Synthesis Techniques for Semi-Custom Dynamically Reconfigurable Superscalar Processors

BY

Jorge Ortiz

Submitted to the graduate degree program in Electrical Engineering and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:

Dr. Perry Alexander: Chairperson

Dr. David Andrews: Co-Chair

Dr. Arvin Agah

Dr. Swapan Chakrabarti

Dr. James Sterbenz

Dr. Kirk McClure

Date Defended

The Dissertation Committee for Jorge Ortiz certifies
that this is the approved version of the following dissertation:

Synthesis Techniques for Semi-Custom Dynamically Reconfigurable Superscalar Processors

Committee:

Dr. Perry Alexander: Chairperson

Dr. David Andrews: Co-Chair

Dr. Arvin Agah

Dr. Swapan Chakrabarti

Dr. James Sterbenz

Dr. Kirk McClure

Date Approved

Abstract

The accelerated adoption of reconfigurable computing foreshadows a computational paradigm shift, aimed at fulfilling the need of customizable yet high-performance flexible hardware. Reconfigurable computing fulfills this need by allowing the physical resources of a chip to be adapted to the computational requirements of a specific program, thus achieving higher levels of computing performance. This dissertation evaluates the area requirements for reconfigurable processing, an important yet often disregarded assessment for partial reconfiguration.

Common reconfigurable computing approaches today attempt to create custom circuitry in static co-processor accelerators. We instead focused on a new approach that synthesized semi-custom general-purpose processor cores. Each superscalar processor core's execution units can be customized for a particular application, yet the processor retains its standard microprocessor interface. We analyzed the area consumption for these computational components by studying the synthesis requirements of different processor configurations. This area/performance assessment aids designers when constraining processing elements in a fixed-size area slot, a requirement for modern partial reconfiguration approaches. Our results provide a more deterministic evaluation of performance density, hence making the area cost analysis less ambiguous when optimizing dynamic systems for coarse-grained parallelism.

The results obtained showed that even though performance density decreases with processor complexity, the additional area still provides a positive contribution to the aggregate parallel processing performance. This evaluation of parallel execution density contributes to ongoing efforts in the field of reconfigurable computing by providing a baseline for area/performance trade-offs for partial reconfiguration and multi-processor systems.

I dedicate this dissertation to the three most important women in my life.

To my mother, who shaped me into the man I am today.

To my sister, for encouraging me to touch the sky.

To my fiancée, for sharing her life with me.

Acknowledgments

I would like to thank my committee members for helping me decide on an area of focus for my education through their coursework.

A special thanks to Dr. David Andrews, my advisor, for initiating me into research, and allowing me enough freedom to choose a project of my liking.

I wish to also thank my labmates for their insights and feedback for the implementation of this project, especially Jason, whose clever solutions saved me weeks of work.

Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

1.1 Background

Huge advancements made in general-purpose computing have created several paradigm shifts within the field of computer architecture over the last three decades. The great improvement of the individual components of computing systems, such as memory, processing, and communication, led to a consequential improvement of the systems as a whole. Microprocessor availability pushed the development of personal computers, and the wide acceptance of the latter drove the research to increase performance and productivity. Studies on the importance of instruction sets led to the emergence of architectures tailored toward unit-cycle base instructions (RISC) and multi-cycle complex instructions (CISC). Instruction pipelining increased throughput, while alternative ways to solve data-dependencies and conditional branch instructions were unveiled. Deeper pipelining became a mainstream practice for its consequential advantage of producing higher clock frequencies. To reduce pipeline stalling penalties, out-of-order execution and branch prediction were added to the architectures. The inherent timing differences of heterogeneous instructions were exposed, giving birth to dedicated execution units.

As hardware became faster, smaller, and cheaper, it became an affordable resource. Application-specific hardware became common practice for custom circuitry that demanded dedicated computation. For general-purpose processors, the conservative utilization of hardware was sacrificed to move away from single-instruction, single data stream (SISD) architectures and exploit instruction, data and thread parallelism. Specialized single-instruction, multiple data stream (SIMD) processors appeared to map onto parallelizable problem domains, such as array and vector processing, and were closely followed by processors whose circuitry allowed for multiple instructions per cycle (VLIW). Returning to its roots, general-purpose computing inherited supercomputer techniques used to exploit instruction-level parallelism in order to become superscalar processors. Tomasulo's algorithm [1] emerged to resolve data dependencies in varying operational latency execution units. Thus, Tomasulo's algorithm gave birth to register renaming, a now common performance-enhancing technique both in commercial and scientific computing.

Throughout this whole era, both application-specific and general-purpose static processors were designed, manufactured, and used, followed by their inevitable fall to disuse and obsolescence, as new product-lines and processors were implemented. At the same time, the evolution of software code by means of high abstraction-level programming showed how important flexibility and reusability is in a market of primarily static processing. The notion of hardware with characteristics similar to its software counterpart suddenly became relevant.

1.2 Reconfigurable Computing

Because of the increased programmability of reconfigurable hardware, reconfigurable computing, a common ground for high-performance specialized systems and low-cost general-purpose processors, was created. The emergence of reconfigurable computing provided an alternate computing paradigm for engineers to exploit. This computing paradigm, which allows the use of flexible hardware, was first proposed by Estrin in the late 1950's [2]. Within the last 10 years, the technology base has matured to a point that enables us to use Estrin's original vision, by statically configuring and allocating reconfigurable components.

Reconfigurable computing invited interesting insights into the decades-old struggle between application-specific and general computing. Initially, tightly-coupled reconfigurable co-processing units could be added as extensions to the general-purpose Central Processing Unit (CPU) instruction set. They provided performance boosts by mapping portions of the application onto a faster hardware circuit They also provided compile-time versatility, by allowing different applications to be compiled and synthesized onto a single board. Current trends continue along this path and move the reconfigurable fabric even closer to the general-purpose CPU. These trends seek to decrease the communication costs of moving data between the CPU and field programmable gate arrays (FPGA) across a general-purpose system bus, allowing data to be shared through the CPU's register set. Research work such as GARP [3] went as far as defining a new CPU architecture that embedded reconfigurable fabric within the CPU. The reconfigurable fabric could then be configured as a specialized execution unit within the CPU. Unfortunately this approach cannot exploit sufficient amounts of parallelism to justify the overhead of moving data and instructions into and out of the reconfigurable fabric.

In contrast to the GARP approach of exploiting fine-grained parallelism through loop unrolling, researchers are now investigating approaches to support coarse-grained parallelism within FPGAs. Current work investigates thread level parallelism, such as research done by Vuletic [4, 5] for static systems, Platzner [6, 7] for dynamic systems, and Hybrid Threads [8, 9, 10] for hybrid systems. These approaches follow modern methods of allowing programmers to express applications as sets of independent threads that can communicate through shared memory or message-passing paradigms.

1.3 General-purpose specialized processor

Our approach, named Hephaestus, follows the trend of supporting thread-level parallelism within a computational core, but at the same time allows the exploitation of instruction-level parallelism through modern computer architecture approaches. The processor core can statically reside in the reconfigurable fabric and allow different configurations in terms of instruction issue, execution units, reservation stations, and common data bus width. These configurations can be tailored to try to achieve an optimal design in which an application can achieve maximum number of instructions per cycle.

While multiple processors try to integrate reconfigurable logic into the processor itself [3, 11, 12, 13], Hephaestus takes an orthogonal approach by implementing the processor onto the reconfigurable logic. Other soft-core processors have been mapped onto reconfigurable logic [14, 15, 16, 17, 18], providing scalar single-issue performance. Implementing a superscalar architecture in reconfigurable fabric is much slower than the processor's built-in execution units for standard computations, such as floating point and complex integer arithmetic [3, 11]. However, we can expect this timing cost to be amortized by flexibility of design, ability to increase the number of instructions issued, and rising clock frequencies and density for reconfigurable logic.

The main motivation for this project was studying the relationship between the increasing density advantage of FPGAs [19] versus the area cost associated with superscalar issue width growth, in which overhead increases quadratically [13]. Another one was examining the number of opportunities for exploiting instruction-level parallelism with different processor configurations.

1.4 Processor characteristics

A key element for the flexibility of the number of Execution Units (EU) and their associated reservation stations is the adaptive Tomasulo's algorithm, which controls instruction issue and completion to/from the EUs. This register-renaming algorithm is implemented in hardware and uses the processor's configuration as parameters. At synthesis time, the system assigns tags to all available EUs for name resolution. Register renaming, with its well-documented advantages, is used when issuing instructions, broadcasting results, and completing instructions. The processor handles the non-trivial tasks of multi-instruction issue, name resolution, and instruction organization, dispatch, execution, broadcast and completion. These were implemented for a variable number of instruction widths, numbers and types of EUs, and sizes and bandwidth of the register file.

To get the comparable speedups of custom-dedicated hardware, Hephaestus allows application-specific user-logic to be implemented in its architecture. A user can reserve one or multiple execution units for such purpose, similar to the way a typical co-processing unit architecture uses a co-processor to enhance performance. A restriction placed on the user-logic is that it must adhere to the EU interface to fit in the system.

1.5 Processor evaluation

The different configurations of the Hephaestus processor provides great system-component versatility, and consequently, vastly different use of reconfigurable fabric resources. We synthesized an array of configurations, varied by discrete differences in the system's independent variables:

- Instruction issue
- Number of execution units
- Common data bus width
- Register file size
- Register file memory organization
- Reservation station organization
- Reorder buffer size

The effect of these variables on the system's architecture is further explained in section 4.1. The dependent variables were then evaluated from the resulting maximum system clock frequency and resource use from synthesizing the permutations of the independent variables.

1.6 Toward dynamic run-time reconfiguration

Finally, for the purposes of dynamic reconfiguration, an additional task was to partition the superscalar processor and its execution unit configuration into regularly-sized areas. This allowed for synthesis of individual configurations, given a base system. These pre-synthesized configurations are prime candidates to be replaced using run-time reconfiguration, provided the host reconfigurable platform supports the communication needs between static and dynamic modules of the design.

1.7 Dissertation Organization

The rest of this dissertation is organized as follows: The objectives of the project are presented in Chapter 2; related work is presented in Chapter 3, while Chapter 4 describes the implementation methodology. Chapter 5 explains the evaluation of the project and the results are shown on Chapter 6. Finally, Chapter 7 places this work in the current state of the field.

Chapter 2

Objectives

This dissertation investigated new directions for synthesizing semi-custom circuits for dynamically reconfigurable systems. Our approach was to focus on synthesis techniques for computational-cores that can be confined within regularly sized reconfiguration slots.

This enabled us to pursue the main objectives of the Hephaestus project:

- Create a configurable superscalar processor
- Synthesize the processor onto reconfigurable fabric
- Monitor resource use under different processor configurations
- Evaluate resource use and area expenditure
- Constrain the area for slotted synthesis

2.1 Create a configurable superscalar processor

In our approach we examined processor structures that allow a variety of execution Units (EU's) to operate in unison and exploit intrinsic instruction level parallelism (ILP).

