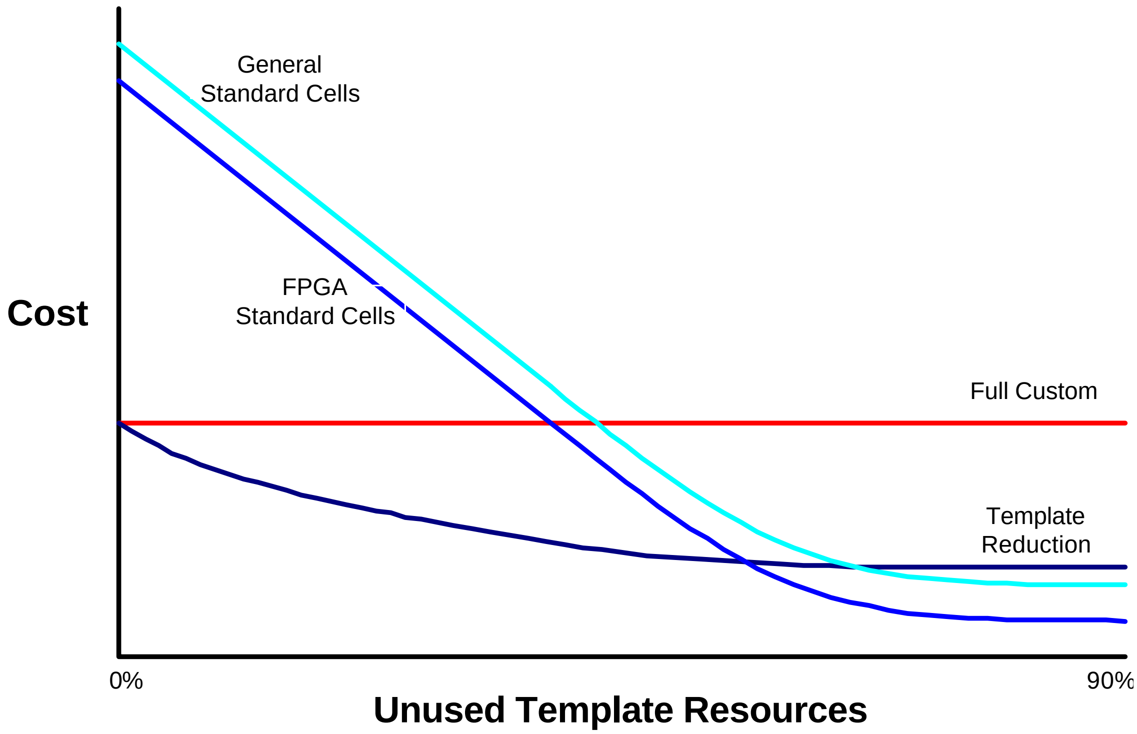
One way of creating very regular circuits is through the use of generators. Circuit generators are used to great effect in the memory industry, and it is our belief that we will be able to achieve similar results. FPGA components, like memories, have well-known, constrained structures, positioning them as viable candidates for circuit generators.

Circuit generators will be able to create structures that are of higher quality than those created by the standard cell method. However, unlike the template reduction method, the circuit generators will be able to handle a wider variety of possible architectures. Thus, circuit generators will be positioned to fill the gap between the inflexible, but powerful, template reduction method and the very flexible, but less efficient, standard cell method.

Circuit generators will be implemented to create the parts of the FPGA that inherently have regularity. This includes generators for the routing channels, LUTs, and muxes and demuxes for routing interconnect. To create an entire reconfigurable subsystem out of blocks of logic that circuit generators have created, one would only need to abut the blocks together. Therefore, all of these generators will be combined to create a method that is capable of generating a complete reconfigurable subsystem.

4.4. Expectations

Graph 4-1 is an estimate of where we expect each approach to fall in the search space. The y-axis represents the cost of the architecture as measured by its area, performance, and power consumption, with a lower value meaning a higher quality design. The x-axis represents how much a template utilizes the resources present in a superset reference design. A lower value implies that a template is more like the reference design, and increasing values represent templates that have more and more gates removed.



Graph 4-1 This graph shows our initial guess of the cost of a given design vs. the utilization of template resources, defined by how many resources a given template contains, when compared to a reference macro cell. A lower cost design is preferable.

5. Experimental Setup and Procedure

5.1. Setup

Synopsys

To retain as much flexibility as possible in our standard cell implementation, behavioral Verilog representations were created for all of the RaPiD components. Therefore, Synopsys was used to synthesize the behavioral Verilog to produce structural Verilog that uses our standard cells [16]. This will enable us 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.

Silicon Ensemble

Silicon Ensemble (SE) is a place-and-route tool that is part of the Cadence Envisia Tool Suite [17]. SE is capable of routing multiple layers of metal, including over-the-cell routing. SE receives as input a file in the library exchange format (LEF) of the standard cells and a Verilog netlist representation of the circuit that will be placed and routed. The final output of the tool is a design exchange format (DEF) file representation of the placed and routed array that is ready to be read back into Cadence. One powerful feature of SE is its ability to run from macro files, minimizing the amount of user intervention. SE will be used to place-and-route all circuits that are created using the standard cells. Note that SE should not be confused with the Totem place-and-route tool, since Totem's place-and-route tool maps netlists onto physical designs that are created by SE.

Cadence

Cadence was chosen as our schematic and layout editor because it is a very robust tool set that is widely used in industry [18]. Cadence also has tools for every aspect of the design flow. We are currently using the TSMC 0.25micron design rules for all layouts created in Cadence. As technology scales, we will be able to scale our layouts down without a loss of quality in our results, since all designs have been laid-out in the same technology. Therefore, any performance increases that could be achieved by scaling down to a more aggressive technology will be felt across all designs.

RaPiD

The RaPiD components that were used in benchmarking were laid out by Carl Ebeling's group for the RaPiD I powertest. All circuits were laid out using the Magic Layout Editor for the HP 0.35micron process. The designs were ported over to Cadence and the TSMC 0.25micron process.

Tanner Standard Cells

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 0.25micron process. This led us to the Tanner standard cell library that is available through the MOSIS prototyping production service [19]. This library has thirty-two basic blocks at its core, which can then be applied to produce any combination of the 1400+ functional blocks that Tanner provides in its Tanner SchemLib symbol library.

Epic Tools

We are using the Epic Tool Suite to analyze the performance and power consumption of all of the circuits that have been created. Synopsys has developed the Epic Tool Suite as a robust circuit simulator that enables designers to verify circuit performance at both pre-layout and post-layout without fabrication of the design [20]. The Epic tools use a version of the SPICE engine for circuit simulation, and they also use the SPICE netlist format as circuit input and the SPICE BSIM3V3 as a transistor model format. The two tools that we will be mainly using from the tool suite are Pathmill and Powermill.

Pathmill

Pathmill is a transistor-level critical path and timing verifier for full chip circuit designs [21]. Pathmill can handle multiple circuit design styles, including static CMOS, domino logic, and pass-gate logic. Through the use of the BSIM3V3 transistor models, Pathmill is able to provide transistor-level delay calculations that are equivalent to SPICE in accuracy. We will use Pathmill to generate the timing numbers for the critical paths of all circuit designs.

Powermill

Powermill is a transistor-level power simulator and analyzer for full chip circuit designs [22]. Powermill is capable of running transistor-level simulations at high accuracy 10 to 1000 times faster than SPICE, through the use of the BSIM3V3 transistor models. By using Powermill, we will be able to estimate the power consumption of all circuit designs.

5.2. Procedure

The experimental procedure was driven by our use of RaPiD as a starting point, and the use of the tools that were mentioned in the experimental setup section. Figure 5-1 shows the tool flow. While there is still considerable manual intervention involved in each step of the flow, our eventual goal is a truly automated process.

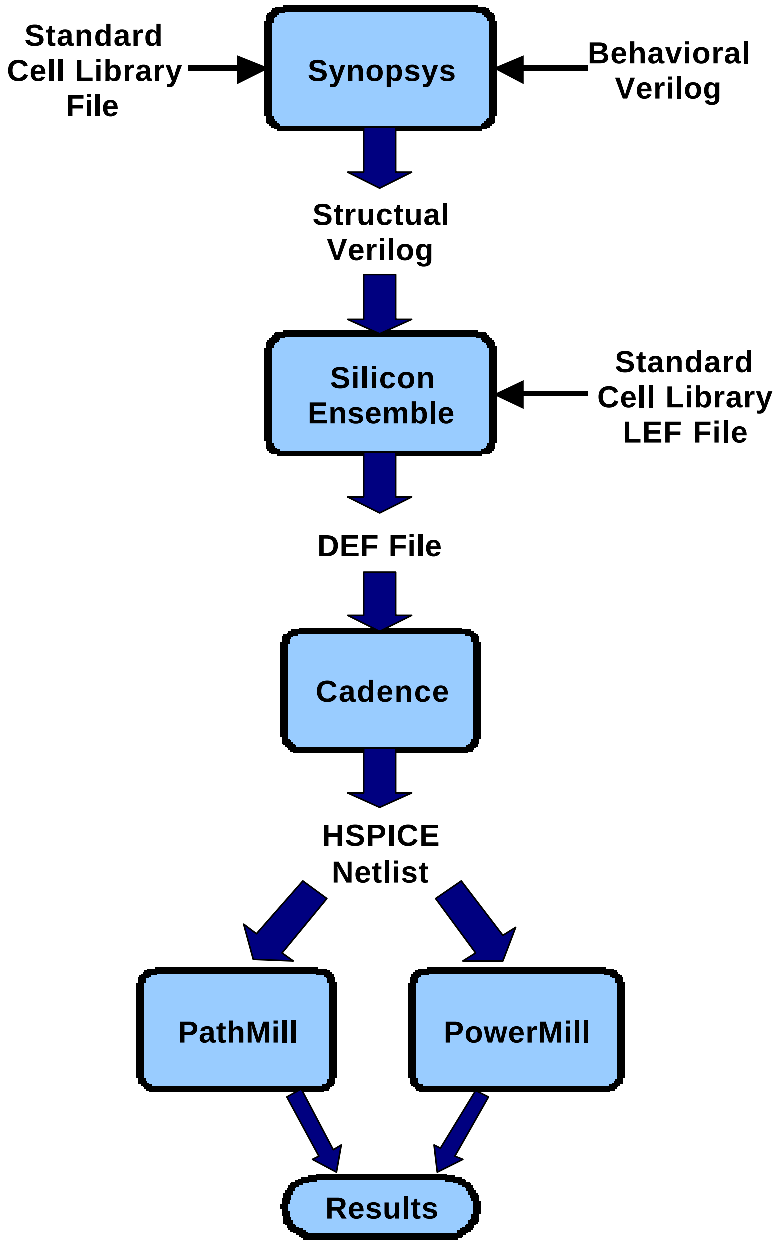


Figure 5-1: Tool-Flow for automating the creation of standard cell templates.

We first imported the RaPiD I powertest components from the Magic Layout Editor using a HP 0.35micron process to Cadence using a TSMC 0.25micron process. To do this, the files were first exported out of the Magic Layout Editor in a CIF file. We then proceeded to modify these CIF files to force compatibility with the TSMC 0.25micron process. Once this was done, the files were then imported into Cadence, and all remaining design errors were corrected by hand. Schematic and Verilog representation of the RaPiD components were also created.

The next step was to find an appropriate standard cell library. As stated above, we settled on the Tanner Standard Cell library. Even though the library was targeted at the TSMC 0.25micron process, the layouts were generated using the more aggressive deep sub-micron version of the process with a lambda of 0.12microns. However, we are currently using the deep-micron version of the TSMC 0.25micron process with a lambda of 0.15microns. This caused some minor problems that were cleaned up by hand using pre-import scripts and some post-import modifications. A library information file representation of the Tanner Cells was also created for Synopsys.