For our superscalar processor, we explored classic computer architecture techniques that allow multiple instructions to complete during any given clock cycle. Pipelining, with its associated control logic, was used to allow our techniques to exploit the performance increases from technology advancements. This is an important consideration and motivation within field programmable gate array (FPGA) reconfigurable devices, which currently follow Moore's law and will benefit from higher clock frequencies over the next decade. The FPGA level of technology is at a level of speed and density comparable to 1980's CMOS fabrication techniques, promising to evolve at the same or faster rate [19].

Hardware description languages (HDL) allow us to define hardware behavior in terms of software code. We employed VHDL to define the model, interface, interconnect and parameters of our processor. With it, we can change the entire processor's configuration by simply altering the system parameters to match our desired setup. This is another motivation of the project, to allow a designer to easily balance the flexibility offered by a general-purpose programmable processor and the benefits of custom circuits.

Our semi-custom approach further enables a system designer to include user-logic into the Hephaestus processor. One (or more) of the generic execution units included with the system can be replaced as needed by a designer, to accomplish specific tasks on the target reconfigurable platform.

2.2 Processor's synthesis onto reconfigurable fabric

Reconfigurable computing allowed us to define flexible system specifications for hardware, and in addition, map the configured hardware onto its fabric.

We aimed not only to define the hardware specification but also to make it synthesizable onto modern, commercially-available reconfigurable architectures. Hence, Hephaestus' correct behavior is not only established by simulation, but also by synthesizing onto these architectures.

The HDL source code for the reconfigurable processor was kept as technology-independent as possible. This permitted the whole project to be easily portable between different reconfigurable platforms. Design portability is an important concern in reconfigurable computing, since cross-platform implementation is directly dependent on it. Lack of portability sometimes cripples development time when migration between devices is needed, even within the same device family [20]. HDL coding styles were used to make the design independent of technology or vendor-specific pragmas.

2.3 Monitoring resource use under different configurations

The component configuration of our processor can have a startling impact on performance, however, so can the nature of the program running on the provided processor configuration. Performance maximization in a reconfigurable system is analogous to customization. Thus, depending on the program's requirements, our reconfigurable processor can provide an appropriate configuration. This customization can enhance and optimize the worst, average and best case computational scenarios as required, by adjusting structural composition.

The varying types, numbers and complexity of the resources used by these structural composition were collected for the various permutations of processors configurations, to be analyzed into comprehensive relationships between requirements and necessary resources.

2.4 Evaluating resource use and area expenditure

The flexibility of most reconfigurable architectures, especially FPGAs, is usually associated and exploited from its homogeneous arrays of logic gates [21, 22]. Nowadays, they tend to be more heterogeneous, often including memory units, specialized processing elements, and even embedded processors cores [14, 23]. Synthesizing tools referee the use of device-specific resources for performance gains, but most of the system logic is mapped onto homogeneous reconfigurable fabric. The evaluation of this area consumption makes a metric suitable for resource use under different structural compositions.

The analysis portion of this work focuses on quantifying this performance versus generalization trade off. Clearly, performance benefits of a semi-custom processor structure may fall short of those gained by a fully-custom implementation. However, the area, reconfiguration overhead and scheduling implementation costs of the fully custom approach outweigh the loss of peak performance gains for semi-custom synthesis. By collecting performance needs, and then correlating them with the associated resource use, we evaluated the performance benefits of using our processor.

2.5 Constraining area for slotted synthesis

We do not claim to achieve dynamic, or even partial, reconfiguration in our project. Dynamic reconfiguration is still evolving, in search of the right problem for its benefits to solve [24]. However, we did show that it can be adapted to different sized slotted areas through (mostly vendor-specific) synthesis techniques. This property, in conformity to reconfiguration techniques, can then support the necessary conditions to allow partial reconfiguration to be used for multiple configurations of our processor(s).

Hence, one could take an array of suitable processor configurations, synthesize them with the same area constraints in different board locations, and download one or more of these onto an FPGA board. In this way, our project is able to meet the current requirements for reconfigurable computing.

2.6 Thesis Statement

Our proposed objectives, mixed with the need to address area issues in reconfigurable computing, thus combine into our thesis statement:

> Synthesis techniques can confine soft-core processors to have a regularly sized area while enhancing parallelism. Evaluation of the performance density provides a baseline for area / performance trade-offs for partial reconfiguration and multi-processor systems.

2.7 Contributions

The end goal of Hephaestus is probing for computational performance gains while subsidizing reconfiguration costs through synthesis. Other specific objectives of the project were to attain competitiveness in performance, by using historically-significant advances in processor design, mapping and pipelining heterogeneous instructions to different EUs, increasing instruction throughput, and achieving higher clock frequencies. The methods for achieving applicability, adaptability, optimization and reconfigurability are further described in Chapter 4. The implementation of the reconfigurable superscalar processor architeture contributes to the field by:

- achieving a balance between flexibility offered by a general-purpose programmable processor and the benefits of custom circuits,
- creating a flexible, extensible processor that adapts to an application's computational needs, without the long implementation times needed for fully-custom systems, and
- establishing a hybrid solution that allows custom parallel logic within standard processor architecture framework.

Also notably, this work enhances modern reconfiguration techniques by defining and refining a set of synthesis constraints to facilitate a multi-threaded fusion of general-purpose and specialized computing. Application of specialized and general-purpose processing in the reconfigurable computing research area is detailed in Chapter 3. The data collection and analysis from processor performance and area use for different configurations contributes to the field by:

- evaluating the attainable parallel execution for custom applications in a reconfigurable processor framework,
- quantifying the area requirements for parallel-execution gains, and
- calculating and predicting the performance in a confined area for partially reconfigurable and multiprocessor systems.

Chapter 3

Related Work

3.1 Introduction

Increasing performance for a specific application has traditionally been achieved by customizing circuits for it. The gains achieved by customization are far superior to those achieved by modifying commercial off-the-shelf components. With dedicated hardware, the end product is usually an optimized system, meeting or exceeding the application's speed and area requirements.

Another way to enhance performance is by exploiting parallelism. The underlying idea is to think smarter, not harder. With parallelism, performance is gained from the decomposition of the problem into independent computations, and not from faster dedicated hardware. These computations can then be executed simultaneously, at the instruction, data or thread level. Early adopters of this paradigm included the CDC 6600 computer [25], which was the predecessor of pipelined Reduced Instruction Set Computers (RISC), and IBM's System 360/91 [26], which provided out-of-order instruction execution through floating point units executing in parallel with the main processor.

Custom circuitry with parallel computations combines the advantages of both approaches. However, each approach has disadvantages that compound each other. Custom circuits have lengthy development and time-to-market times, are not reusable, and are expensive in terms of non-recurring engineering and low-scale production costs.

Parallel computing consumes an increased area in hardware, has a complex design, and does not scale well. Custom circuitry with parallel computing generally results in a poor implementation for the large array of applications which require high performance, but can't sacrifice adaptability, reusability, programmability, area and cost.

Emerging technologies like reconfigurable computing can address the balance between performance and system cost. The wide variety of existing and proposed reconfigurable systems deal with different aspects of what constitutes performance, and what constitutes system cost. In this chapter we review these options and justify the motivations, objectives and methodology of the Hephaestus project.

3.2 ASIC technologies

A well-established method of meeting a program's computational demands is by using an Application-Specific Integrated Circuit (ASIC). ASICs allow the use of custom logic to tackle the demands mentioned before: optimization, area consumption and increased performance. However, the cost, inflexibility, and time-to-ship of ASICs do not make them a satisfactory solution in most of these cases [27, 28]. In fact, development costs for ASICs are approaching \$20 million for a 90-nm ASIC design with an embedded System-on-Chip (SoC), and \$40 million for the 45-nm version [29].

Non-recurring engineering costs comprise most of the ASIC's price tag. While the mass production of a finished ASIC is cheap, the costs associated with the development of the circuit are not. FPGAs solve this problem by allowing implemented circuits to be downloaded onto a single platform multiple times. They are becoming the default platform for reconfigurable computing and even ASIC development, because in sharp contrast to ASIC implementations, FPGAs can provide a SoC for a few hundred dollars (at the time of writing). This is a common reason why hardware designers turn to reconfigurable computing, either to design ASICs, to implement their application in an FPGA, or a hybrid of both [30].

But ASICs are not without merit. When writing about the future of reconfigurable systems, Scott Hauck, points out a law of FPGAs vs. ASICs [31]:

> For any FPGA-based application, there is an ASIC implementation of the system that is AT LEAST as fast, dense, power-efficient, and cheap in (very) high volumes as the FPGA-based solution.

Regardless, because of the inflexible limitations of ASICs, their hold on the hardware-design market is diminishing. Conversely, reconfigurable computing and FPGA's use are flourishing because of their flexibility, as seen in Figure 3.1. Thus, reconfigurable computing is an appropriate approach to deal with the defined problem, without the disadvantages of ASICs.

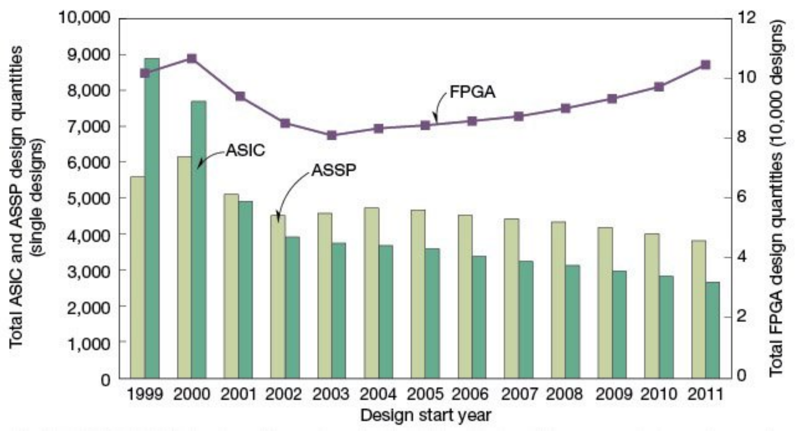


Figure 3.1: ASIC vs FPGA Market - Market trends show diminished implementation in ASICs and growing demand for FPGA systems [32].

3.3 Reconfigurable computing background

Multiple implementation solutions have addressed the performance and cost issue. These solutions provided alternative hardware-based benefits where efficiency, speed, power consumption, system cost, throughput, or other units of interest in particular applications and systems are concerned.

Recent years have seen a rise in Systems-on-Chip (SoC) [33, 34, 35], Hardware/-Software Co-Design [18, 36, 37], Hardware Accelerators [17, 38, 39, 40] and Reconfigurable Computing [41, 42, 43, 44]. Each of these solutions are a response to inadequate hardware mapping for efficiency to available computer infrastructures. This inadequacy generally stems from the not-surprising fact that domain-specific problems that demand high computational power can perform much better on specialized hardware.

As we established, ASICs are not the most appropriate platform for implementation, and alternative solutions to ASICs include the aforementioned ones. Systems-on-Chip have tightly-coupled microcontrollers and memory among other interface components. This design improves the functionality of the chip by adding peripherals. The

SoC peripherals act as common extensions to a variety of applications like counters, timers, and digital and analog functions. Hardware/Software Co-Design attempts to capture this added computational flexibility not only in SoC, but also in programmable and configurable processors. If these processors have a configurable fabric like that of a PLD or FPGA attached to them, then it would be a common option to add an application-specific hardware accelerators to boost performance. These hardware accelerators usually provide condensed multi-cycle computations for problems like digital signal processing, matrix operations, and fused multiplication/addition, and are loosely coupled with the processor itself.