To generate the standard cell version of RaPiD, the tool-flow shown in figure 5 was used. A behavioral Verilog representation of RaPiD was first created. Synopsys was then used to synthesize this Verilog file to create a structural Verilog file that used the Tanner standard-cells as modules. With this structural Verilog, SE was able to then place-and-route the entire design. The utilization level of SE, 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 SE was done creating the layout, the EPIC tool-set was used to evaluate the quality of the circuit that was created.

6. Results

Graph 4-1 above shows what we initially expected. Namely, that the full custom design would cost appreciably less than either version of the standard cells. But as logic is reduced from the standard cell templates, the difference in cost becomes minimal. We also expect that the FPGA standard cells would outperform the general standard cells. The following data show our actual results.

6.1. Test Environment

The tool-flow and procedure were discussed in detail in section 5.1 and 5.2 above. The tools were run on nine Sun Ultra Five workstations, four machines with 512MB of memory and 5 machines with 384MB of memory. The runtime of the entire tool-flow to generate each template was approximately six hours.

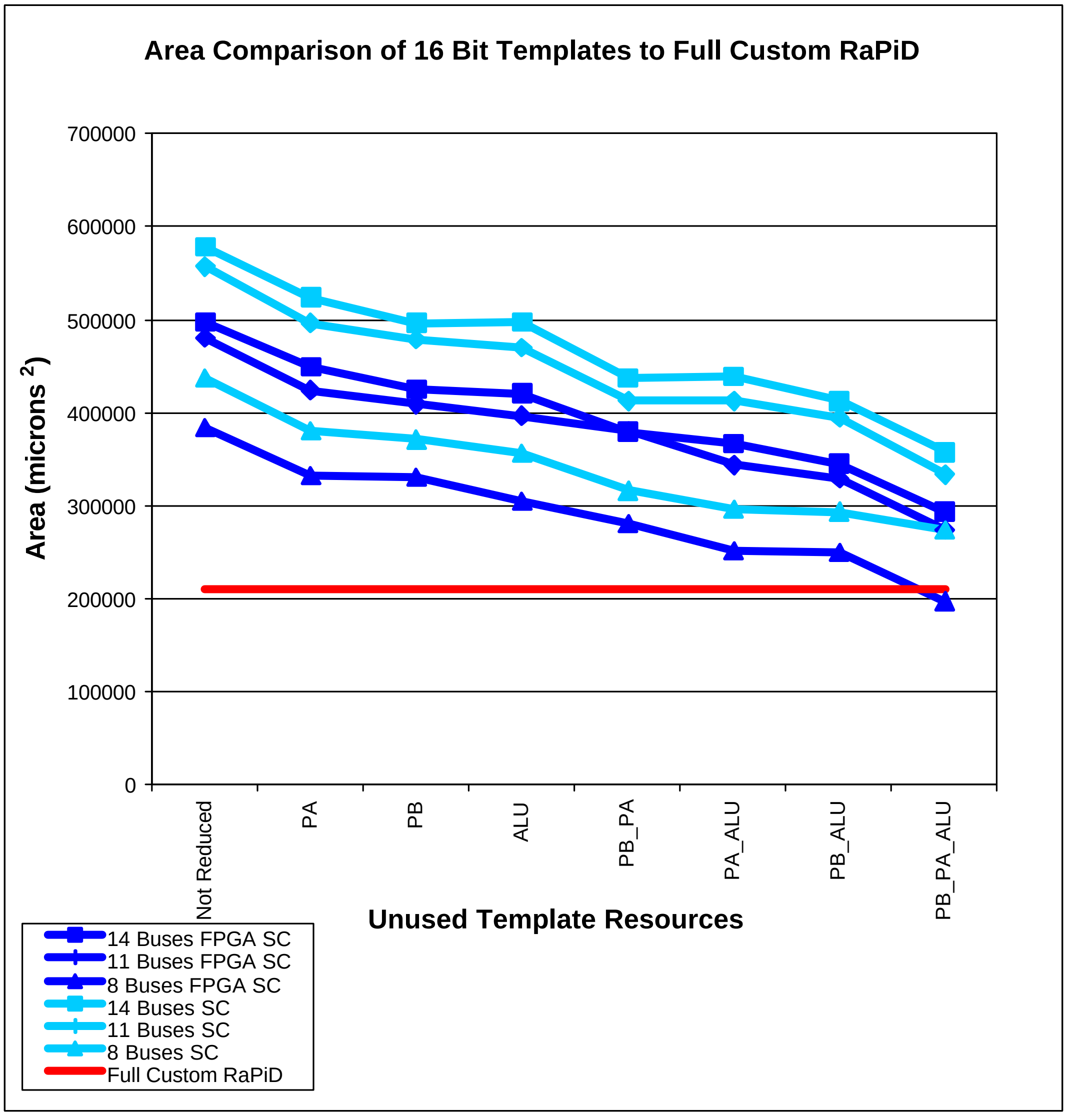
As a template is customized towards an application domain, gains in performance and reductions in power and area are possible. To test this, both the Tanner and FPGA standard cell templates were reduced to reflect possible design scenarios. The scenarios included reducing the ALU to and adder, reducing the word size, reducing the number of pipeline registers, reducing the ALU size, and all of the associated cross products. Table 6-1 below describes the eight different designs that were created. These designs were further implemented with 8 buses, 11 buses, and 14 buses for both the Tanner and FPGA standard cells. Also, all designs were created with both an eight and a sixteen bit word size.

Template	Description
Not_reduced	Full template
PA	Removed pipeline registers after the second ALU
PB	Removed pipeline registers before the first ALU
ALU	Converted the ALUs to adders
PB_PA	Removed pipeline registers before the first ALU and after the second ALU
PA_ALU	Removed pipeline registers after the second ALU and reduced the ALUs to adders
PB_ALU	Removed pipeline registers before the first ALU and reduced the ALUs to adders
PB_PA_ALU	Reduced the ALUs to adders and removed pipeline registers before the first ALU and after the second ALU

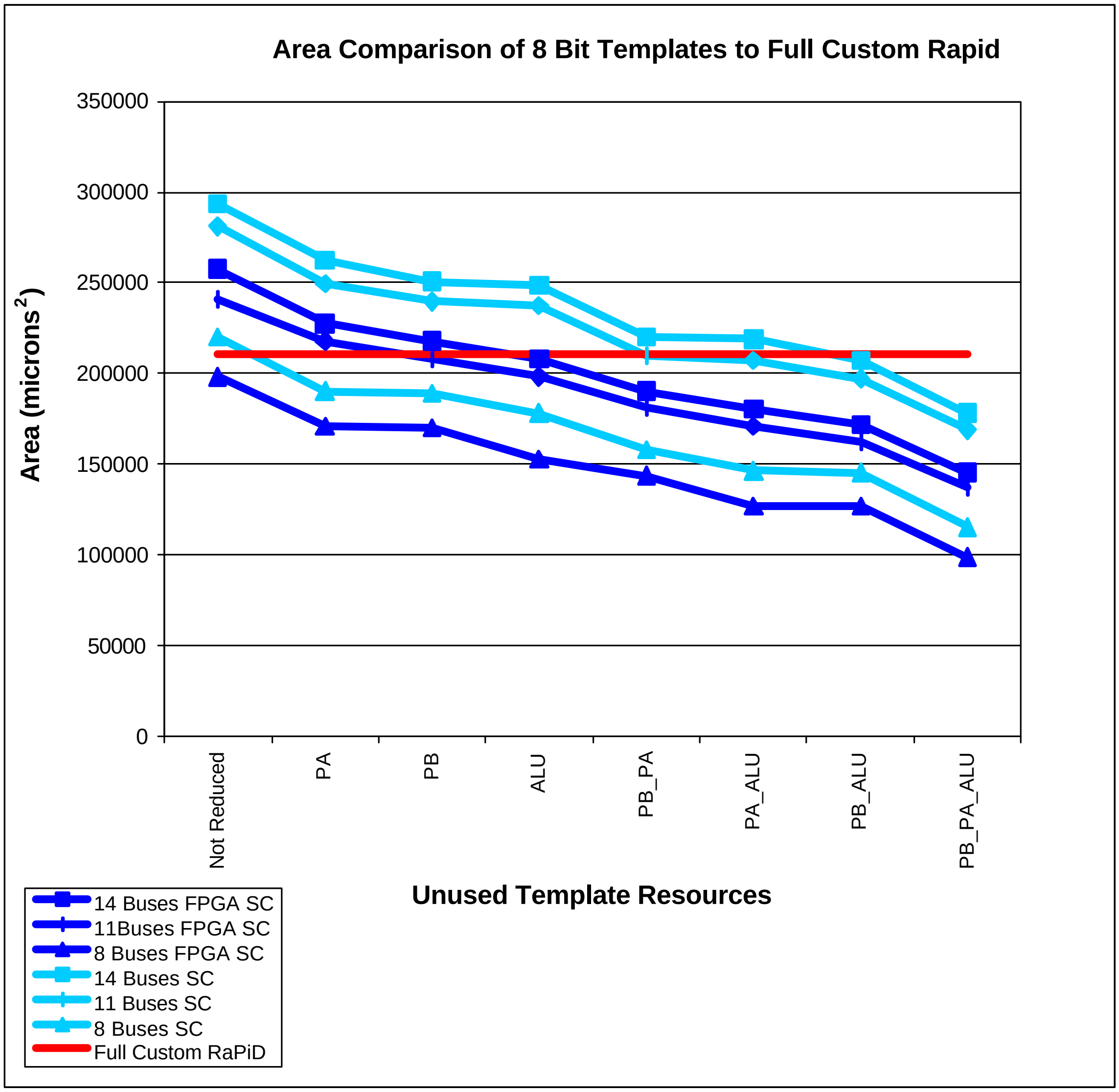
Table 6-1 Template nomenclature and description.

6.2. Area

Graph 6-1 and Graph 6-2 and Table 6-2 through Table 6-8 show the area in microns^2 of full custom RaPiD, the general standard-cell templates, and the FPGA standard cell templates. The area of full custom RaPiD is smaller than all standard cell templates in all of the 16 bit cases. The 8 bit templates have lower area then full custom RaPiD about half the time. Also, the FPGA standard cell templates are smaller than the Tanner standard cell templates overall.



Graph 6-1 This graph shows the area of full custom RaPiD as well as all of the different versions of the 16 bit templates vs. unused template resources. The x-axis is a list of each version of template, with the versions that do not have any logic units removed from them on the left, ranging to versions that have an increasing amount of logic removed from them.



Graph 6-2 This graph shows the area of full custom RaPiD as well as all of the different versions of the 8 bit templates vs. unused template resources. The x-axis is a list of each version of template, with the versions that do not have any logic units removed from them on the left, ranging to versions that have an increasing amount of logic removed from them.

Full Custom RaPiD Area Comparison			
	Area (microns^2)	**% Area Improvement over 14 Buses FPGA SC**	**% Area Improvement over 14 Buses Tanner SC**
Full Custom RaPiD	210,577.050	**57.7**	**63.6**

Table 6-2 Area of full custom RaPiD vs. the 14 buses not reduced FPGA standard cell template and the 14 buses not reduced Tanner standard cell template.