The last alternative is reconfigurable computing. While not new, this technology is still in its adolescent stages, trying to establish an identity which helps define the paradigm shift into programmable hardware. Researchers in both academia and industry have addressed the possibilities and limitations of reconfigurable computing, creating systems which make use of its flexibility, reusability and programmability. These systems can then provide optimizations, speedups, and increased performance by allowing the parallel execution of data in reconfigurable fabric. This increased interest in reconfigurable systems explains the growth that FPGAs have had for the past 10 years. Its importance can also be seen in the growing market for FPGA-based solutions shown in Figure 3.2.

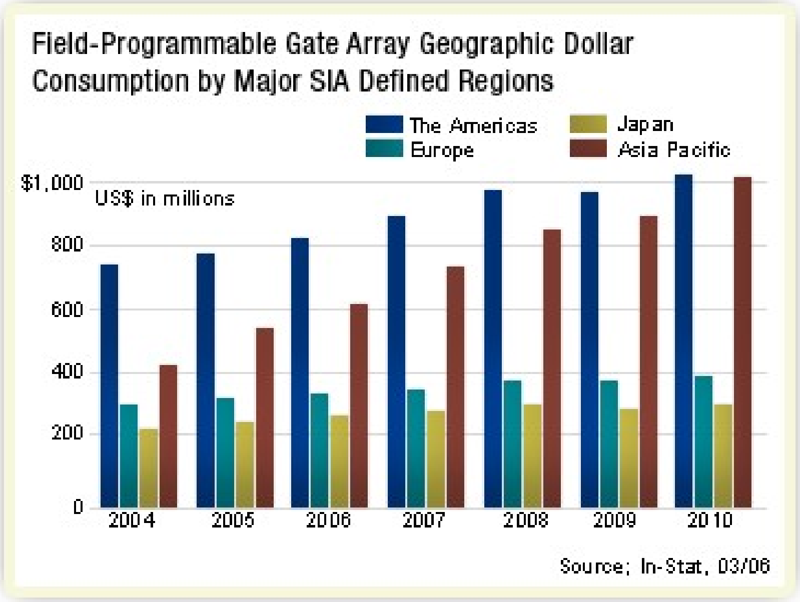


Figure 3.2: FPGA Dollar Consumption - FPGA market history and forecast [45].

The rise of available technologies to support reconfigurable computing enabled researchers to tackle the demands of higher computational power in a variety of methods, which we discuss next.

3.4 Types of reconfigurable computing

Reconfigurable computing allows several configurations that adapt to different computational environments. There are two main types of data processing in reconfigurable computing: Reconfigurable architectures and soft-core processors.

3.4.1 Reconfigurable architectures

Several reconfigurable architectures exist, the most prevalent being FPGAs. These architectures vary in their composition, with the three most important characteristics of reconfigurable architectures being granularity, structure, and reconfigurability.

3.4.1.1 Granularity

Granularity, the datapath width of the architecture, determines the number and size of the architecture's Processing Elements (PEs). Low levels of granularity are called fine-grained, while larger are coarse-grained. Granularities for these architectures vary between 1, 4, 8, 16 and 32 bits. Fine-grained architectures are the most flexible, allowing bit-routing capabilities, while coarse-grained ones provide area-efficiency for regular multi-bit datapaths.

3.4.1.2 Structure

Structure, the positioning and inter-communication of the architecture's PEs, establishes the channels necessary for communication among PEs. A common practice for hardware processing is trading one fast-clocked large CPU, for a number of lower-clocked and more efficient PEs [46]. The multiple PEs are then structured to encourage nearest-neighbor links, increasing their communication bandwidth. Routing resources are implemented in the reconfiguration fabric and used according to these links. Figure 3.3 shows some of the structures that reconfigurable architectures have taken.

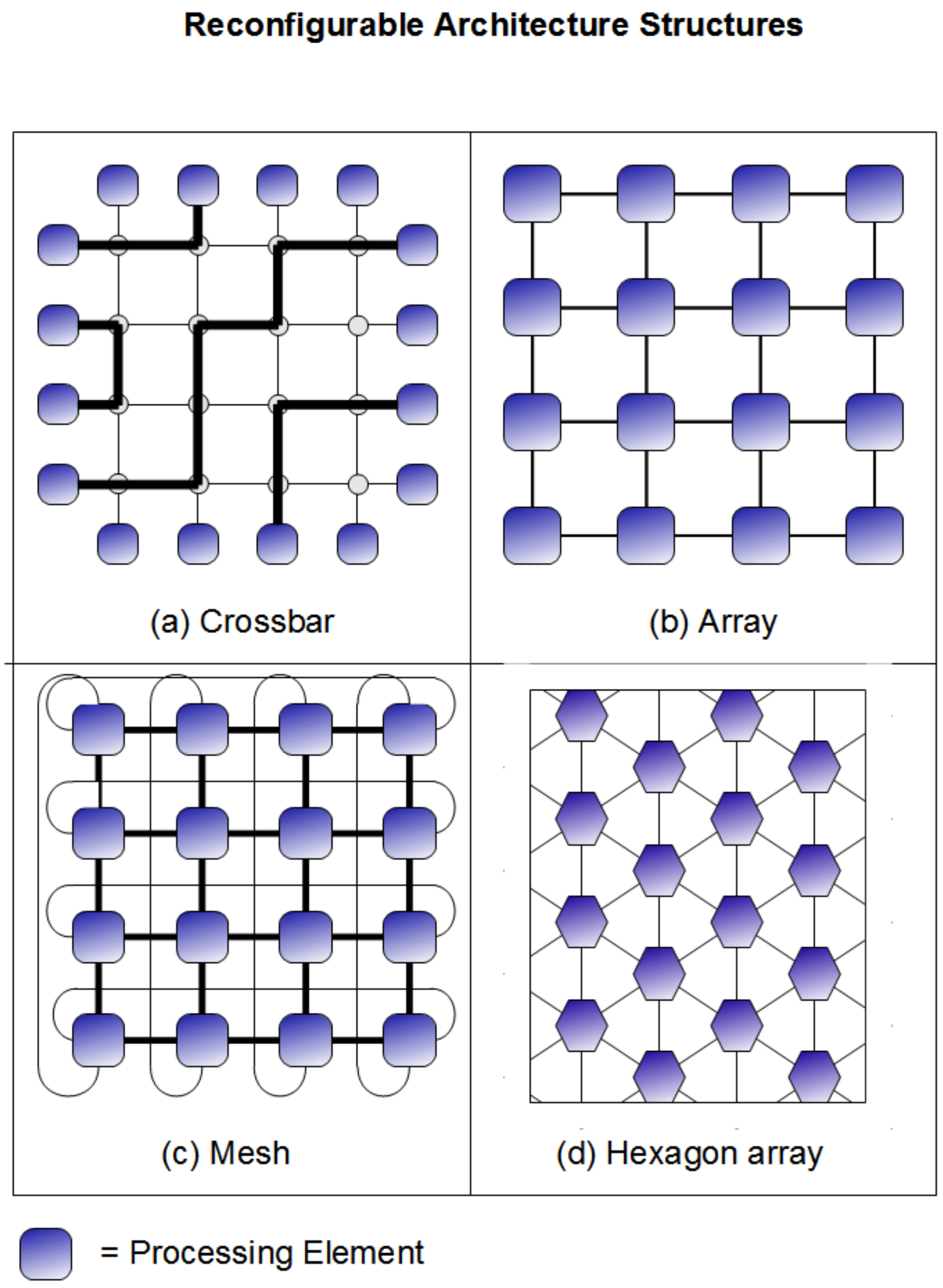
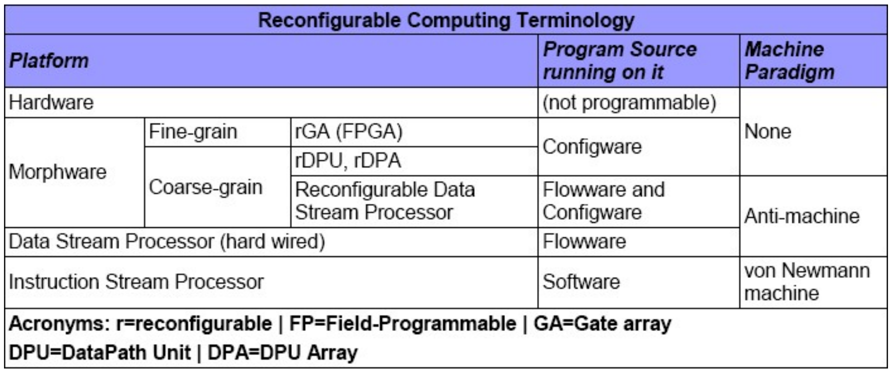


Figure 3.3: Regular Reconfigurable Architectures - Boxes represent processing elements, links are represented by lines. a) Crossbar allows for dynamic paths between PEs b) Arrays are static links c) Meshes provide circular struture d) Hexagon arrays (like Chess [47]) allow links with 6 neighbors.

Once the structure is selected, the architectures can then reconfigure PEs functionality and the communication links between PEs to achieve a specific task. Depending on the granularity used, they can be labeled in particular ways. To help clarify the loosely used terminology for “reconfigurable architectures” and other terms used in this proposal, we use the nomenclature proposed by morphware.net (Figure 3.4), which disambiguates labels used in our research area.



Reconfigurable Computing Terminology				
Platform			***Program Source running on it***	***Machine Paradigm***
Hardware			(not programmable)	None
Morphware	Fine-grain	rGA (FPGA)	Configware	
	Coarse-grain	rDPU, rDPA		
		Reconfigurable Data Stream Processor	Flowware and Configware	Anti-machine
Data Stream Processor (hard wired)			Flowware	
Instruction Stream Processor			Software	von Newmann machine
Acronyms: r=reconfigurable | FP=Field-Programmable | GA=Gate array DPU=DataPath Unit | DPA=DPU Array				

Figure 3.4: Morphware Terminology - Nomenclature for configurable computing, taken from Morphware.net.

The main approach for reconfigurable DataPath Unit Arrays (rDPAs) is to use meshes, arrays or crossbar interconnects between processing units in the architecture [41]. By using multiple nearest-neighbor interconnects between PEs, a high level of communication is achieved for efficient parallelism. Some of these architectures even present multi-granular options, such as Matrix [48], RAW [49], Chess [47], and Pleiades [50]. Others like RaPiD [51] and PipeRench [52] present a linear approach, increasing pipeline efficiency. To exploit data parallelism, these architectures gain performance by adding more processing cores, but this approach degrades rapidly due to communication overhead. Due to limited memory bandwidth, the performance of these machines will level off or even decline with more cores [53].