16 Bit 8 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	383,395.680	437,179.680	**12.3**
PA	90	331,763.040	380,440.800	**12.8**
PB	90	330,389.280	370,863.360	**10.9**
ALU	89/90	304,702.560	355,687.200	**14.3**
PB_PA	90	280,052.640	316,068.480	**11.4**
PA_ALU	90	250,983.360	296,136.000	**15.2**
PB_ALU	90	249,791.040	293,544.000	**14.9**
PB_PA_ALU	90	197,134.560	237,362.400	**16.9**

Table 6-3 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with 16 bits and 8 buses. Note that the ALU FPGA SC template was generated with 89% utilization while the Tanner standard cell was generated at 90% utilization. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

16 Bit 11 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	480,245.760	557,111.520	**13.8**
PA	90	423,792.000	496,173.600	**14.6**
PB	90	409,082.400	478,586.880	**14.5**
ALU	90	396,135.360	469,476.000	**15.6**
PB_PA	85/90	380,440.800	412,140.960	**7.7**
PA_ALU	90	343,621.440	412,140.960	**16.6**
PB_ALU	90	329,015.520	394,632.000	**16.6**
PB_PA_ALU	90	273,093.120	333,136.800	**18.0**

Table 6-4 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with 16 bits and 11 buses. Note that the PB_PA FPGA SC template was generated with 85% utilization while the Tanner standard cell was generated at 90% utilization. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

16 Bit 14 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	497,858.400	577,886.400	**13.8**
PA	90	449,167.680	523,596.960	**14.2**
PB	90	425,347.200	496,173.600	**14.3**
ALU	90	420,681.600	497,858.400	**15.5**
PB_PA	85/90	378,963.360	437,179.680	**13.3**
PA_ALU	90	366,508.800	438,760.800	**16.5**
PB_ALU	90	345,021.120	412,140.960	**16.3**
PB_PA_ALU	90	293,544.000	357,112.800	**17.8**

Table 6-5 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with 16 bits and 14 buses. Note that the PB_PA FPGA SC template was generated with 85% utilization while the Tanner standard cell was generated at 90% utilization. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

8 Bit 8 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	198,197.280	220,125.600	**10.0**
PA	90	170,890.560	190,252.800	**10.2**
PB	90	169,905.600	189,216.000	**10.2**
ALU	90	152,565.120	178,378.200	**14.5**
PB_PA	90	143,791.200	158,170.320	**10.0**
PA_ALU	90	127,020.960	146,512.800	**13.3**
PB_ALU	90	127,020.960	145,605.600	**12.8**
PB_PA_ALU	90	98,845.920	115,305.120	**14.3**

Table 6-6 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with 16 bits and 8 buses. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

8 Bit 11 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	240,861.600	281,322.720	**14.4**
PA	90	217,896.500	249,791.000	**12.8**
PB	90	208,474.560	239,695.200	**13.0**
ALU	90	198,197.300	237,362.400	**16.5**
PB_PA	90	181,452.960	209,563.200	**13.4**
PA_ALU	90	170,890.560	207,385.920	**17.6**
PB_ALU	90	162,557.280	197,134.560	**17.5**
PB_PA_ALU	90	137,039.040	168,920.640	**18.9**

Table 6-7 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with 16 bits and 11 buses. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

8 Bit 14 Buses Area Comparison				
Templates	% Utilization	Area of FPGA SC (microns^2)	Area of Tanner SC (microns^2)	**% Area Improvement of FPGA vs. Tanner**
Not_reduced	90	257,657.760	293,544.000	**12.2**
PA	90	227,525.760	262,530.720	**13.3**
PB	90	217,896.480	250,983.360	**13.2**
ALU	90	208,474.560	248,598.720	**16.1**
PB_PA	90	190,252.800	220,125.600	**13.6**
PA_ALU	90	180,442.080	219,011.040	**17.6**
PB_ALU	90	171,875.520	207,385.920	**17.1**
PB_PA_ALU	90	145,605.600	178,420.320	**18.4**

Table 6-8 Area comparison of Tanner standard cell templates vs. FPGA standard cell templates with a word size of 14 bits. Utilization refers to the effort level of Silicon Ensemble to densely place cells.

.

One conclusion that can be drawn from these results is that full custom RaPiD is smaller then almost all of the reduced templates. But, as the application domain narrows, the standard cell methods create compelling designs. Also, with a minimal investment in resources, the creation of a few FPGA specific cells greatly enhances the quality of the designs generated, with reductions in area as high as 18.9%. Therefore, an optimized standard cell library should be considered over a general standard cell library in all cases.

6.3. Performance

As described in section 5.1, performance numbers were generated using PathMill. The following tables present the performance numbers for full custom RaPiD and all of the templates. To measure performance, we have listed the critical paths of each design defined in nanoseconds. The number of stages is the number of equivalent inverter delays the critical path equals. Also listed is the ratio of full custom RaPiD's longest delay to each template's longest delay. A value greater than one represents a template that has a shorter critical path than full custom RaPiD.

16 Bit 8 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)	Number of Stages	
FC_RaPiD	8.875	128	
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	12.782	132	0.69
PA	12.197	126	0.73
PB	11.817	123	0.75
ALU (89%)	8.960	98	0.99
PB_PA	11.220	117	0.79
PA_ALU	8.347	92	1.06
PB_ALU	7.928	88	1.12
PB_PA_ALU	7.335	82	1.21
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	12.837	157	0.69
PA	12.117	143	0.73
PB	12.701	149	0.70
ALU	9.572	126	0.93
PB_PA	10.964	129	0.81
PA_ALU	8.895	114	1.00
PB_ALU	8.479	110	1.05
PB_PA_ALU	7.793	98	1.14

Table 6-9 Performance numbers for both versions of templates with 16 bits and 8 buses. Note that PathMill would not run PB.

16 Bit 11 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)		Number of Stages
FC_RaPiD	8.875		128
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	13.836	140	0.64
PA	13.247	134	0.67
PB	12.681	131	0.70
ALU	9.990	106	0.89
PB_PA (85%)	12.076	123	0.73
PA_ALU	9.407	100	0.94
PB_ALU	8.821	94	1.01
PB_PA_ALU	8.236	88	1.08
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	13.963	181	0.64
PA	13.254	169	0.67
PB	12.644	161	0.70
ALU	10.774	150	0.82
PB_PA	11.920	149	0.74
PA_ALU	10.100	138	0.88
PB_ALU	9.474	130	0.94
PB_PA_ALU	8.775	118	1.01

Table 6-10: Performance numbers for both versions of templates with 16 bits and 11 buses.

16 Bit 14 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)		Number of Stages
FC_RaPiD	8.875		128
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	13.730	144	0.65
PA	13.125	138	0.68
PB	12.599	133	0.70
ALU	9.889	110	0.90
PB_PA (85%)	12.029	127	0.74
PA_ALU	9.274	104	0.96
PB_ALU	8.701	98	1.02
PB_PA_ALU	8.134	92	1.09
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	14.022	181	0.63
PA	13.297	169	0.67
PB	12.701	161	0.70
ALU	10.921	150	0.81
PB_PA	12.012	149	0.74
PA_ALU	10.144	138	0.87
PB_ALU	9.504	130	0.93
PB_PA_ALU	9.159	118	0.97

Table 6-11: Performance numbers for both versions of templates with 16 bits and 14 buses.

8 Bit 8 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)	Number of Stages	
FC_RaPiD	8.875	128	
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	10.973	116	0.81
PA	10.337	110	0.86
PB	9.980	107	0.89
ALU	7.621	82	1.16
PB_PA	9.381	101	0.95
PA_ALU	7.033	76	1.26
PB_ALU	6.596	72	1.35
PB_PA_ALU	6.005	66	1.48
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	12.121	135	0.73
PA	11.077	127	0.80
PB	10.617	124	0.84
ALU	8.305	110	1.07
PB_PA	10.049	101	0.88
PA_ALU	7.591	98	1.17
PB_ALU	7.204	94	1.23
PB_PA_ALU	6.513	82	1.36

Table 6-12 Performance numbers for both versions of templates with 8 bits and 8 buses. Note that PathMill would not run Tanner SC PB_PA.

8 Bit 11 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)		Number of Stages
FC_RaPiD	8.875		128
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	11.925	128	0.74
PA	10.337	110	0.86
PB	10.743	115	0.82
ALU	8.562	94	1.04
PB_PA	10.158	111	0.87
PA_ALU	7.966	88	1.11
PB_ALU	7.398	82	1.20
PB_PA_ALU	6.799	76	1.31
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	13.056	164	0.68
PA	12.236	151	0.73
PB	11.600	144	0.77
ALU	9.484	134	0.94
PB_PA	10.881	131	0.82
PA_ALU	8.784	122	1.01
PB_ALU	8.179	114	1.09
PB_PA_ALU	7.23	100	1.23

Table 6-13 Performance numbers for both versions of templates with 8 bits and 11 buses. Note that PathMill would not run FPGA SC PA, ALU, PB_PA, and Tanner SC PA.

8 Bit 14 Buses Performance Comparison to Full Custom RaPiD			
	Longest Delay (ns)	Number of Stages	
FC_RaPiD	8.875	128	
FPGA Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/FPGA SC Longest Delay
Not_reduced	12.017	124	0.74
PA	10.829	120	0.82
PB	10.834	115	0.82
ALU	8.373	100	1.06
PB_PA	10.251	95	0.87
PA_ALU	8.093	84	1.10
PB_ALU	7.501	78	1.18
PB_PA_ALU	6.918	72	1.28
Tanner Standard Cells Template	Longest Delay (ns)	Number of Stages	Ratio of FC RaPiD Longest Delay/Tanner SC Longest Delay
Not_reduced	13.135	164	0.68
PA	12.27	150	0.72
PB	11.698	144	0.76
ALU	9.521	134	0.93
PB_PA	10.981	131	0.81
PA_ALU	8.841	122	1.00
PB_ALU	8.217	114	1.08
PB_PA_ALU	7.540	102	1.18

Table 6-14 Performance numbers for both versions of templates with 8 bits and 14 buses. Note that PathMill would not run FPGA SC PB and Tanner SC PB_PA_ALU.

The performance results from Table 6-9 through Table 6-11 reflect what one would expect to see based upon the difference in area of each template, with a few exceptions. While there are only two 16-bit templates that have a smaller area than full custom RaPiD, there are eight FPGA standard cell templates and three Tanner standard cell templates that have a shorter critical path. Also, the 8 bit templates routinely perform better then full custom RaPiD. This is a very promising result, and can be attributed to SE's aggressive place and route algorithms.

6.4. Power Utilization

As described in section 5.1, power utilization numbers were generated using PowerMill. The following tables present the power utilization numbers for full custom RaPiD and all of the templates. To measure power utilization, we have listed the average vdd and gnd current of each design defined in micro-amps. Also listed is the ratio of full custom RaPiD's power utilization to each template's power utilization for both vdd and gnd. A value greater than one represents a template that has a lower average current.

16 Bit 8 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		VDD Average Current (mA)	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	6.64e+02	95.93	-6.77e+02	94.09
PA	6.68e+02	95.36	-6.80e+02	93.68
PB	8.41e+02	75.74	-8.54e+02	74.59
ALU	6.59e+02	96.66	-6.71e+02	94.93
PB_PA (85%)	8.10e+02	78.64	-8.23e+02	77.40
PA_ALU	6.73e+02	94.65	-6.85e+02	92.99
PB_ALU	8.32e+02	76.56	-8.44e+02	75.47
PB_PA_ALU	8.14e+02	78.62	-8.25e+02	77.21
Tanner Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	6.96e+03	9.15	-6.96e+03	9.15
PA	7.32e+03	8.70	-7.32e+03	8.70
PB	ND	ND	ND	ND
ALU	4.49e+03	14.19	-4.49e+03	14.19
PB_PA	5.77e+03	11.04	-5.77e+03	11.04
PA_ALU	4.17e+03	15.28	-4.17e+03	15.28
PB_ALU	3.74e+03	17.03	-3.74e+03	17.03
PB_PA_ALU	3.21e+03	19.84	-3.21e+03	19.84

Table 6-15: Power numbers for both versions of templates with 16 bits and 8 buses. Note that PowerMill would not run PB.