Whereas multiple reconfigurable DataPath Units (rDPUs) and rDPU Arrays (rDPAs) are being or have been developed, especially in academia, none are commercially successful. Those architectures which have seen commercial applications, like Cray’s XD1 [54], are in essence fine-grained hardware accelerators in an FPGA fabric.

rDPUs and rDPAs propose a platform implementation to fit the needs of high granularity datapath reconfiguration. Their purpose is to create a coarse-grained architecture which facilitates low overhead run-time reconfigurability and favors data processing over bit manipulation. The main production bottleneck for these architectures is indeed their end goal: implementability. To make progress in the construction process, these rDPUs and rDPAs are frequently implemented in fine-grained, commercially available FPGAs [55]. FPGAs are essentially fine-grained array structures, which can implement and map all of the other architectures, at the cost of high placement and routing costs. Hence, even though the offline synthesis, placement and routing costs for FPGAs might be elevated, FPGAs are still cheap, flexible and available. These were essentially the reasons we chose FPGAs as our reconfigurable architecture; section 4.2.1 provides more details on our rationale for a specific FPGA device.

3.4.1.3 Reconfigurability

The reconfigurability of a platform can be divided into architectural, micro-architectural, and implementation-level configurability [42]. Our project, Hephaestus, uses architectural reconfigurability, in which the processor architecture can be configured in a variety of ways. Processors with a reconfiguration fabric for custom instructions (including all soft-core processors) have micro-architecture reconfigurability. Implementation-level reconfiguration platforms include multiple resource-efficient system implementations, with power consumption minimization being a common design factor [46, 56]. FPGAs support all of the above types by providing bit-level fine-grained reconfiguration.

3.4.2 Soft-core processors

The second type of reconfigurable computing are soft-core processors. We established that the first type, reconfigurable architectures, are a poor match for general-purpose programming and common applications. These architectures coarse granularity and reconfigurability prevent fine-tuning, and their structures are immutable. These rigid structures benefit from regular datapaths, but they also limit the customization of hardware for performance gains in non-structured applications. The alternative, microprocessors, provide a more general-purpose solution. However, being hard-core, they are not configurable.

Soft-core processors solve this by providing adaptability to an otherwise static design. They are implemented in high abstraction languages, or distributed as synthesized netlists by FPGA vendors. Because their behaviors and structures can be manipulated by a designer, this makes them particularly appropriate for reconfigurable computing.

Predictably, the last few years have witnessed a growing trend to use embedded soft-core microprocessors in FPGA designs [57] as shown in Figure 3.5.

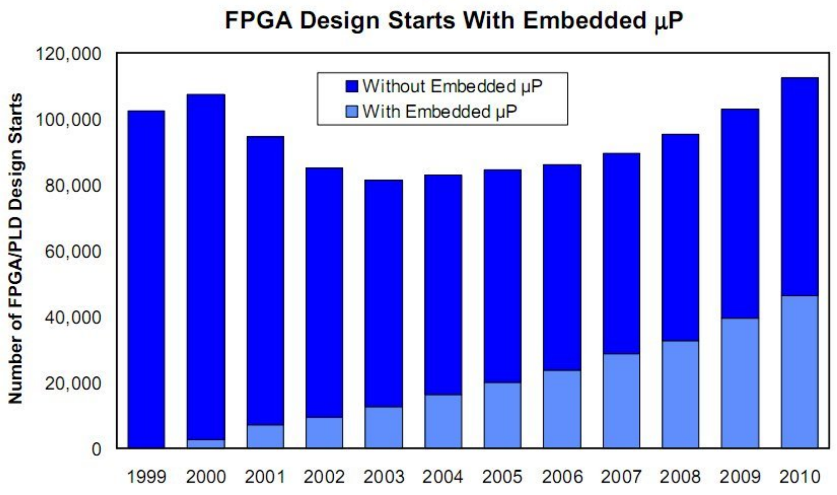


Figure 3.5: Microprocessor Use in FPGAs - FPGA Designs with embedded microprocessors [58].

Soft-core processors have many advantages over reconfigurable architectures. First, optimizations can be achieved by implementing specific instructions into the processor. Common choices for optimization are frequently-used and high-cost instructions or sets of instructions. This is remarkably similar to having a reconfigurable co-processor. Garp [3], Chimaera [12], OneChip [59], and the Cray XD1 [54] all use a fixed processor core with a co-processing reconfigurable unit. However, pure soft-core processors are also implemented in reconfigurable fabric. Hence, the optimization is part of the processor itself, without the need of costly communication with a co-processor.

Second, because customization is done within the soft-core processor, communication costs between the general purpose and custom modules are minimized. This in turn can reduce delays between synchronous elements of the processor, increasing the clock frequency and raw speed.

Last, the placement of the processor is dynamic. This allows for some mobility inside the reconfiguration fabric, which can place the processor closer to input and output pins or relevant surrounding components. As will be seen in the Evaluation Methodology in Chapter 5, placement and area constraints are of particular importance for partial reconfiguration. This method is used to time-multiplex the resources for FPGA functionality, allowing logic blocks to be replaced, while others continue to execute their tasks.

FPGA vendors provide soft-core processors for their platforms, but other soft-core processors are built from the ground up. These latter ones are necessary when existing soft-core processors do not provide enough flexibility.

3.5 Current State of soft-core processors

To make soft-core processors competitive, vendors optimize them for each of their particular platforms. This guarantees an efficient use of the platform's resources for the users, at the expense of providing only limited control over the processor's structure. While this is an acceptable situation for pure general-purpose computing, the fact that designers implement their systems in reconfigurable logic in the first place suggests a desire for customization for an optimal design. These designers working in reconfigurable computing need general-purpose soft-core processor, yet they also need them to be more flexible and customizable.

However, current soft-core processors are effectively vendor and technology dependent, limiting further optimization by users and making these processors not portable. In addition, since the processors are implemented using proprietary tool chains, their high-level hardware description language definition are not available for customization, except for a purchase price of several thousand dollars.

The amount of customization for current soft-core processors is defined mostly by a small number of features and optional inclusion of multiple system peripherals. Further customization can be achieved by implementing a co-processing unit, instead of modifying the main soft-core processor setup. Nevertheless, co-processing units for custom instructions tend to degrade the performance due to communication costs. Hence, it is preferable to customize the main processor itself, and to avoid co-processors (and their associated costs).

Current soft-core processors only allow small amounts of customization. A list of the most popular ones is presented below.

- Xilinx's MicroBlaze$^{\text{TM}}$: 32-bit DLX, RISC architecture soft microprocessor. Includes performance and size optimizations. Supports clock frequencies as high 200 MHz.

- Altera's Nios II$^{\text{TM}}$: 32-bit instruction set based on RISC architecture. Includes fast, economy and standard implementations. Supports clock frequencies as high as 200 MHz.
- Lattice's LatticeMico32$^{\text{TM}}$: 32-bit Harvard, RISC architecture soft microprocessor.

As is evident with Xilinx's MicroBlaze and Altera's Nios II implementations, soft-core processors can be optimized for area or speed, depending on the inclusion of different system resources. A compromise between area, versatility and clock frequency should be selected by the system designer, but these processors will only allow a small amount of datapath customization. Thus, we provide a soft-core processor that not only allows datapath customization, but also lets a designer fine-tune its architecture to meet computational needs.

We now take a closer look at the soft-core processors mentioned above.

3.5.1 Xilinx - MicroBlaze

The MicroBlaze processor has the following parameters relevant to data processing [60]:

- Barrel shifter: can enable speedup of shift operations
- Embedded divider: speedup integer division operations
- Embedded multiplier: speedup integer multiplications
- Floating Point Unit (FPU): For arithmetic and comparison operations with floating point numbers

MicroBlaze uses a scalar pipelined DLX architecture [15]. This means that the pipeline is stalled for high-latency operations, even if there were other independent instructions which could be executing at the time. Six integer parallel instructions could be executed while waiting for a floating point multiplication result, or thirty during a floating point division. Because no instructions can be executed while another one is already in execution, there is also no point in pipelining the floating point unit (FPU). A pipelined FPU can handle multiple independent instructions in each of its pipeline stages. Xilinx also states that the size of their FPU can not be reduced [61], even if only a subset of the floating point operations are needed for a design.

3.5.2 Altera - Nios II

Another soft-processor that allows for customization is Altera's Nios II. This soft-core processor comes with default compromises between speed and area: fast, economy and standard implementations [23]. The economy version does not allow additional features due to its small size, but the fast and standard implementations do. These features are:

- Hardware multiply: Can use DSP blocks, embedded multiplier, Logic Elements (LE), or none (software implementation)
- Hardware divide: Implemented in LEs
- Custom Instructions: Implemented as a co-processing unit to the Nios II processor.
- Floating Point Unit (FPU): For arithmetic and comparison operations with floating point numbers
- Floating point divider, which consumes more resources, is optional.

Custom instructions are implemented in the processor's datapath [62]. This effectively stalls the processor from executing additional operations while multi-cycle instructions are being resolved. Nios does allow for its most area-expensive operation, floating point divide, to be an optional addition to the design. A Hardware description language implementation of the Nios processor has been made at the University of Toronto [16] for added customization, but it does not address issues of performance or parallelism.

3.5.3 Lattice - LatticeMico32

LatticeMico32 is a 32-bit soft processor core for Lattice FPGA devices [63]. Most of the customization is done by selecting the size of data and instruction caches [64]. Customization is also done by selecting:

- Hardware multiply: Can use DSP blocks for pipelined multiplier, or multi-cycle implementation in Look Up Tables (LUTs)
- Hardware divide: Implemented in LUTs
- Floating Point Unit (FPU): Not implemented

- Barrel shifter: Pipelined to speedup shift operations

The Mico32 processor has a six-stage scalar pipeline to enhance efficiency, but performance improvement is not achieved through parallelism.

3.5.4 Other soft-core processors

Other non-commercial soft-core processors have been developed to deal with shortcoming of the above commercial ones. A good example is Leon3, a synthesizable VHDL model of a 32-bit processor. This processor can implement up to 16 CPU cores for asymmetric or synchronous multiprocessing [65]. Leon3 is a good platform to exploit thread-level parallelism (TLP), yet it does not effectively use available hardware resources to exploit instruction-level parallelism (ILP).

In a sense, Hephaestus is an extension of most soft-core processors. Multiple (different) Hephaestus cores can be instantiated in a system to exploit TLP, but in addition, each superscalar Hephaestus processor explicitly defines efficient configurations to exploit ILP in each computational thread. This satisfies the performance desires of specific applications.

3.6 Instruction-level parallelism

Instruction Level Parallelism is important since it is one of the main sources of application parallelism. Mechanisms for exploiting ILP can be implemented fully in hardware, abstracting the underlying processor configuration to programmers. In addition, ILP can be implemented without communication and synchronization overheads like those of TLP mechanisms.

Evidence of the shifting paradigm from linear to parallel computations emerge when analyzing the maximum amount of ILP different applications have had over the past decade. In 1991, David W. Wall researched the limits of instruction-level parallelism, suggesting that the maximum parallelism available for benchmarks was 3.2 [66] and limited to a maximum of 5 [67]. But a decade later, conventional compilers showed reasonable success extracting ILP in the range of 4-10 for VLIW and superscalar architectures [68].

Instruction-level parallelism has not been efficiently used in the past decade. One of the main reasons for this abandonment is that efforts have been rightfully concentrated on Data-Level Parallelism (DLP) for vector-style processing, and Thread-Level Parallelism (TLP) for multi-processor systems, a more common approach. DLP and TLP techniques aim to exploit parallelism at a coarse-grained level, but neglect the fact that ILP is not necessarily mutually-exclusive from them.

A problem with TLP is the need for a coherent global memory state with multiple processors modifying it in parallel. This usually results in a serialized access to global memory, becoming a system bottleneck. In addition, high performance requires access to multiple memory ports, which creates coherence issues at the memory level, making memory chips more expensive. To circumvent this issue, local memories are added to each processor core, but the bandwidth to the global memory is still limited by serial access.

Multi soft-core processors share the same issues of multi-core systems. Soft-core processors have access to a local memory, from which they get a single stream of instructions, working on a single set of data. Multiple processors working on completely independent sets of instructions and data are defined as embarrassingly parallel systems. These systems are only ideal for distributed computing and graphic processing. However, typical numerical computations will have instruction, data and control dependencies among multiple processors, which require a communication setup among them.

Soft-core processor communication can be standardized over a bus, or can be more a complex approach, similar to the communication structures of reconfigurable architectures in Figure 3.3. A communication bus is the more common of the two, with tool chains from FPGA vendors setting it as default. Once inter-processor communication links are in place, multiple soft-core processors can be instantiated and placed to potentially increase performance, if the inter-processor communication costs does not actually degrade it.

Freitas et al showed the performance and area evaluation of multiple MicroBlaze processors connected over a peripheral bus to shared memory [22]. The communication latencies between processors were 4-8 cycles of arbitration and 3-4 cycles for bus read and write requests. They also implemented a second design based on a 2D-mesh topology, with a lower communication latency of 3 cycles for routing and 4 cycles for bus reads and writes.

Multi soft-core processor systems are also attainable with Altera Nios II [69]. Communication is implemented by shared memory through the use of a mutex. This mutex provides atomic operations on shared memory, but it is a mechanism that relies on software arbitration to ensure the integrity of the data in memory. This arbitration is not free, and comes at a cost of synchronization time.

The need to exploit TLP is evident, and necessary. Even though the multiprocessor systems above boast performance gains, they do so at increased communication, synchronization and arbitration costs. Minimizing the amount of processors decreases communication costs, but increases the workload on the computational threads running on each processor. Each increased workload will include additional control, data and instruction dependencies, which current soft-core processors can not run efficiently at the instruction level.

A soft-core processor was needed, which could handle ILP. Additionally, it needed to be flexible enough to be instantiated multiple times in an FPGA to exploit TLP, while easily implementing custom logic. Hence, implementing a superscalar soft-core processor in hardware was appropriate for performance and flexibility.

3.7 Hephaestus' approach

Hephaestus provides a superscalar solution to the above issues. By exploiting ILP, it gives high-performance to the soft-core processors in charge of computational threads, which presumably, are also exploiting TLP. The amount of ILP to exploit is configured at synthesis time; evaluation of the amount of resources used to support such system is given in Chapter 5.

We are not the first to use soft-processors for executing multiple instructions in parallel. OneChip provides a superscalar system in which a reconfigurable co-processor can execute instructions in parallel with the main hard processor [70]. This provides an adequate enhancement to performance. Their system claims to be superscalar not because it can fetch multiple instructions at once, but because it can issue them to the hard processor and soft co-processor in parallel: an augmentation of a hardware accelerator. The throughput of the system is still delimited by a scalar processing limit of one instruction per cycle.

Additionally, a fully soft-core reconfigurable FPGA processor with multiple functional units was created at the Swiss Federal Institute of Technology [71]. This design provides a bit more flexibility than OneChip, by providing customizable functional units for use by the system. Unfortunately, the functional units are dynamically reconfigured into the system each time a custom instruction enters the pipeline. This means that 1) the system performance pays the price of reconfiguration each time a switch in instruction types happen and 2) only one functional unit can be active at any given time.

Our processor is a fully-implemented superscalar processor. It fetches multiple instructions at once and uses register renaming to solve data, output and anti-dependencies in the multi-instruction stream. As soon as instructions are ready they enter the execution stage, in parallel, and their results are broadcast to waiting dependent instructions. Finally, multiple instructions can complete and be written to memory at once.

Hephaestus' customization depends not on optional peripherals, but on specifying parameters for the underlying structure of the superscalar processor. Our vendor-independent approach allows this configuration, on top of portability. However, implementing a flexible and reconfigurable soft-core processor design with these desired characteristics uses an increased amount of resources.

However, FPGA's resources, capacity and speed continue to increase at an accelerated pace, following Moore's Law, while their prices steadily decrease. Figure 3.6 shows these different trends that make FPGAs an enticing platform. With FPGAs evolving so rapidly, it is difficult to assess the amount of computation that we can fit in a specific area within the FPGA.

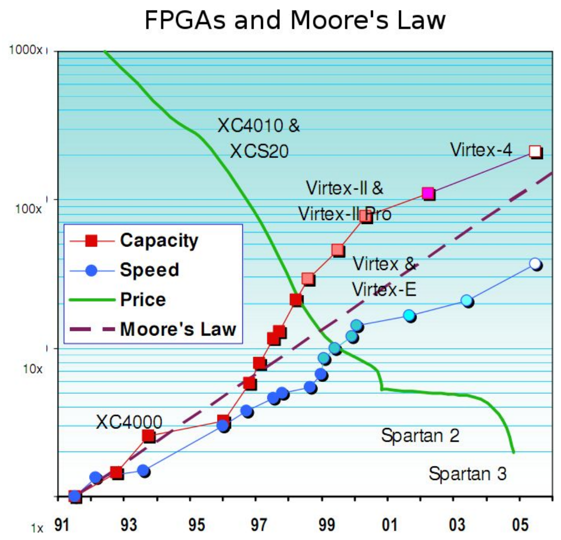


Figure 3.6: FPGAs and Moore's Law - This graph from Xilinx shows the trends on capacity, speed and price for FPGAs with respect to Moore's Law [72].

An computational density analysis of performance against area is then needed to evaluate the efficiency of our processor. Wentzlaff et al provided a insightful performance/area analysis of bit-level computations for ASICs, FPGAs and microprocessors [73]. Their rules of thumb are:

- ASICs provide a 2–3 times absolute performance improvement over an FPGA implementation.
- FPGAs provide a 2–3 times absolute performance improvement over a microprocessor implementation.
- ASICs provide 5–6 orders of magnitude better performance per area than software implementation on a microprocessor.
- FPGAs provide 2–3 orders of magnitude better performance per area than software implementation on a microprocessor.
- Parallel implementations on Tiled architectures yield competitive absolute performance to that of FPGAs but use at least an order of magnitude more area to do so.

We are careful to note that for bit-level, fine-grained computations, the playing-field is biased towards FPGAs. Regardless, Wentzlaff et al show that ASICs provide the fastest, smallest and most inflexible solution, followed by FPGAs with similar speed, greater flexibility, but larger area consumption, and reconfigurable architectures with similar performance, reduced flexibility and extra added area use. Their analysis is a good foreshadowing of findings in our project. Hephaestus, working at word-level computations, also provided some insights into the performance/area tradeoff of soft-core microprocessors implemented in FPGAs.

This tradeoff was also refereed by constraints placed on the processor's area. Hence, the shape and location of the processor were explicitly determined, to allow for slotted synthesis. Useful synthesis techniques for processor area and location have been proposed by Natale et al [74]. In them, they use a 2D packing algorithm to efficiently place Altera's Nios II processors to maximize performance over area. However, currently only Xilinx devices support partial reconfiguration. While we could use an efficient placement of processors to yield performance density gains, we are limited to slotted synthesis to support partial reconfiguration.

This conformance to partial reconfiguration placement will potentially make the processor more useful, as partial and dynamic reconfiguration techniques and their necessary tool-chain support become more standardized. For example, a Hephaestus system could be implemented to allow dynamically reconfigurable placement of superscalar processors to enhance multi-threaded processing.

3.8 Summary

Application-specific circuits can be mapped onto reconfigurable logic, but reconfigurable architectures do not provide enough flexibility of design as FPGAs. At the cost of complex routing, the latter provide fine-grained reconfiguration, which can also support coarse-grained solutions. This flexibility makes FPGAs particularly appropriate to implement general-purpose soft-core processors, while taking advantage of upcoming next-generation reconfigurable platforms through high-level hardware description languages.

Current soft-core processors lack flexibility, as they are mostly provided by FPGA vendors without access to their lower level architectural details. The processors' main characteristic is to be generic enough to handle different computational tasks through thread-level parallelism. But the costs of TLP become prohibitive with the increased number of processors, and a reduced number of them either increases the processor's complexity or decreases their performance.

To avoid these performance penalties, we increased the complexity of soft-core processors by allowing them to exploit instruction-level parallelism. Hephaestus' superscalar processors consume an increased area and resources, but are able to increase local performance by exploiting ILP on each processor. Additionally, each processor can still be synthesized into slotted areas in the FPGA to support partial reconfiguration techniques. Thus, we fill the need in the reconfigurable computing area for a general-purpose, customizable, and high performance soft-core processor. Should multiple of these processors be needed for TLP, they can also conform to dynamic reconfiguration, making run-time custom processor swapping a possibility.

Chapter 4

Implementation Methodology

This chapter rationalizes Hephaestus' choice of the development platform and programming language, outlines the implementation methodology and describes the data acquisition techniques.

4.1 Structure and configuration variables

Most simple soft-core processors have scalar throughput, meaning that they can complete a maximum of one instruction each clock cycle. While the processor may be configured to execute multiple instructions at once, a performance bottleneck occurs when forwarding each instruction's results to other dependent instructions. In addition, in-order execution is enforced, preventing parallel execution of independent sets of instructions. The main purpose of using parallel Execution Units (EUs) in simple processors is to reduce the penalty of high latency instructions.

In our approach, we developed a full superscalar processor with multi-instruction issue, various EUs, and broadcasting of results. The use of multiple EUs allows not only the reduction of heterogeneous instruction-latency execution penalties, but also multiple instructions to complete concurrently. This effectively avoids the Instructions Per Cycle (IPC) upper limit of 1 for scalar processors.

The main configuration variables define the component, logic and interconnect generation rules of our system. Flexibility and ease-of-configuration were prime concerns for the processor structure. Most of the system can be modified by simply changing the numerical value of a parameter and re-synthesizing, without any further changes. Figure 4.1 shows a simplified architectural view of a Hephaestus processor with the main components controlled by system parameters. The architecture retrieves instructions, and looks for the instructions' operands in the register file. If they are not found, instructions must wait in the reservation stations until their operands become ready, at which point they proceed to the execution units for processing. Their results are broadcast back to any dependent instructions waiting in the reservation stations. The final stage sort the results in program order to recover from interruptions in the instruction flow.