16 Bit 11 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		VDD Average Current (mA)	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	1.61e+03	39.57	-1.63e+03	39.08
PA	6.62e+03	9.62	-6.63e+03	9.61
PB	7.85e+03	8.11	-7.87e+03	8.09
ALU	2.28e+03	27.94	-2.30e+03	27.70
PB_PA (85%)	4.42e+03	14.41	-4.44e+03	14.35
PA_ALU	2.49e+03	25.58	-2.51e+03	25.38
PB_ALU	1.62e+03	39.32	-1.64e+03	38.84
PB_PA_ALU	1.78e+03	35.79	-1.80e+03	35.39
Tanner Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	1.14e+03	55.88	-1.14e+03	55.88
PA	1.12e+03	56.88	-1.12e+03	56.88
PB	6.15e+03	10.36	-6.15e+03	10.36
ALU	4.60e+03	13.85	-4.60e+03	13.85
PB_PA	5.05e+03	12.61	-5.05e+03	12.61
PA_ALU	3.37e+03	18.90	-3.37e+03	18.90
PB_ALU	3.99e+03	15.96	-3.99e+03	15.96
PB_PA_ALU	1.56e+03	40.83	-1.56e+03	40.83

Table 6-16: Power numbers for both versions of templates with 16 bits and 11 buses.

16 Bit 14 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		**VDD Average Current (mA)**	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	8.90e+02	71.57	-9.08e+02	70.15
PA	8.50e+02	74.94	-8.68e+02	73.39
PB	3.65e+03	17.45	-3.66e+03	17.40
ALU	8.63e+02	73.81	-8.81e+02	72.30
PB_PA (85%)	6.05e+03	10.53	-6.06e+03	10.51
PA_ALU	8.77e+02	72.63	-8.96e+02	71.09
PB_ALU	8.80e+02	72.39	-8.99e+02	70.86
PB_PA_ALU	8.93e+02	71.33	-9.12e+02	69.85
Tanner Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	1.18e+03	53.98	-1.18e+03	53.98
PA	1.17e+03	54.44	-1.17e+03	54.44
PB	1.21e+03	52.64	-1.21e+03	52.64
ALU	1.14e+03	55.88	-1.14e+03	55.88
PB_PA	6.80e+03	9.37	-6.80e+03	9.37
PA_ALU	1.18e+03	53.98	-1.11e+03	53.98
PB_ALU	3.41e+03	18.68	-3.41e+03	18.68
PB_PA_ALU	2.58e+03	24.69	-2.58e+03	24.69

Table 6-17: Power numbers for both versions of templates with 16 bits and 14 buses.

8 Bit 8 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		**VDD Average Current (mA)**	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	3.45e+02	184.64	-3.51e+02	181.48
PA	3.31e+02	192.45	-3.38e+02	188.46
PB	4.06e+02	156.90	-4.12e+02	154.61
ALU	3.43e+02	185.71	-3.49e+02	182.52
PB_PA	4.09e+02	155.75	-4.15e+02	153.49
PA_ALU	3.33e+02	191.29	-3.40e+02	187.35
PB_ALU	3.98e+02	160.05	-4.04e+02	157.67
PB_PA_ALU	4.16e+02	153.13	-4.22e+02	150.95
Tanner Standard Cell Template	GND Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (µA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	9.78e+03	6.51	-9.78e+03	6.51
PA	3.16e+03	20.16	-3.16e+03	20.16
PB	3.27e+03	19.48	-3.27e+03	19.48
ALU	2.29e+03	27.82	-2.29e+03	27.82
PB_PA	ND	ND	ND	ND
PA_ALU	1.84e+03	34.62	-1.84e+03	34.62
PB_ALU	1.76e+03	36.19	-1.76e+03	36.19
PB_PA_ALU	1.64e+03	38.84	-1.64e+03	38.84

Table 6-18: Power numbers for both versions of templates with 8 bits and 8 buses. Note that PowerMill would not run Tanner SC PB_PA.

8 Bit 11 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		**VDD Average Current (mA)**	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	2.39e+03	26.54	-2.40e+03	26.54
PA	ND	ND	ND	ND
PB	1.81e+03	35.19	-1.82e+03	35.00
ALU	ND	ND	ND	ND
PB_PA	ND	ND	ND	ND
PA_ALU	4.24e+02	150.24	-4.33e+02	147.11
PB_ALU	1.33e+03	47.89	-1.33e+03	47.89
PB_PA_ALU	4.09e+02	155.75	-4.19e+02	152.03
Tanner Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	5.75e+02	110.78	-5.75e+02	110.78
PA	ND	ND	ND	ND
PB	3.71e+03	17.17	-3.71e+03	17.17
ALU	2.12e+03	30.05	-2.12e+03	30.05
PB_PA	5.98e+02	106.52	-5.98e+02	106.52
PA_ALU	1.60e+03	39.81	-1.60e+03	39.81
PB_ALU	6.01e+02	105.99	-6.01e+02	105.99
PB_PA_ALU	1.29e+03	49.38	-1.29e+03	49.38

Table 6-19: Power numbers for both versions of templates with 8 bits and 11 buses. Note that PowerMill would not run FPGA SC PA, PB, and PB_PA, and Tanner SC PA.

8 Bit 14 Buses Power Comparison to Full Custom RaPiD				
	GND Average Current (mA)		VDD Average Current (mA)	
FC_RaPiD	6.37e+04		-6.37e+04	
FPGA Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	2.39e+03	26.65	-2.40e+03	26.54
PA	3.32e+02	191.87	-3.38e+02	191.87
PB	ND	ND	ND	ND
ALU	4.42e+02	144.12	-4.51e+02	144.12
PB_PA	4.43e+02	143.79	-4.52e+02	143.79
PA_ALU	8.08e+02	78.84	-8.17e+02	77.96
PB_ALU	8.63e+02	73.81	-8.73e+02	72.96
PB_PA_ALU	4.55e+02	140.00	-4.64e+02	152.03
Tanner Standard Cell Template	GND Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template	VDD Average Current (μA)	Ratio of Full Custom RaPiD GND Average Current/ FPGA SC Template
Not_reduced	5.75e+02	110.78	-5.75e+02	110.78
PA	5.71e+02	111.56	-5.71e+02	111.56
PB	5.92e+02	107.60	-5.92e+02	107.60
ALU	5.93e+02	107.42	-5.93e+02	107.42
PB_PA	6.34e+02	100.47	-6.34e+02	106.52
PA_ALU	1.75e+03	36.40	-1.75e+03	39.81
PB_ALU	6.03e+02	105.64	-6.03e+02	105.64
PB_PA_ALU	1.56e+03	40.83	-1.56e+03	40.83

Table 6-20: Power numbers for both versions of templates with 8 bits and 14 buses. Note that PowerMill would not run FPGA SC PB.

The power results from Table 6-16 through Table 6-20 are very compelling. The full custom RaPiD consumes on average an order of magnitude more power than any of the templates. While the Tanner and FPGA standard cells were not developed as low power libraries, they consistently consume less power. This was unexpected. One possible explanation is the extensive use of poly-silicon in the full customs RaPiD design as a means of routing signals, leading to high parasitic capacitance. Also, since RaPiD is geared to high performance, a larger number of n-type transistors were used then p-type transistors. This design choice meant that keeper-gates were needed to pull up the network, which could further increase the consumption of power. Further investigation is required to come to a completely satisfactory answer.

7. Conclusions

As SOCs move into the mainstream, there is little doubt that FPGAs will play a major role in providing the post-fabrication modification that these devices will require. This presents some interesting opportunities for creating high performance FPGAs that are targeted at specific application domains, instead of random logic. To implement these new architectures in a timely fashion, automation of the design flow is a necessity.

Table 7-1 summarizes the results of automating the design through the use of standard cells by providing an easy reference for choosing a particular optimization to reach specific design goal. For example, if a design called for a reduction in area, and you do not need the functions that an adder can provide, than modifying ALUs to Adders would give you a 30% reduction in area, 40% increase in performance, and reduce power consumption by 40%.

	Improvements		
Basic Optimization	Area	Performance	Power
16 Bit to 8 Bit	2.0x	1.1x	1.1x
ALU to Adder	1.3x	1.4x	1.4x
Not Reduced to PA	1.1x	1.1x	0.8x
Not Reduced to PB	1.2x	1.1x	0.4x
Tanner SC to FPGA SC	1.2x	1.0x	2.0x
SC to Full Custom	**1.3x**	**1.2x**	**0.03x**

Table 7-1 Benefit gained by implementing a design for optimization.

In this work we have shown that automation of layout generation for domain specific FPGAs is possible. We have further shown that as a target application domain narrows, the savings gained from removing unused logic from a design enables a standard cell method of layout generation to approach that of a full custom layout in area and performance, and in some cases surpass them. Finally, by adding to a standard cell library a few key cells that are used extensively in FPGAs, great improvements can be achieved.

8. Future Work

In the near future, we would like to replicate our work using various standard cell libraries. For example, a low power library would be especially beneficial, since low power consumption is becoming increasingly important as technology scales. We would also like to truly automate all of the tools, removing any manual intervention.

In addition, we are going to begin exploring the template reduction method of automating layout. We believe that this method would further improve upon the results presented and have the potential to create designs that achieve a performance level that is near ASICs.

Finally, we will begin to develop circuit generators to create designs that are of higher quality than ones created by the standard cell method. Therefore, we would further close the gap between our generated architectures and ASICs.

9. Acknowledgments

The author would first like to thank Prof. Herman Schmit for proposing the construction of FPGAs from standard cells. The author would also like to thank the RaPiD team at the CS department of the University of Washington, especially Chris Fisher who created the layout of the original RaPiD I. The author would finally like to thank Larry McMurchie who spent countless hours getting all of the tools in the tool-flow up and running, and Joan Phillips for all of her support.

This research was funded by grants from the National Science Foundation (NSF) and by the Defense Advanced Research Projects Agency (DARPA).