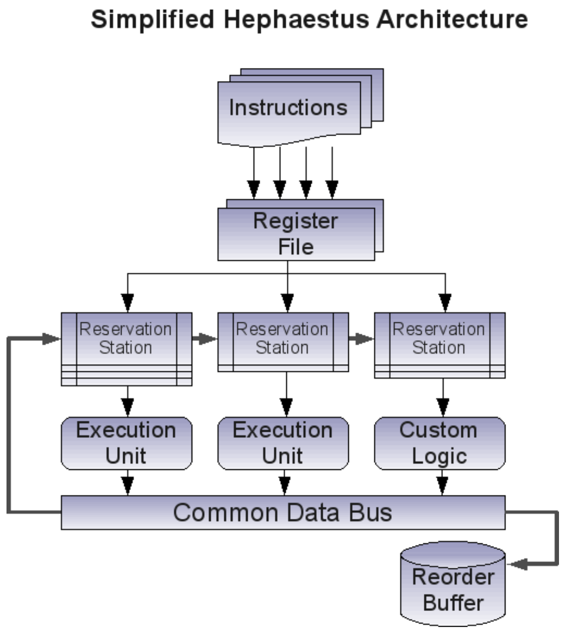


Figure 4.1: Simplified Hephaestus Structure - The main structure components altered by a Hephaestus processor's system parameters.

These main configuration variables for a superscalar processor were identified as:

- `Instruction_Issue:` Indicates the total number of instructions issued from instruction memory at once. Also configures the instruction memory's organization and read/write ports.

- `Number_of_Execution_Units:` Sets the total number of general-purpose and custom logic EUs in the system. Does not necessarily indicate types of EUs, except when all EUs are different. EUs are manually added to allow design customization.

- `Common_Data_Bus_width`: The maximum number of broadcasted instruction results each clock cycle. The results of finished instructions are broadcast out-of order to the reservation stations, but in-order completion of instructions is left for the reorder buffer.

- `Register_File_size`: Indicates how many general-purpose registers the register file has. It includes addressable ISA registers and architected registers available for register renaming.

- `Register_File_memory_organization`: Organizes the register file to have multiple read and write ports at the cost of redundancy.

- `Reservation_Station_organization`: Sets up the maximum number of instructions waiting to execute in each EU. Each individual EU's reservation station can be configured to have a different number of waiting stations.

- `Reorder_Buffer_size`: Deterministic variable that sets the size of the reorder buffer, which ensures in-order completion of instructions for precise interrupts.

Other variables are mostly dependent on the type of customization a developer might use, but the ones above depict a clear configuration for a superscalar processing system. With more control over the underlying processor architecture, a more suitable system can be created for a particular application, even in systems with thread-level parallelism.

4.1.1 Rationale

To deal with Thread-Level Parallelism (TLP), designers tend to add more processors to their systems, increasing communication and synchronization costs between threads. These simple processors can handle reduced independent work loads. However, when dependent instructions come into play, especially across threads, inefficient synchronization policies and high latency mechanisms take a heavy toll on system performance and throughput.

Capitalizing on TLP generally yields positive results, but we can also take advantage of its non-mutually exclusive relationship with ILP. Parallelism at the instruction level within a thread can prevent the use of extra threads, reducing the communication and synchronization overhead. Thus, the rationale behind using superscalar processors is to benefit from both TLP and ILP. By examining their performance and area relationship, one can quickly assess potential benefits and detriments of replacement of scalar processors by superscalar ones.

4.1.2 Advantages

The performance/area assessment for a soft-core superscalar processor, provided in section 5.3, lets designers evaluate (before implementation) whether potential performance gains of ILP would be too costly in terms of area or clock frequency degradation. In the case of potential benefits, the configuration of our system matches the application's degree of parallelism.

This refined configuration allows the system to maximize architectural parallelism without crossing the point of diminishing performance returns. As a result, the processor's complexity is kept at a minimum, thus enhancing clock frequency and reducing the required area. Applications with high-ILP are configured in complex parallel superscalar architecture processors, offering concurrent execution of instructions. In contrast, low-ILP applications also benefit from processor configurations with high speed, low complexity and reduced area. When there is no potential ILP, the processor's configuration allows a single instruction issue, and consequently Hephaestus processors gracefully degrade into pipelined RISC processors.

The independence of execution units provides great benefits to the system. EU independence optionally allows a detachment from typical ALU-style combinatorial circuits for execution units. Mechanisms for EUs may also include finite state machines (FSM), variable multi-cycle execution, and use of the platforms dedicated resources like local memory. For example, the default load/store execution unit is set up to access fast, small local memory directly, but it can also be set up for variable high-latency large RAM memories (though FSMs). Unless extra inputs or outputs are needed, the mechanism of each EU is hidden behind a default interface and protocol policy.

4.1.3 Disadvantages

A careful balance must be kept between functionality and performance degradation. The more functionality we put into execution units or processor configurations, the higher the chance of detrimental costs. As such, the main expected disadvantages of our approach were area and resource use. Because the processor fetches instructions in-order, executes them out-of-order, and then reorders them for completion, much information has to be replicated in each stage, especially during out-of-order execution. During this execution stage, instructions wait for their operands to become available, so we have to provide additional circuitry for each waiting instruction. These are the reservation stations. Consequently, as these instructions execute out-of-order, we need larger storage until they are reshuffled in chronological order by the reorder buffer.

Because we have an 11-stage pipeline to maximize clock frequency, a control or data hazard in any stage effectively stalls all previous stages. Hazards can only be detected after they happen, and they disrupt the natural flow of instructions through the pipeline. When this occurs, all previous stages in the pipeline are stalled. This means that each stage must keep their output signals constant until the hazard is solved and the pipeline flows again. The output signals have an added storage cost per pipeline stage, implemented as registered outputs of combinatorial stages.

These storage requirements use more resources and area, but fortunately FPGAs continue to provide more resources with each generation evolving into faster, larger and more dense devices. Figure 4.2 shows the trend for increasing density and resources for FPGAs from major vendors, indicating how area is not necessarily a high price to pay.

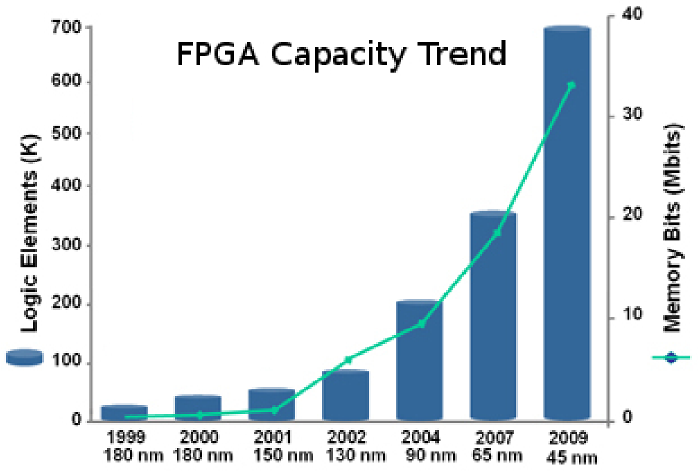


Figure 4.2: FPGA logic element and memory capacity - Density and resource increase with each generation of FPGA devices [75].

Having identified the overall system configuration parameters, which provide an appropriate amount of architectural parallelism for specific applications, we now specify the means by which the Hephaestus project is implemented.

4.2 Instrumentation

A soft-core processor can be implemented using a variety of methods. In addition, multiple tools exist to define, simulate, and implement logic circuits into multiple reconfigurable devices. Each selection has a potential impact on our processor's resulting performance. In addition, metrics of area use change depending on the devices chosen for implementation. Careful consideration was taken when selecting the project's platform, programming language, simulator, and synthesis tools.

4.2.1 Platform Selection

Our previous discussion of reconfigurable platforms identified FPGAs as prime candidates for our project. FPGAs are commercially available in various speeds and sizes by multiple vendors. They are fine-grained array structures, which allow bit manipulation but also implement coarse-grained datapaths at the expense of place and route costs.

The main vendors for FPGAs are Altera and Xilinx, each with a variety of devices under their belt. Since logic can be synthesized into each vendor's FPGA devices, we had to pay attention to some of the more subtle differences between them. Some of these details could make or break the underlying assumptions that we were using to build our project. The two assumptions that we found were important system characteristics were present in only one FPGA vendor, Xilinx.

First, Xilinx FPGAs contain both small (RAM based on configurable logic blocks) and large (Block RAM) quantities of embedded RAM memory, while Altera only has large ones. Distributed RAM based on Configurable Logic Blocks (CLB) is used to implement the processor's register file, which is only a few hundred bits in size, yet heavily accessed. Routing congestion is eased by not using Block Ram memory, hence keeping the CLB-based RAM close to the points of origin and destination for address and data.

The second and main difference is the ability to partially reconfigure the device. Altera does not support this feature. Xilinx allows this through the configuration of column-wise slots in their Virtex FPGAs. It is important to note that while this project does not use partial-reconfiguration, one of our objectives is to support the area shape requirements to enable it in the future. Both vendors provide area constraints on synthesis, yet only Xilinx allows partial reconfiguration.

We chose a Xilinx Virtex-II Pro board as our implementation platform.

4.2.2 Programming language

FPGAs allow a variety of languages to describe the logic which will be mapped onto them. We used VHDL as our high-level hardware description language because it is mature and very well supported. In fact, synthesis tools can automatically infer components like finite state machines and RAM from well-structured VHDL code. This frees us from using pragmas to use device-specific components, and consequently making our code vendor-independent and portable.

4.2.3 Simulation tools

Most of the processor development required careful design and thorough testing. Synthesis times were considerably lengthy at times (about an hour per run), which made simulation the best way to quickly evaluate correct behavior until the end design was done. Most of the time spent creating the superscalar processor was used for debugging the data, control and hazard signals through the simulation.

We used ModelSim SE 6.0a to create the test benches that tested incomplete versions of the processor's individual components. Once all of the components were completed and shown to be true and interconnected, there was no further need for test benches. The simulator would set up the processor's instructions and data memories from the same files that would load its initial values during synthesis.

ModelSim [76] was then mostly used to monitor signals internal to the processor rather than its inputs and outputs. Because of the lack of a compiler for our processor, we used a Python script to correctly format assembly code into the binary instructions and data. This script allowed great flexibility, especially since the binary encoding of instructions changed with different sets of execution units. These were also stored in different locations according to the instruction issue width.

Additionally, the script also sets up initial data and register memory values.

4.2.4 Synthesis tools

Each FPGA family is tightly coupled with corresponding synthesis tools. For Xilinx devices, the main tools are Xilinx Platform Studio Embedded Development Kit (EDK) and Xilinx Integrated Software Environment (ISE). EDK includes a suite of tools to easily drop IP cores, and create software to run on MicroBlaze designs. ISE instead is focused on hardware description language (HDL) synthesis and implementation.

We used Xilinx ISE 8.2.2 to synthesize, implement and download our project to the FPGA board. Additionally, we synthesized our project using Altera Quartus II. This last part was to remove any vendor-dependent HDL code and to compare and contrast area use for our processors.

4.3 Model implementation

The processor model was first created in VHDL, simulated, and then synthesized. The first iterations of the processor made use of VHDL generics, to pass appropriate parameters to each individual component at synthesis-time. When the list of generics and number of components became too big to handle, we switched to a VHDL package of constants, which were propagated into all components at once. A sample of the VHDL package code is shown in Code 4.1.

Code 4.1: VHDL package definition

```
PACKAGE Hephaestus_Configuration IS
    -- ************************************
    -- ***** USER SELECTED PARAMETERS *****
    -- ************************************
    -- Main Superscalar Parameters
    CONSTANT Instruction_Issue   : INTEGER   := 04;
    CONSTANT Number_of_EUs       : INTEGER   := 03;
    CONSTANT Common_Data_Bus     : INTEGER   := 02;
    -- Instruction Parameters
    CONSTANT Instruction_size    : INTEGER   := 32;
    CONSTANT Opcode_Regs         : INTEGER   := 02;
    CONSTANT Opcode_Function     : INTEGER   := 04;
    CONSTANT Reg_Address_size    : INTEGER   := 04;
    CONSTANT PC_Address_size     : INTEGER   := 08;
```

Once this became the central repository for system parameters, main work started on the design of the processor. We used a mix of classic superscalar architecture literature [77] and some more modern approaches [78]. Once the main design was in place, all components were created, interconnected, and the design hierarchy was analyzed to simplify control signals throughout the pipeline.

4.3.1 Design Overview

The design of a processor generally follows a chronological trip down its pipeline. First the instructions are fetched and then decoded to extract the sources to be used. With the retrieval of the sources, execution can begin. The results are then saved back to the register file, and finally memory is accessed for load/store instructions. This method follows the DLX RISC processor pipeline proposed by Hennessy and Patterson [79].

Because our superscalar processor allows instructions to execute out-of-order, several more components form our processor structure. To minimize the effect of the critical path of performance, we separated each expensive operation into one or multiple additional pipelines stages, ending up with a total of 11 stages.

It is useful here to expand the simplified diagram in Figure 4.1 to include all of the components for the superscalar processor. This figure only showed those components directly affected by system parameters, while in reality all components are indirectly affected by all system parameters. Figure 4.3 shows the main architecture of a Hephaestus processor, and the 11 stages composing the processor's instruction pipeline.

This complex design follows the typical pipeline functionality of a 5-stage pipeline like DLX. The novel functionality stems from the synthesis of reconfigurable parameters into the pipeline stages. In the first stage, the Program Counter inputs the address into the Instruction Memory, and a number of instructions equal to the `Instruction_Issue` system parameter are returned in the second stage. These are checked in the next stage by the Jump Decoder to update the program counter with any jump control signals which do not depend on calculated values. The instructions' destinations and sources are register-renamed in the fourth stage to avoid data-, output- and anti-dependencies. With these dependencies handled, we are ready to execute them out-of-order. Because there are multiple Execution Units, as defined by the `Number_of_EUs` parameter, we need to demultiplex each instruction into its correct EU path in stage five. In the stage after that, each of the instructions' sources are examined in the Instruction Dispatch component, and contention for the register or future file ports is resolved. Results are retrieved from either the Future File or the Register File in the seventh stage. Usually the Future File will contain the most recently updated (out-of-order) value, while the Register File has the latest (in-order) completed value. The retrieved data values and the instructions then go into Reservation Stations at stage eight, until all of their operands are retrieved. This can happen immediately if all values were retrieved from the Future File, but usually an instruction must wait for other ones to complete and broadcast their results. When an instruction is ready, it goes into an Execution Unit for execution of its operands in stage nine. Once execution is done, it is then this instruction's turn to do the broadcasting through the Common Data Bus (CDB) located in stage ten. The CDB will broadcast a maximum number of results indicated by the `Common_Data_Bus` system

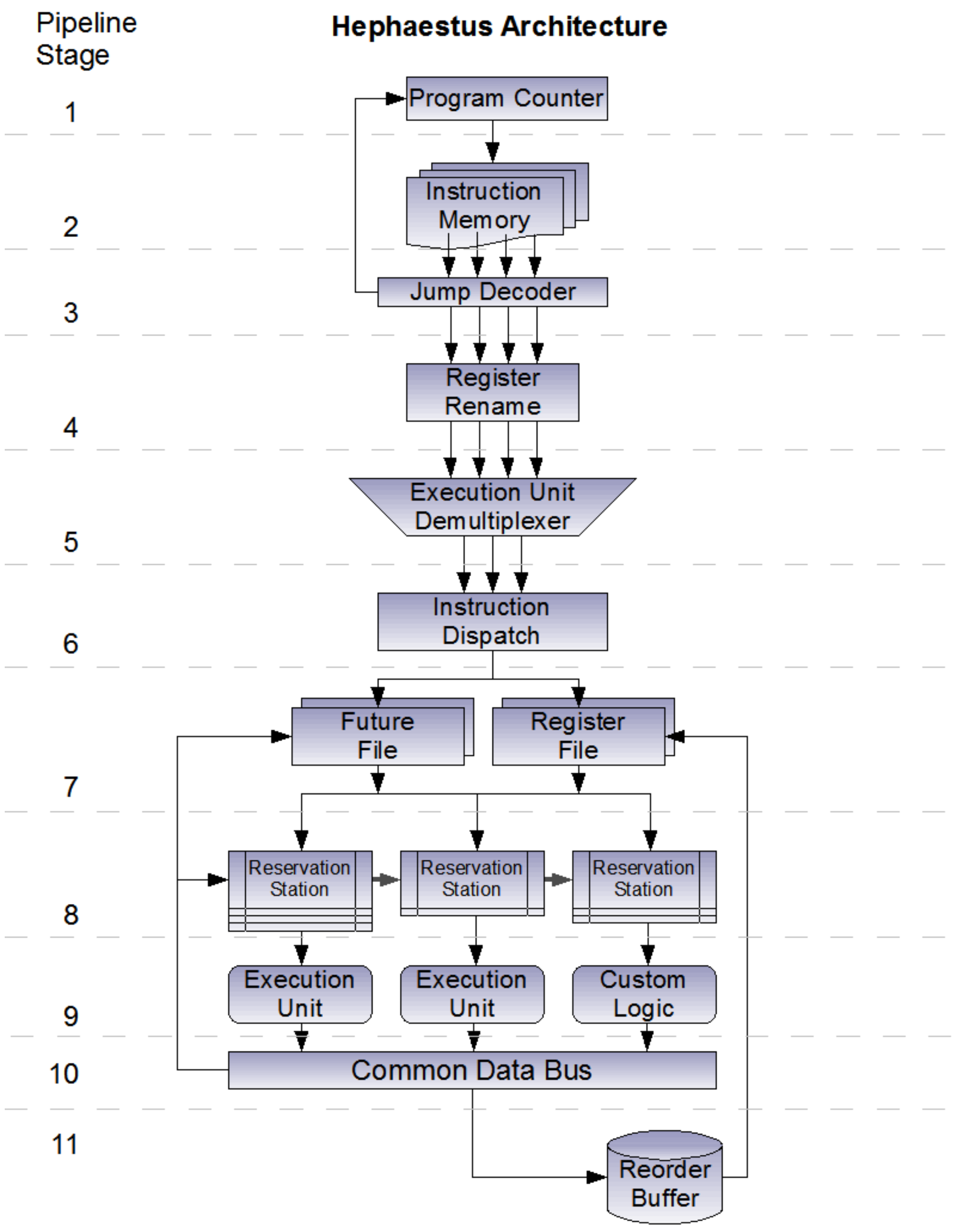


Figure 4.3: Hephaestus Superscalar Structure - All the components of the Hephaestus processor are shown here with their interconnections.

parameter. Finally, in the eleventh stage, the Reorder Buffer will store all out-of-order broadcasted results and sort them in program-order, writing the values in the Register File in the same order, allowing for precise interrupts.

4.3.2 Components

We now take a closer look at the functionality and methodology for creating each of the processor's components, listed below.

- Program Counter - Controls instruction address.
- Instruction Memory - Stores and output instructions.
- Jump Decoder - Decodes jump instructions and services them immediately.
- Register Rename - Renames destinations and sources to avoid false dependencies.
- Execution Unit Demultiplexer - Forwards each instruction to its corresponding EU.
- Instruction Dispatch - Solves contention for the register file's ports.
- Register File/Future File - Writes/reads in- and out-of-order data.
- Reservation Station Manager - Stores instructions until operands are ready.
- Execution Unit Manager - Starts EUs for ready instructions.
- Common Data Bus - Arbitrates broadcasting of results.
- Reorder Buffer - Sorts instructions in program order.
- Data Memory - Contains extra memory besides register file
- Pipeline Control - Provides centralized control for pipeline hazards.

4.3.2.1 Program Counter

The program counter (PC) feeds the address lines in the instruction memory with continuously updating values, which ensures new instructions are fetched each clock cycle. It also receives jump and branch instructions, which update the PC's value accordingly. The PC is also responsible for numbering outgoing instructions in ascending order.

Normally, a program counter refers to the location in instruction memory of the next instruction, but with superscalar processors, it refers to a set of instructions. When jumps and branches modify the PC, they pinpoint a particular instruction within the multiple instructions issued per cycle. This means that jump/branch locations do not have to be aligned with the PC address but can point to a particular instruction within the ones being retrieved from the PC address. When this situation is solved, the PC validates or invalidates instructions accordingly and passes them down the pipeline.

Figure 4.4 shows the update modes of the PC with a sample Hephaestus processor with an instruction issue of 4. In mode (a), the PC increases by one each cycle, making the next set of 4 instructions valid. In (b) a jump instruction sets the PC to instruction 28. Since we have an issue of 4 and 28 is divisible by 4, the jump destination is aligned with the jump address and all instructions are valid. Finally, situation (c) shows what happens when a jump to an instruction does not align with the address. Instruction 18 is located in address 4, with an offset of 2. The PC sets the new address, but invalidates previous instructions (16 and 17) that were not the target of the jump instruction.

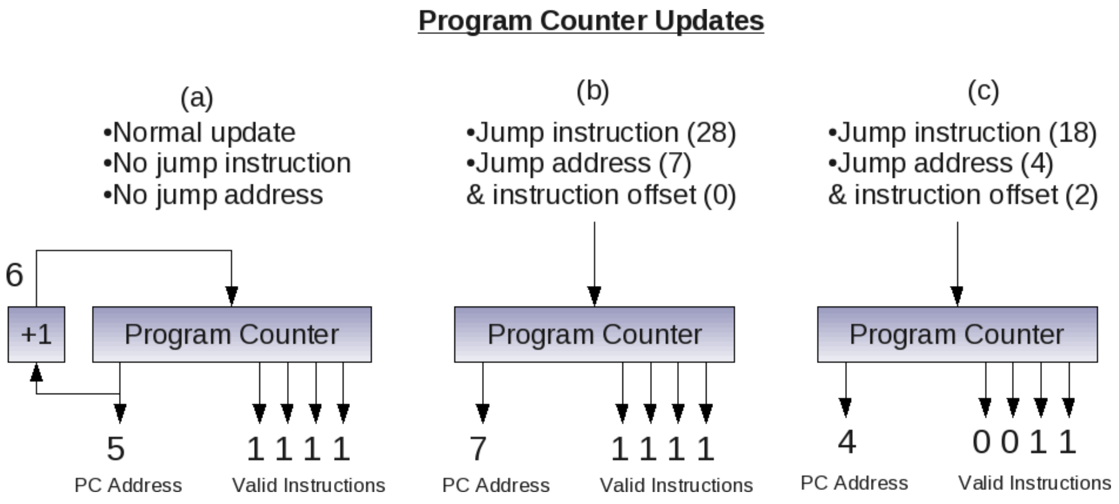


Figure 4.4: Program counter updates - Three different kinds of updates: a) Normal update b) Jump with instruction align c) Jump with no instruction align.