10. Appendix

10.1. Base Verilog Code of RaPiD

```
//
//ALU
//
module alu(Ina, Inb, Cin, Arith, Function3, Function2, Function1, Function0, Cout, Overflow, Zero, Sign,
Out);
input [15:0] Ina;
input [15:0] Inb;
input Cin;
input Arith;
input Function3;
input Function2;
input Function1;
input Function0;
output Cout;
output Overflow;
output Zero;
output Sign;
output [15:0] Out;
reg Cout, Overflow, Zero, Sign;
reg [15:0] Out;

always @(Cin or Ina or Inb or Arith or Function3 or Function2 or Function1 or Function0)
begin
casex ({Arith, Function3, Function2, Function1, Function0})
        5'b00000: {Cout,Out} = {1'b0,Ina};
        5'b00001: {Cout,Out} = {1'b0,Ina|Inb};
        5'b00010: {Cout,Out} = {1'b0,Ina&~Inb};
        5'b00011: {Cout,Out} = {1'b0,Ina^Inb};
        5'b00100: {Cout,Out} = {1'b0,Ina|~Inb};
        5'b00101: {Cout,Out} = {1'b0,{16{1'b1}}};
        5'b00110: {Cout,Out} = {1'b0,~Inb};
        5'b00111: {Cout,Out} = {1'b0,~(Ina&Inb)};
        5'b01000: {Cout,Out} = {1'b0,Ina&Inb};
        5'b01001: {Cout,Out} = {1'b0,Inb};
        5'b01010: {Cout,Out} = {1'b0,{16{1'b0}}};
        5'b01011: {Cout,Out} = {1'b0,~Ina&Inb};
        5'b01100: {Cout,Out} = {1'b0,Ina~^Inb};
        5'b01101: {Cout,Out} = {1'b0,~Ina|Inb};
        5'b01110: {Cout,Out} = {1'b0,~(Ina|Inb)};
        5'b01111: {Cout,Out} = {1'b0,~Ina};

        5'b10000: {Cout,Out} = {1'b0,Ina}+Cin;
        5'b10001: {Cout,Out} = {1'b0,Ina|Inb}+Cin;
        5'b10010: {Cout,Out} = {1'b0,(Ina&Inb)}+{1'b0,Ina}+Cin;
        5'b10011: {Cout,Out} = {1'b0,Ina}+{1'b0,Inb}+Cin;
        5'b10100: {Cout,Out} = {1'b0,Ina|~Inb}+Cin;
        5'b10101: {Cout,Out} = {1'b0,{16{1'b1}}}+Cin;
        5'b10110: {Cout,Out} = {1'b0,(Ina&Inb)}+{1'b0,(Ina|~Inb)}+Cin;
        5'b10111: {Cout,Out} = {1'b0,(Ina&Inb)}+{1'b0,{16{1'b1}}}+Cin;
```

```
		5'b11000: {Cout,Out} = {1'b0,(Ina&~Inb)}+{1'b0,Ina}+Cin;
		5'b11001: {Cout,Out} = {1'b0,(Ina&~Inb)}+{1'b0,(Ina|Inb)}+Cin;
		5'b11010: {Cout,Out} = {1'b0,Ina}+{1'b0,Ina}+Cin;
		5'b11011: {Cout,Out} = {1'b0,Ina}+{1'b0,(Ina|Inb)}+Cin;
		5'b11100: {Cout,Out} = {1'b0,Ina}+{1'b0,~Inb}+Cin;
		5'b11101: {Cout,Out} = {1'b0,(Ina&~Inb)}+{1'b0,{16{1'b1}}}+Cin;
		5'b11110: {Cout,Out} = {1'b0,Ina}+{1'b0,(Ina|~Inb)}+Cin;
		5'b11111: {Cout,Out} = {1'b0,Ina}+{1'b0,{16{1'b1}}}+Cin;
endcase

Overflow = (Ina[15]&Inb[15]&~Out[15])|(~Ina[15]&~Inb[15]&Out[15]);
Zero = ( (Out == 0) && (Cout != 1) ) ? 1 : 0;
Sign = Out[15];

end

endmodule

//
//Delay_reg
//
module delay_reg(In, Delay, CLK, Out);

input In;
input [1:0] Delay;
input CLK;
output Out;
reg Out;

reg delay_reg_one;
reg delay_reg_two;
reg delay_reg_three;

always @(In or Delay or delay_reg_one or delay_reg_two or delay_reg_three)
begin
case (Delay)
		2'b00: begin Out = In; end
		2'b01: begin Out = delay_reg_one; end
		2'b10: begin Out = delay_reg_two; end
		2'b11: begin Out = delay_reg_three; end
endcase
end

always @(posedge CLK)
begin
		delay_reg_three = delay_reg_two;
		delay_reg_two = delay_reg_one;
		delay_reg_one = In;
end

endmodule

//
//Driver_three
//
module driver_three(In, Select, Out_bus);
```

```
// Port declarations
input In;
input [3:0] Select;
output [13:0] Out_bus;

assign Out_bus[0]  = In & (~Select[3] & ~Select[2] & ~Select[1] & ~Select[0]);
assign Out_bus[1]  = In & (~Select[3] & ~Select[2] & ~Select[1] & Select[0]);
assign Out_bus[2]  = In & (~Select[3] & ~Select[2] & Select[1] & ~Select[0]);
assign Out_bus[3]  = In & (~Select[3] & ~Select[2] & Select[1] & Select[0]);
assign Out_bus[4]  = In & (~Select[3] & Select[2] & ~Select[1] & ~Select[0]);
assign Out_bus[5]  = In & (~Select[3] & Select[2] & ~Select[1] & Select[0]);
assign Out_bus[6]  = In & (~Select[3] & Select[2] & Select[1] & ~Select[0]);
assign Out_bus[7]  = In & (~Select[3] & Select[2] & Select[1] & Select[0]);
assign Out_bus[8]  = In & (Select[3] & ~Select[2] & ~Select[1] & ~Select[0]);
assign Out_bus[9]  = In & (Select[3] & ~Select[2] & ~Select[1] & Select[0]);
assign Out_bus[10] = In & (Select[3] & ~Select[2] & Select[1] & ~Select[0]);
assign Out_bus[11] = In & (Select[3] & ~Select[2] & Select[1] & Select[0]);
assign Out_bus[12] = In & (Select[3] & Select[2] & ~Select[1] & ~Select[0]);
assign Out_bus[13] = In & (Select[3] & Select[2] & ~Select[1] & Select[0]);

endmodule

//
// 16-to-1 multiplexer
//
module mux16_to_1(In_bus, Select, Out);

// Port declarations
input [13:0] In_bus;
input [3:0] Select;
output  Out;
//output declared as register
reg Out;

always @(In_bus or Select)
begin
  case (Select)
    4'b0000: Out = 1'b0;
    4'b0001: Out = 1'b1;
    4'b0010: Out = In_bus[0];
    4'b0011: Out = In_bus[1];
    4'b0100: Out = In_bus[2];
    4'b0101: Out = In_bus[3];
    4'b0110: Out = In_bus[4];
    4'b0111: Out = In_bus[5];
    4'b1000: Out = In_bus[6];
    4'b1001: Out = In_bus[7];
    4'b1010: Out = In_bus[8];
    4'b1011: Out = In_bus[9];
    4'b1100: Out = In_bus[10];
    4'b1101: Out = In_bus[11];
    4'b1110: Out = In_bus[12];
    4'b1111: Out = In_bus[13];
  endcase
end
```

```
endmodule

//
//sc_test_one
//
//This is the verilog version of the Rapid powertest design
//This is a one rapid-bit slice of the design up to the
// first alu.
//`include "delay_reg.v"
//`include "driver_three.v"
//`include "mux16_to_1.v"
module sc_test_one(In_bus, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2, Sel_delay3,
Sel_drive2, Sel_mux3, Sel_mux4, Out_bus);

// I/O port declerations
input [13:0] In_bus;
input CLK;
input [3:0] Sel_mux1;
input [1:0] Sel_delay1;
input [3:0] Sel_drive1;
input [3:0] Sel_mux2;
input [1:0] Sel_delay2;
input [1:0] Sel_delay3;
input [3:0] Sel_drive2;
input [3:0] Sel_mux3;
input [3:0] Sel_mux4;
output [1:0] Out_bus;

//Internal nets
wire muxw1, delayw1, muxw2, delayw2, delayw3;
wire [13:0] drivebus1, drivebus2;

//Instantiate the modules and connect them up
mux16_to_1 mux1(In_bus, Sel_mux1, muxw1);
delay_reg delay1(muxw1, Sel_delay1, CLK, delayw1);
driver_three drive1(delayw1, Sel_drive1, drivebus1);
mux16_to_1 mux2(drivebus1, Sel_mux2, muxw2);
delay_reg delay2(muxw2, Sel_delay2, CLK, delayw2);
delay_reg delay3(delayw2, Sel_delay3, CLK, delayw3);
driver_three drive2(delayw3, Sel_drive2, drivebus2);
mux16_to_1 mux3(drivebus2, Sel_mux3, Out_bus[0]);
mux16_to_1 mux4(drivebus2, Sel_mux4, Out_bus[1]);

endmodule

//
//sc_test_one_x_16
//
//This is the verilog version of the Rapid powertest design
//This is a 16 wide high slice of rapid up to the first alu
//`include "sc_test_one.v"
module sc_test_one_x_16(In_bus1, In_bus2, In_bus3, In_bus4, In_bus5, In_bus6, In_bus7, In_bus8,
In_bus9, In_bus10, In_bus11, In_bus12, In_bus13, In_bus14, In_bus15, In_bus16, CLK, Sel_mux1,
Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2, Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus);
```

```
// I/O port declerations
input [13:0] In_bus1;
input [13:0] In_bus2;
input [13:0] In_bus3;
input [13:0] In_bus4;
input [13:0] In_bus5;
input [13:0] In_bus6;
input [13:0] In_bus7;
input [13:0] In_bus8;
input [13:0] In_bus9;
input [13:0] In_bus10;
input [13:0] In_bus11;
input [13:0] In_bus12;
input [13:0] In_bus13;
input [13:0] In_bus14;
input [13:0] In_bus15;
input [13:0] In_bus16;
input CLK;
input [3:0] Sel_mux1;
input [1:0] Sel_delay1;
input [3:0] Sel_drive1;
input [3:0] Sel_mux2;
input [1:0] Sel_delay2;
input [1:0] Sel_delay3;
input [3:0] Sel_drive2;
input [3:0] Sel_mux3;
input [3:0] Sel_mux4;
output [31:0] Out_bus;

//Internal nets

//Instantiate the modules and connect them up
sc_test_one col1_bus1(In_bus1, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[1:0]);
sc_test_one col1_bus2(In_bus2, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[3:2]);
sc_test_one col1_bus3(In_bus3, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[5:4]);
sc_test_one col1_bus4(In_bus4, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[7:6]);
sc_test_one col1_bus5(In_bus5, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[9:8]);
sc_test_one col1_bus6(In_bus6, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[11:10]);
sc_test_one col1_bus7(In_bus7, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[13:12]);
sc_test_one col1_bus8(In_bus8, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[15:14]);
sc_test_one col1_bus9(In_bus9, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[17:16]);
sc_test_one col1_bus10(In_bus10, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[19:18]);
sc_test_one col1_bus11(In_bus11, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[21:20]);
sc_test_one col1_bus12(In_bus12, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[23:22]);
```

```
sc_test_one col1_bus13(In_bus13, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[25:24]);
sc_test_one col1_bus14(In_bus14, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[27:26]);
sc_test_one col1_bus15(In_bus15, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[29:28]);
sc_test_one col1_bus16(In_bus16, CLK, Sel_mux1, Sel_delay1, Sel_drive1, Sel_mux2, Sel_delay2,
Sel_delay3, Sel_drive2, Sel_mux3, Sel_mux4, Out_bus[31:30]);

endmodule

//
//sc_test_two
//
//This is the verilog version of the Rapid powertest design
//This is a one rapid-bit slice of the design up to the
// first alu.
//`include "delay_reg.v"
//`include "driver_three.v"
//`include "mux16_to_1.v"
module sc_test_two(In, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2, Out_bus);

// I/O port declerations
input In;
input CLK;
input [1:0] Sel_delay1;
input [1:0] Sel_delay2;
input [3:0] Sel_drive1;
input [3:0] Sel_mux1;
input [3:0] Sel_mux2;
output [1:0] Out_bus;

//Internal nets
wire delayw1, delayw2, muxw1, muxw2;
wire [13:0] drivebus1;

//Instantiate the modules and connect them up

delay_reg delay1(In, Sel_delay1, CLK, delayw1);
delay_reg delay2(delayw1, Sel_delay2, CLK, delayw2);
driver_three drive1(delayw2, Sel_drive1, drivebus1);
mux16_to_1 mux1(drivebus1, Sel_mux1, Out_bus[0]);
mux16_to_1 mux2(drivebus1, Sel_mux2, Out_bus[1]);

endmodule

//
//sc_test_two_x_16
//
//This is the verilog version of the Rapid powertest design
//This is a 16 wide high slice of rapid up to the first alu
//`include "sc_test_two.v"
module sc_test_two_x_16(In1, In2, In3, In4, In5, In6, In7, In8, In9, In10, In11, In12, In13, In14, In15,
In16, CLK, Sel_delay1,  Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2, Out_bus);

// I/O port declerations
```

```
input In1;
input In2;
input In3;
input In4;
input In5;
input In6;
input In7;
input In8;
input In9;
input In10;
input In11;
input In12;
input In13;
input In14;
input In15;
input In16;
input CLK;
input [1:0] Sel_delay1;
input [1:0] Sel_delay2;
input [3:0] Sel_drive1;
input [3:0] Sel_mux1;
input [3:0] Sel_mux2;
output [31:0] Out_bus;

//Internal nets

//Instantiate the modules and connect them up
sc_test_two col2_bus1(In1, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[1:0]);
sc_test_two col2_bus2(In2, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[3:2]);
sc_test_two col2_bus3(In3, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[5:4]);
sc_test_two col2_bus4(In4, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[7:6]);
sc_test_two col2_bus5(In5, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[9:8]);
sc_test_two col2_bus6(In6, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[11:10]);
sc_test_two col2_bus7(In7, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[13:12]);
sc_test_two col2_bus8(In8, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[15:14]);
sc_test_two col2_bus9(In9, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[17:16]);
sc_test_two col2_bus10(In10, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[19:18]);
sc_test_two col2_bus11(In11, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[21:20]);
sc_test_two col2_bus12(In12, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[23:22]);
sc_test_two col2_bus13(In13, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[25:24]);
sc_test_two col2_bus14(In14, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[27:26]);
```

```
sc_test_two col2_bus15(In15, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[29:28]);
sc_test_two col2_bus16(In16, CLK, Sel_delay1, Sel_delay2, Sel_drive1, Sel_mux1, Sel_mux2,
Out_bus[31:30]);

endmodule

//
//sc_test_three
//
//This is the verilog version of the Rapid powertest design
//This is a one rapid-bit slice of the design from the second alu
// to the end of the array
//`include "delay_reg.v"
//`include "driver_three.v"
//`include "mux16_to_1.v"
module sc_test_three(In, CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2, Sel_mux2,
Sel_delay3, Sel_delay4, Sel_drive3, Out_bus);

// I/O port declerations
input In;
input CLK;
input [1:0] Sel_delay1;
input [3:0] Sel_drive1;
input [3:0] Sel_mux1;
input [1:0] Sel_delay2;
input [3:0] Sel_drive2;
input [3:0] Sel_mux2;
input [1:0] Sel_delay3;
input [1:0] Sel_delay4;
input [3:0] Sel_drive3;
output [13:0] Out_bus;

//Internal nets
wire delayw1, muxw1, delayw2, muxw2, delayw3, delayw4;
wire [13:0] drivebus1, drivebus2;

//Instantiate the modules and connect them up
delay_reg delay1(In, Sel_delay1, CLK, delayw1);
driver_three drive1(delayw1, Sel_drive1, drivebus1);
mux16_to_1 mux1(drivebus1, Sel_mux1, muxw1);
delay_reg delay2(muxw1, Sel_delay2, CLK, delayw2);
driver_three drive2(delayw2, Sel_drive2, drivebus2);
mux16_to_1 mux2(drivebus2, Sel_mux2, muxw2);
delay_reg delay3(muxw2, Sel_delay3, CLK, delayw3);
delay_reg delay4(delayw3, Sel_delay4, CLK, delayw4);
driver_three drive3(delayw4, Sel_drive3, Out_bus);

endmodule

//
//sc_test_three_x_16
//
//This is the verilog version of the Rapid powertest design
//This is a 16 wide high slice of rapid from the last alu
// to the end of the array
```

```
//`include "sc_test_three.v"
module sc_test_three_x_16(In_bus[15:0], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2,
Sel_drive2, Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus1, Out_bus2, Out_bus3, Out_bus4,
Out_bus5, Out_bus6, Out_bus7, Out_bus8, Out_bus9, Out_bus10, Out_bus11, Out_bus12, Out_bus13,
Out_bus14, Out_bus15, Out_bus16);

// I/O port declerations
input [15:0] In_bus;
input CLK;
input [1:0] Sel_delay1;
input [3:0] Sel_drive1;
input [3:0] Sel_mux1;
input [1:0] Sel_delay2;
input [3:0] Sel_drive2;
input [3:0] Sel_mux2;
input [1:0] Sel_delay3;
input [1:0] Sel_delay4;
input [3:0] Sel_drive3;
output [13:0] Out_bus1;
output [13:0] Out_bus2;
output [13:0] Out_bus3;
output [13:0] Out_bus4;
output [13:0] Out_bus5;
output [13:0] Out_bus6;
output [13:0] Out_bus7;
output [13:0] Out_bus8;
output [13:0] Out_bus9;
output [13:0] Out_bus10;
output [13:0] Out_bus11;
output [13:0] Out_bus12;
output [13:0] Out_bus13;
output [13:0] Out_bus14;
output [13:0] Out_bus15;
output [13:0] Out_bus16;

//Internal nets

//Instantiate the modules and connect them up
sc_test_three col3_row1(In_bus[0], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus1);
sc_test_three col3_row2(In_bus[1], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus2);
sc_test_three col3_row3(In_bus[2], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus3);
sc_test_three col3_row4(In_bus[3], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus4);
sc_test_three col3_row5(In_bus[4], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus5);
sc_test_three col3_row6(In_bus[5], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus6);
sc_test_three col3_row7(In_bus[6], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus7);
sc_test_three col3_row8(In_bus[7], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus8);
sc_test_three col3_row9(In_bus[8], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus9);
```

```
sc_test_three col3_row10(In_bus[9], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus10);
sc_test_three col3_row11(In_bus[10], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus11);
sc_test_three col3_row12(In_bus[11], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus12);
sc_test_three col3_row13(In_bus[12], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus13);
sc_test_three col3_row14(In_bus[13], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus14);
sc_test_three col3_row15(In_bus[14], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus15);
sc_test_three col3_row16(In_bus[15], CLK, Sel_delay1, Sel_drive1, Sel_mux1, Sel_delay2, Sel_drive2,
Sel_mux2, Sel_delay3, Sel_delay4, Sel_drive3, Out_bus16);

endmodule

//
//sc_test
//
//`include "sc_test_header.v"
module sc_test(CLK, In_bus1, In_bus2, In_bus3, In_bus4, In_bus5, In_bus6, In_bus7, In_bus8, In_bus9,
In_bus10, In_bus11, In_bus12, In_bus13, In_bus14, In_bus15, In_bus16, Sel_mux1_col1,
Sel_delay1_col1, Sel_drive1_col1, Sel_mux2_col1, Sel_delay2_col1, Sel_delay3_col1, Sel_drive2_col1,
Sel_mux3_col1, Sel_mux4_col1, Sel_delay1_col2,  Sel_delay2_col2, Sel_drive1_col2, Sel_mux1_col2,
Sel_mux2_col2, Sel_delay1_col3, Sel_drive1_col3, Sel_mux1_col3, Sel_delay2_col3, Sel_drive2_col3,
Sel_mux2_col3, Sel_delay3_col3, Sel_delay4_col3, Sel_drive3_col3, Cin_alu1, Arith_alu1,
Function3_alu1, Function2_alu1, Function1_alu1, Function0_alu1, Cout_alu1, Overflow_alu1, Zero_alu1,
Sign_alu1, Cin_alu2, Arith_alu2, Function3_alu2, Function2_alu2, Function1_alu2, Function0_alu2,
Cout_alu2, Overflow_alu2, Zero_alu2, Sign_alu2, Out_bus1, Out_bus2, Out_bus3, Out_bus4, Out_bus5,
Out_bus6, Out_bus7, Out_bus8, Out_bus9, Out_bus10, Out_bus11, Out_bus12, Out_bus13, Out_bus14,
Out_bus15, Out_bus16);

// I/O port declarations
input CLK;
input [13:0] In_bus1;
input [13:0] In_bus2;
input [13:0] In_bus3;
input [13:0] In_bus4;
input [13:0] In_bus5;
input [13:0] In_bus6;
input [13:0] In_bus7;
input [13:0] In_bus8;
input [13:0] In_bus9;
input [13:0] In_bus10;
input [13:0] In_bus11;
input [13:0] In_bus12;
input [13:0] In_bus13;
input [13:0] In_bus14;
input [13:0] In_bus15;
input [13:0] In_bus16;
input [3:0] Sel_mux1_col1;
input [1:0] Sel_delay1_col1;
input [3:0] Sel_drive1_col1;
input [3:0] Sel_mux2_col1;
input [1:0] Sel_delay2_col1;
```

```
input [1:0] Sel_delay3_col1;
input [3:0] Sel_drive2_col1;
input [3:0] Sel_mux3_col1;
input [3:0] Sel_mux4_col1;
input [1:0] Sel_delay1_col2;
input [1:0] Sel_delay2_col2;
input [3:0] Sel_drive1_col2;
input [3:0] Sel_mux1_col2;
input [3:0] Sel_mux2_col2;
input [1:0] Sel_delay1_col3;
input [3:0] Sel_drive1_col3;
input [3:0] Sel_mux1_col3;
input [1:0] Sel_delay2_col3;
input [3:0] Sel_drive2_col3;
input [3:0] Sel_mux2_col3;
input [1:0] Sel_delay3_col3;
input [1:0] Sel_delay4_col3;
input [3:0] Sel_drive3_col3;
input Cin_alu1;
input Arith_alu1;
input Function3_alu1;
input Function2_alu1;
input Function1_alu1;
input Function0_alu1;
output Cout_alu1;
output Overflow_alu1;
output Zero_alu1;
output Sign_alu1;
input Cin_alu2;
input Arith_alu2;
input Function3_alu2;
input Function2_alu2;
input Function1_alu2;
input Function0_alu2;
output Cout_alu2;
output Overflow_alu2;
output Zero_alu2;
output Sign_alu2;
output [13:0] Out_bus1;
output [13:0] Out_bus2;
output [13:0] Out_bus3;
output [13:0] Out_bus4;
output [13:0] Out_bus5;
output [13:0] Out_bus6;
output [13:0] Out_bus7;
output [13:0] Out_bus8;
output [13:0] Out_bus9;
output [13:0] Out_bus10;
output [13:0] Out_bus11;
output [13:0] Out_bus12;
output [13:0] Out_bus13;
output [13:0] Out_bus14;
output [13:0] Out_bus15;
output [13:0] Out_bus16;

//Internal nets
```

```
wire [31:0] Out_bus_Col1;
wire [15:0] Out_Alu1;
wire [31:0] Out_bus_Col2;
wire [15:0] Out_Alu2;

//Instantiate the modules and connect them up
sc_test_one_x_16 Col1(In_bus1, In_bus2, In_bus3, In_bus4, In_bus5, In_bus6, In_bus7, In_bus8, In_bus9,
In_bus10, In_bus11, In_bus12, In_bus13, In_bus14, In_bus15, In_bus16, CLK, Sel_mux1_col1,
Sel_delay1_col1, Sel_drive1_col1, Sel_mux2_col1, Sel_delay2_col1, Sel_delay3_col1, Sel_drive2_col1,
Sel_mux3_col1, Sel_mux4_col1, Out_bus_Col1);

alu Alu1({Out_bus_Col1[30], Out_bus_Col1[28], Out_bus_Col1[26], Out_bus_Col1[24],
Out_bus_Col1[22], Out_bus_Col1[20], Out_bus_Col1[18], Out_bus_Col1[16], Out_bus_Col1[14],
Out_bus_Col1[12], Out_bus_Col1[10], Out_bus_Col1[8], Out_bus_Col1[6], Out_bus_Col1[4],
Out_bus_Col1[2], Out_bus_Col1[0]}, {Out_bus_Col1[31], Out_bus_Col1[29], Out_bus_Col1[27],
Out_bus_Col1[25], Out_bus_Col1[23], Out_bus_Col1[21], Out_bus_Col1[19], Out_bus_Col1[17],
Out_bus_Col1[15], Out_bus_Col1[13], Out_bus_Col1[11], Out_bus_Col1[9], Out_bus_Col1[7],
Out_bus_Col1[5], Out_bus_Col1[3], Out_bus_Col1[1]}, Cin_alu1, Arith_alu1, Function3_alu1,
Function2_alu1, Function1_alu1, Function0_alu1, Cout_alu1, Overflow_alu1, Zero_alu1, Sign_alu1,
Out_Alu1);

sc_test_two_x_16 Col2(Out_Alu1[0], Out_Alu1[1], Out_Alu1[2], Out_Alu1[3], Out_Alu1[4],
Out_Alu1[5], Out_Alu1[6], Out_Alu1[7], Out_Alu1[8], Out_Alu1[9], Out_Alu1[10], Out_Alu1[11],
Out_Alu1[12], Out_Alu1[13], Out_Alu1[14], Out_Alu1[15], CLK, Sel_delay1_col2,  Sel_delay2_col2,
Sel_drive1_col2, Sel_mux1_col2, Sel_mux2_col2, Out_bus_Col2);

alu Alu2({Out_bus_Col2[30], Out_bus_Col2[28], Out_bus_Col2[26], Out_bus_Col2[24],
Out_bus_Col2[22], Out_bus_Col2[20], Out_bus_Col2[18], Out_bus_Col2[16], Out_bus_Col2[14],
Out_bus_Col2[12], Out_bus_Col2[10], Out_bus_Col2[8], Out_bus_Col2[6], Out_bus_Col2[4],
Out_bus_Col2[2], Out_bus_Col2[0]}, {Out_bus_Col2[31], Out_bus_Col2[29], Out_bus_Col2[27],
Out_bus_Col2[25], Out_bus_Col2[23], Out_bus_Col2[21], Out_bus_Col2[19], Out_bus_Col2[17],
Out_bus_Col2[15], Out_bus_Col2[13], Out_bus_Col2[11], Out_bus_Col2[9], Out_bus_Col2[7],
Out_bus_Col2[5], Out_bus_Col2[3], Out_bus_Col2[1]}, Cin_alu2, Arith_alu2, Function3_alu2,
Function2_alu2, Function1_alu2, Function0_alu2, Cout_alu2, Overflow_alu2, Zero_alu2, Sign_alu2,
Out_Alu2);


sc_test_three_x_16 Col3(Out_Alu2, CLK, Sel_delay1_col3, Sel_drive1_col3, Sel_mux1_col3,
Sel_delay2_col3, Sel_drive2_col3, Sel_mux2_col3, Sel_delay3_col3, Sel_delay4_col3, Sel_drive3_col3,
Out_bus1, Out_bus2, Out_bus3, Out_bus4, Out_bus5, Out_bus6, Out_bus7, Out_bus8, Out_bus9,
Out_bus10, Out_bus11, Out_bus12, Out_bus13, Out_bus14, Out_bus15, Out_bus16);

endmodule
```

10.2. Layouts of Templates

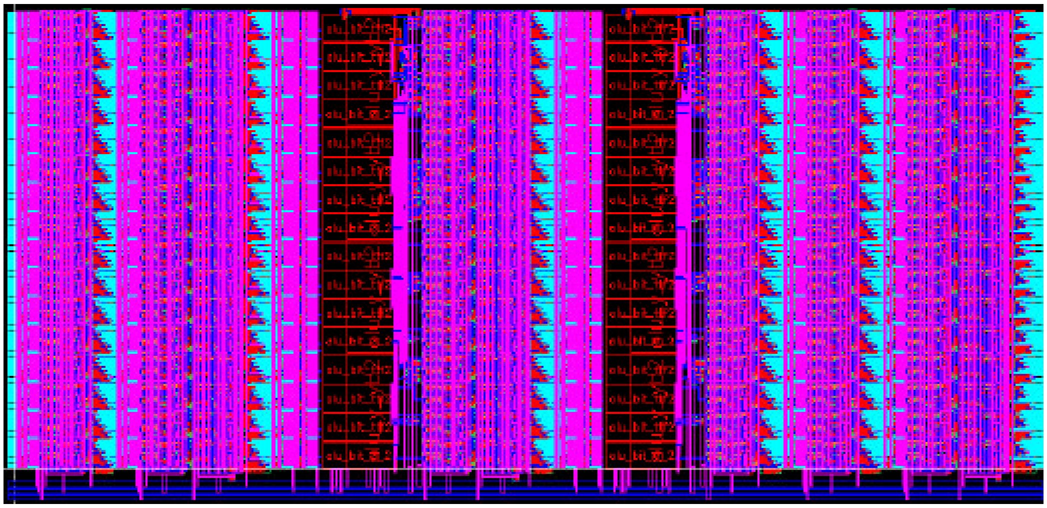


Figure 10-1 Full Custom RaPiD with an area of 680µm by 310µm. Notice the amount of red representing the extensive use of polysilicon.

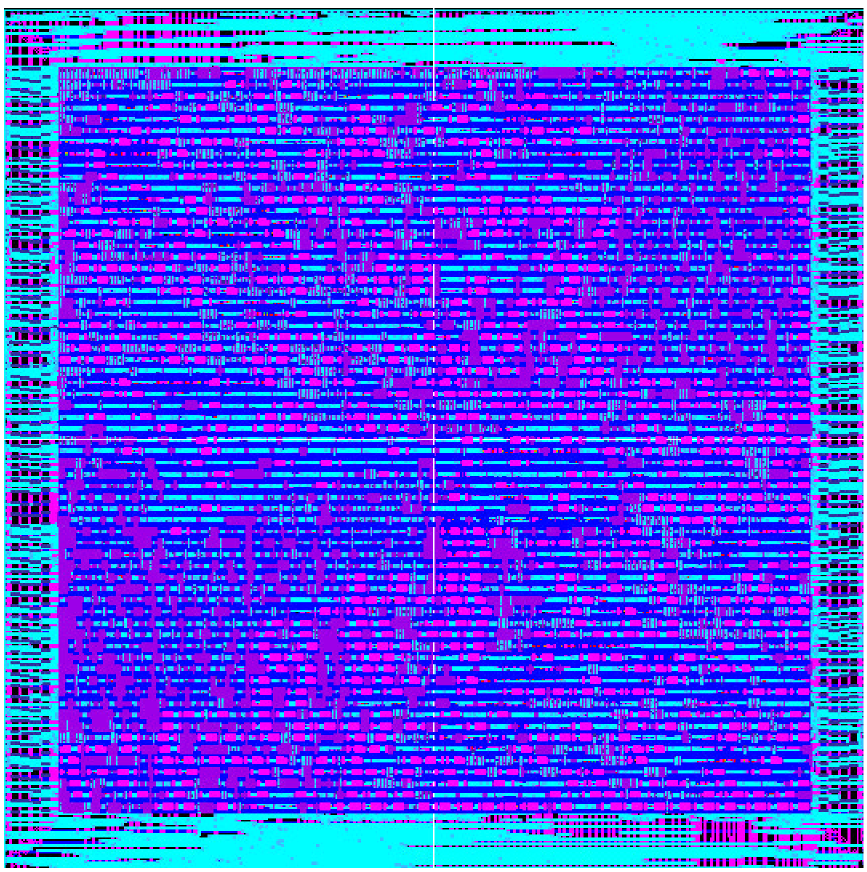


Figure 10-2 FPGA Standard Cell Template with an area of 709µm by 702µm.

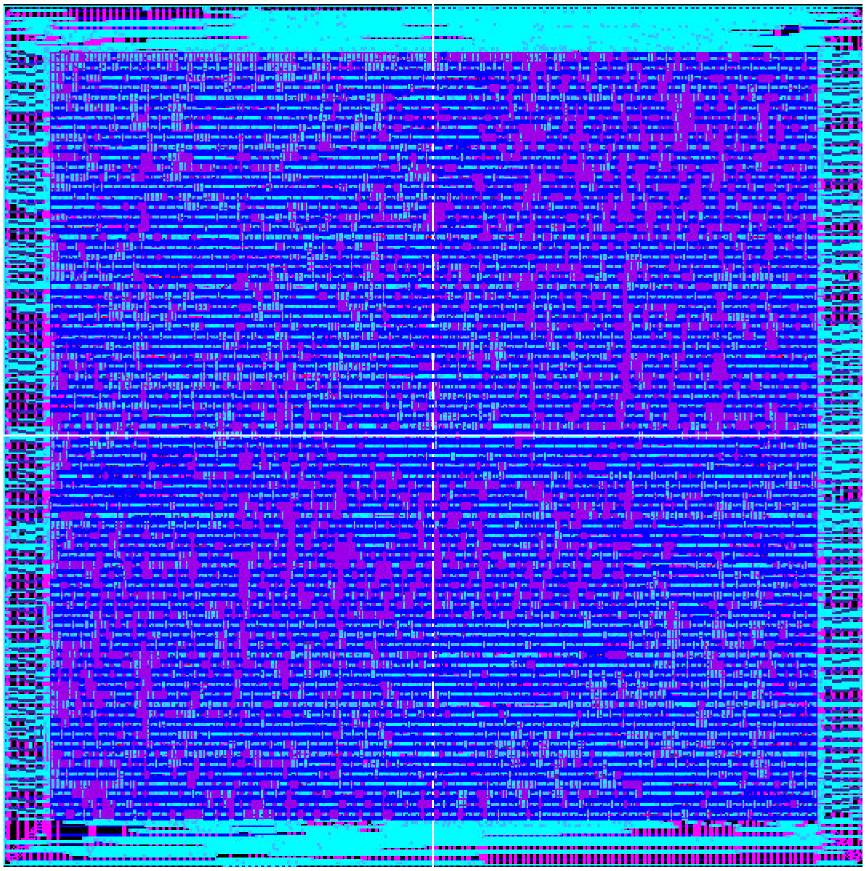


Figure 10-3 General standard cell template with an area of 764µm by 756µm.

10.3. Layouts of FPGA specific Standard Cells

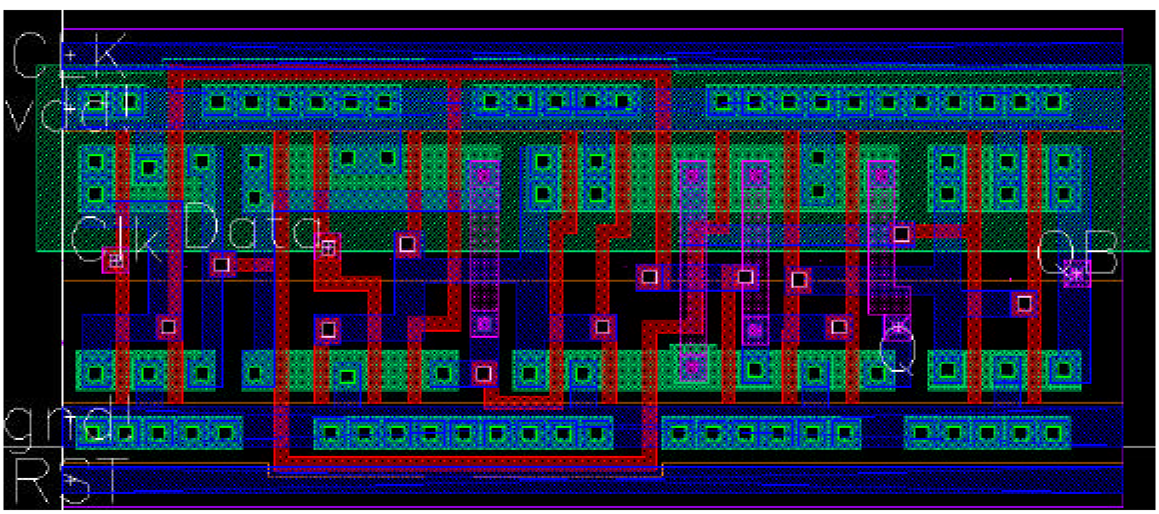


Figure 10-4 DFF FPGA specific standard cell.

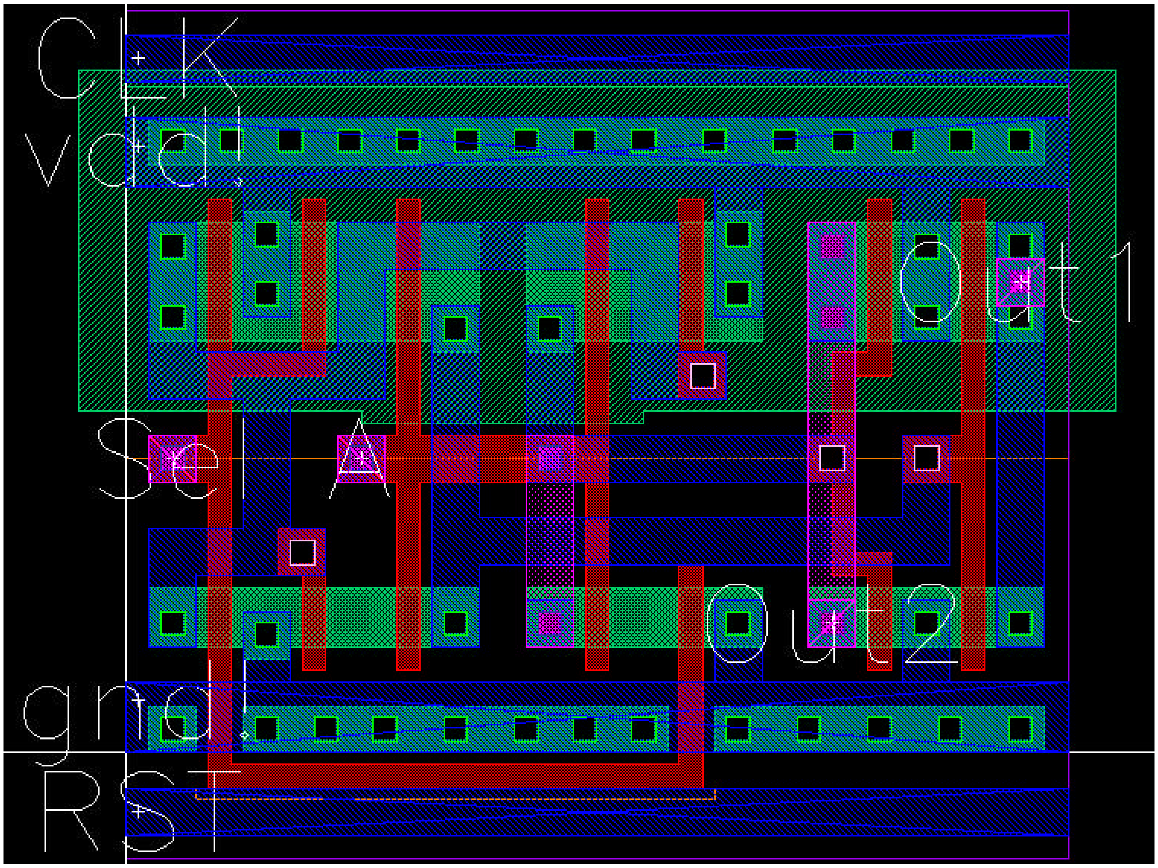


Figure 10-5 Driver 1-to-2 FPGA specific standard cell.

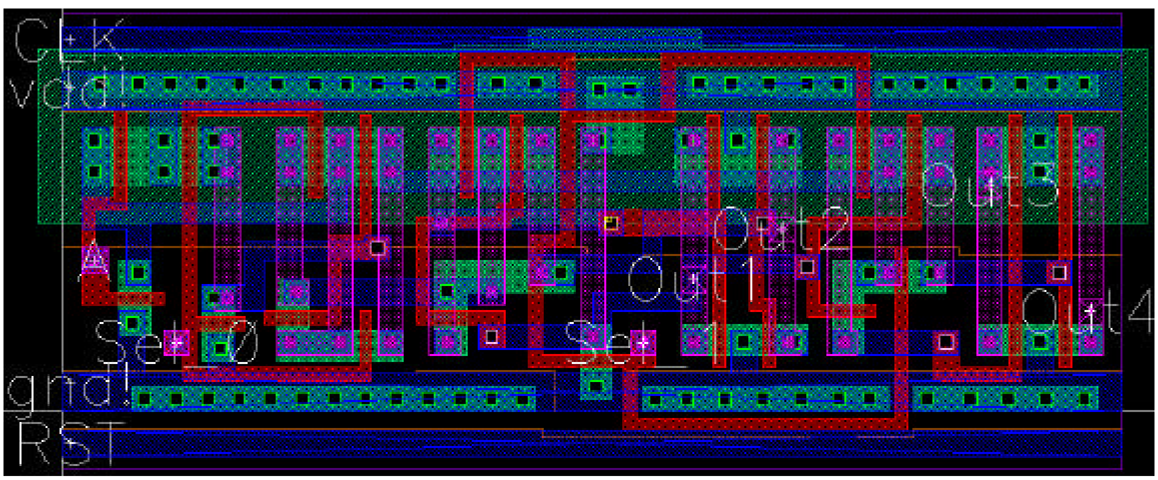


Figure 10-6 Driver 1-to-4 FPGA specific standard cell.

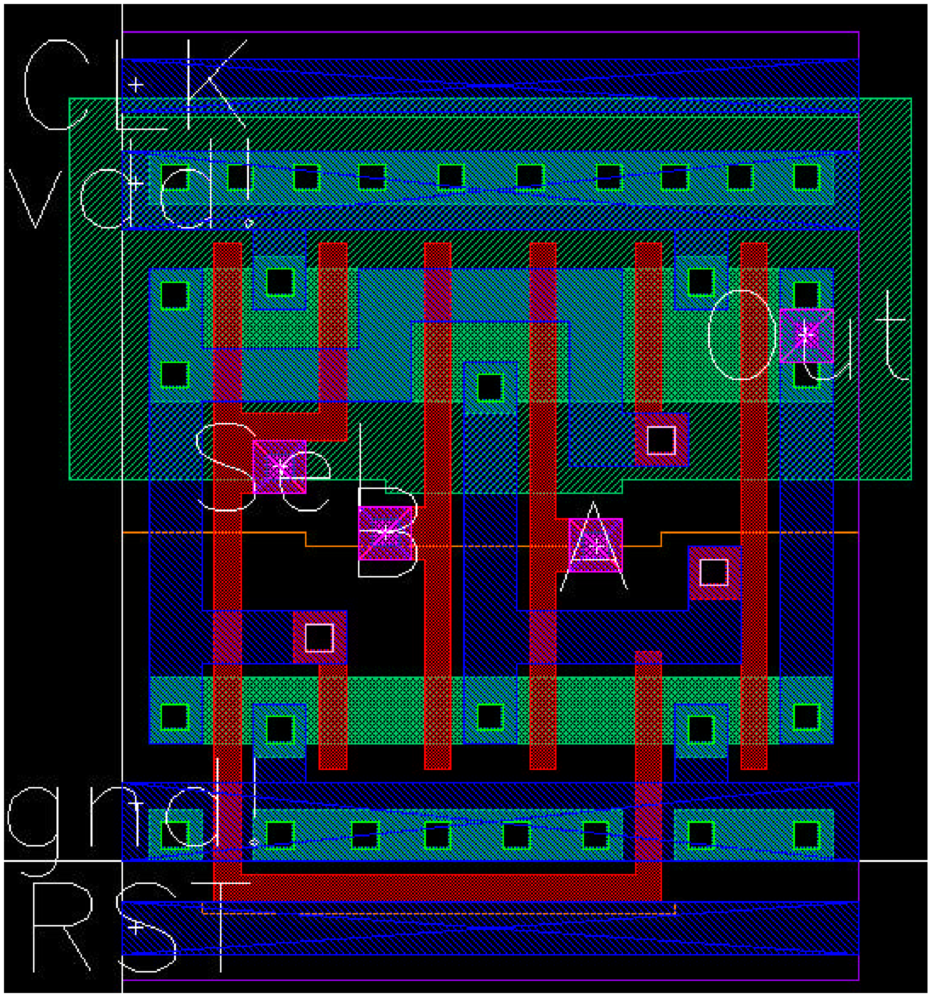


Figure 10-7 Mux 2-to-1 FPGA specific standard cell.

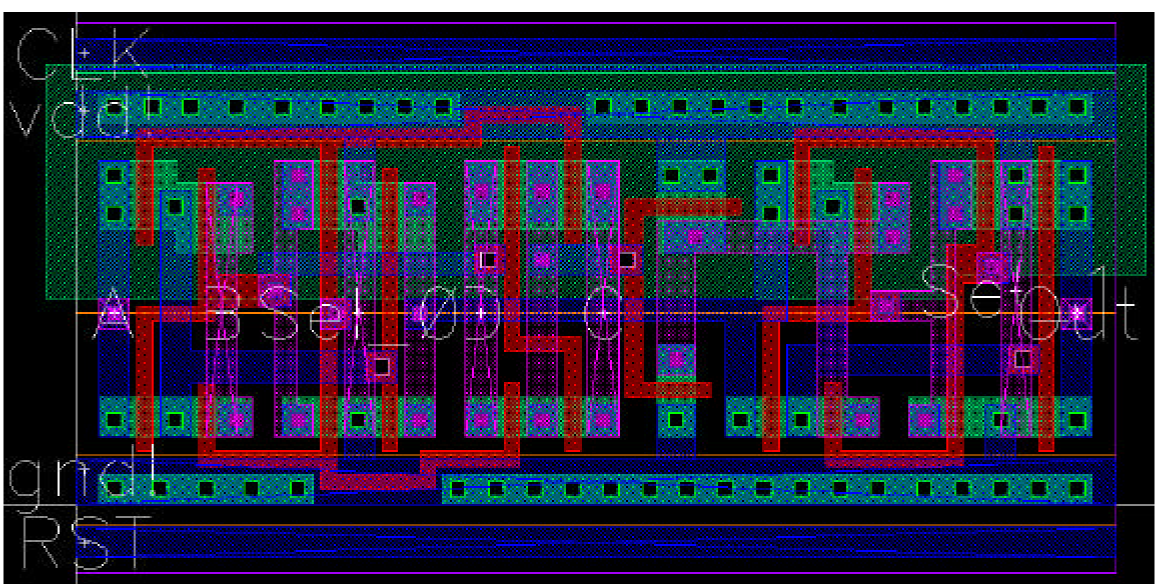


Figure 10-8 Mux 4-to-1 FPGA specific standard cell.

Bibliography

[1] Tilman Glökler, "System –on-a-Chip Case Study: ADSL-Receiver", http://www.ert.rwth-aachen.de/Projekte/VLSI/soc.html.

[2] C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath", *6th Annual Workshop on Field Programmable Logic and Applications*, 1996.

[3] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, "PipeRench: An Architecture and Compiler for Reconfigurable Computing", *IEEE Computer*, 2000.

[4] A. Abnous and J. M. Rabaey, "Ultra-low-power domain-specific multimedia processors," *Proc. of IEEE VLSI Signal Processing Workshop*, Oct. 1996.

[5] Neil H. E. Weste, Kamran Eshraghian, <u>Principles of CMOS VLSI Design: A Systems Perspective</u>, Addison-Wesely Publishing Company, 1993.

[6] Xilinx Inc., "About Xilinx", http://www.xilinx.com/company/about.htm.

[7] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", FCCM Conference, 2001.

[8] Peter Clarke, "Xilinx, ASIC Vendors Talk Licensing", http://www.siliconstrategies.com/story/OEG20010622S0091, 2001.

[9] Actel and Logicvision Partner to Deliver a Complete Embedded Test Solution for Varicore EPGA IP Cores, http://www.actel.com/pressroom/2001pr/LogicVision.html, 2001.

[10] Actel Corporation, "VariCore™ Embedded Programmability - Flexible by Design", http://varicore.actel.com/cgi-bin/varicore.cgi?page=overview.

[11] Brian Fuller, **"**LSI, Partners Dial Programmable Core into ASICs", http://www.eetimes.com/story/OEG20010611S0133, 2001.

[12] Advanced Products: Introducing LiquidLogic Embedded Programmable Logic Core http://www.lsilogic.com/products/asic/advanced_products.html.

[13] Anthony Cataldo, "Startup stakes out ground between FPGAs and ASICs" http://www.eetimes.com/story/OEG20010709S0075, 2001.

[14] http://www.easic.com/products/easicore018.html.

[15] 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.

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

[17] Synopsys, Inc., “Envisia Ultra Reference Manual”, version 2.2, 1999.

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

[19] Tanner Research, Inc., “Tanner CES Products”, http://www.tanner.com/CES/products/files_now/dit_std_cell.htm.

[20] Synopsys, Inc., “Epic Tools User Manual”

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

[22] Synopsys, Inc., “Powermill User Guide”, release 5.4, 2000.