Branches are decoded into addresses and validation bits in a similar way to jump instructions. For different instruction issue widths, the number of instructions per PC address change accordingly.

4.3.2.2 Instruction Memory

The instruction memory stores the assembly-coded instructions that are fed to the rest of the processor. Like a normal memory, it has an address and read/write inputs, paired with incoming and outgoing data ports. Its size depends on the size of the assembly-coded software footprint, and its structure depends on the number of instructions issued at once.

We used code inference to implement specific platform resources (Block RAM) without instantiating these proprietary entities directly. This allows hardware abstractions across multiple devices, device families and vendor platforms [80]. The same hardware abstraction was used when creating the data memory, register file and future file components. Code 4.2 shows the hardware implementation report of this component in Block RAM resources.

Code 4.2: Instruction Memory in BlockRAM

```
Synthesizing Unit <RAM_Memory_0>.
    Related source file is "/users/ortizj/Hephaestus/RAM_Memory.vhd".
    Found 256x32-bit dual-port block RAM for signal <Bram_data>.
    -----------------------------------------------------------------------
    | ram_style          | Auto                                 |         |
    -----------------------------------------------------------------------
    | Port A                                                              |
    |     aspect ratio   | 256-word x 32-bit                    |         |
    |     mode           | read-first                           |         |
    |     clkA           | connected to signal <Clock>          | rise    |
    |     weA            | connected to signal <Write_Enable>   | high    |
    |     addrA          | connected to signal <Write_Add>      |         |
    |     diA            | connected to signal <Value_In>       |         |
    -----------------------------------------------------------------------
    | Port B                                                              |
    |     aspect ratio   | 256-word x 32-bit                    |         |
    |     clkB           | connected to signal <Clock>          | rise    |
    |     addrB          | connected to signal <Read_Add>       |         |
    |     doB            | connected to signal <Value_Out>      |         |
    -----------------------------------------------------------------------
```

The synthesis report in Code 4.3 also shows how the instruction memory is instantiated in multiple Block RAMs, each with a different initialization file. This avoids the problem of writing the instructions to memory after the processor is loaded into the reconfigurable platform.

Code 4.3: Instruction Memory synthesis report

```
Analyzing hierarchy for entity <RAM_Memory> in library <work>
(architecture <Component_instance>) with generics.
    Address_Size = 8
    InitFile = "/users/ortizj/Hephaestus/RAM/Instructions_0.data"
    Value_size = 32

Analyzing hierarchy for entity <RAM_Memory> in library <work>
(architecture <Component_instance>) with generics.
    Address_Size = 8
    InitFile = "/users/ortizj/Hephaestus/RAM/Instructions_1.data"
    Value_size = 32
```

We need multiple Block Rams because instructions need to be read from multiple read ports at every clock cycle. Because we have Block RAMs with fewer reads ports than we need, we interleaved Block RAMs together. This allows each Block RAM to hold the offset value corresponding to the instruction being issued at each particular address. Figure 4.5 shows the arrangement of 4 Block RAMs for a system with an `Instruction_Issue` of 4. The address port A `addrA` in each Block RAM is set by the program counter, and after a single clock cycle, the instruction data is output in each `doA` data output port.

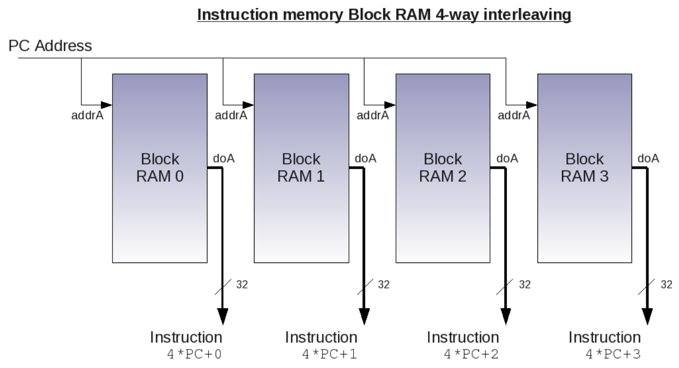


Figure 4.5: Instruction memory Block RAM configuration - All Block RAMs have the same address input from the Program counter, and output `Instruction_Issue` instructions.

Figure 4.6 indicates the location structure of Block RAM memories with 4-way interleaving. At each address, the Instruction Memory can read `Instruction_Issue` instructions at at the same time. The instruction number can be calculated with modulus arithmetic over the total number of instructions issued; 4 are issued in this example.

Instruction location in interleaved Block RAMs

Address	Instructions			
PC	Mem0	Mem1	Mem2	Mem3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23
...	...	...	...	...
1022	4088	4089	4090	4091
1023	4092	4093	4094	4095

Figure 4.6: Instruction memory locations - Instructions are stored with 4-way interleaving when the instruction issue width is 4.

4.3.2.3 Jump Decoder

The main function of the jump decoder component is to detect jump instructions and solve them immediately. Since jump instructions are independent of source registers, we solved them at this stage, without paying the penalty associated with traversing the whole depth of the pipeline. Branch instructions, which are dependent on source registers, do have to go through the whole pipeline to get resolved. This deviates from traditional approaches, where jump instructions are treated like branches, and traverse the whole pipeline before being executed [81].

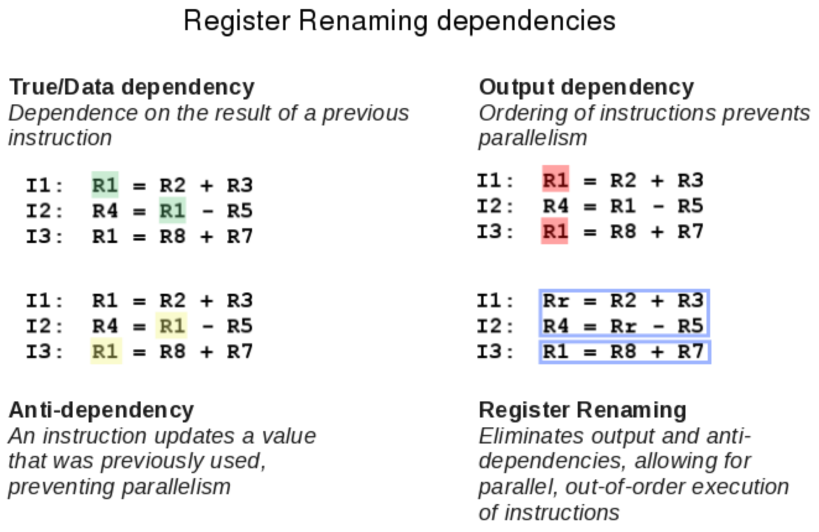
When a jump instruction is detected, the new PC address is either 1) calculated as an offset of the current address or 2) passed as an immediate value. We passed this value back to the program counter, which sets the appropriate validity flags and PC address as depicted earlier in Figure 4.4. While the new instructions are passed into the first 2 pipeline stages, our jump decoder waits, without outputting any new instructions. This continues until the PC indicates through the `Jumped_Address_In` input that it has serviced the pending jump request.

Additionally, the jump decoder tags any instructions following a branch as speculative. Because the target address for a branch can not be resolved until the instruction is executed, the rest of the instructions in the sequence are allowed to execute speculatively. In the case a branch instruction is taken, the Reorder Buffer completes all instructions in order until that branch, then sends the branch address request to the program counter and resets the rest of the pipeline and all intermediate instructions. If the branch instruction is not taken, then the speculative instructions are allowed to complete, without penalties associated with a pipeline flush. This pipeline flush degrades performance, and is a well-known necessary programming cost. Our system default is to assume a branch won't be taken, hence instructions can continue normal execution until they reach the Reorder Buffer at the end of the pipeline. The ReOrder Buffer prevents the writing of speculative results unless the branch was not taken.

4.3.2.4 Register Rename

The register renaming pipeline stage is the most important, as it eliminates output and anti-dependencies, while order-enforcing only the necessary true data dependencies. Register renaming allows for the parallel execution of instructions.

Figure 4.7 shows the three types of inter-instruction dependencies. A true data-dependency enforces program order: a previous instruction must be solved before a new one can start. An output-dependency is usually the result of register reuse and having a limited register file. Finally, anti-dependencies prevent newer instructions from writing to a register, until previous instructions have finished reading the target value. This is a classic example of overwriting a value which is still useful. In terms of reads and writes, a data-dependency involves a read-after-write (RAW) hazard, the output-dependency is a write-after-write (WAW) hazard and anti-dependencies involve write-after-read (WAR) hazards.



Register Renaming dependencies
True/Data dependency
Dependence on the result of a previous instruction
I1: R1 = R2 + R3
I2: R4 = R1 - R5
I3: R1 = R8 + R7
Output dependency
Ordering of instructions prevents parallelism
I1: R1 = R2 + R3
I2: R4 = R1 - R5
I3: R1 = R8 + R7
I1: R1 = R2 + R3
I2: R4 = R1 - R5
I3: R1 = R8 + R7
Anti-dependency
An instruction updates a value that was previously used, preventing parallelism
I1: Rr = R2 + R3
I2: R4 = Rr - R5
I3: R1 = R8 + R7
Register Renaming
Eliminates output and anti-dependencies, allowing for parallel, out-of-order execution of instructions

Figure 4.7: Register-renaming dependencies - Instructions can have true-, output- and anti-dependencies among them. Register renaming can solve two of them.

Register reuse is a common and practical practice, but creates false output dependencies. With register renaming, we make use of extra architected registers to rename the ISA registers and avoid these false dependencies. The example figure shows how two sets of instructions, I1-I2, and I3, can be executed in parallel or out-of-order after being register-renamed.

While register renaming is a technique which allows us to execute in parallel and out-of-order, it is a complicated process. The processor now needs to track which register contains the latest version of a particular value, which registers can be reclaimed for renaming, and finally, which registers have pending reads from them. To implement register renaming, we followed a modified version of the original Tomasulo's algorithm

[1]. This algorithm originally assigned reference tags and a busy bit to each register that did not contain the value needed by future instructions. Tagged values were recovered from broadcasting while untagged values were taken from the register file. However, we can not simply recreate a static FPGA-based Tomasulo's algorithm (an example of a stripped-down static version of an FPGA-based Tomasulo's algorithm was made in [82]) because our system involves added flexibility for the processor's underlying architecture. We implemented a modified version of the algorithm that was adaptable to our customization needs.

To modify Tomasulo's algorithm, we used the techniques outlined by Moudgill et. al in their alternative approach for register renaming and dynamic speculation [83]. We kept track of four different pieces of information about the registers : 1) If a value had been written to a register and hence the instruction is completed, 2) how many pending reads exist per register, 3) which registers are mapped into other renamed registers, and 4) the target value of each renamed register.

To accomplish the task of correctly reading, using and modifying these pieces of information, we declared four different signals in the VHDL description of the register renaming components, as shown in the Code 4.4.

Code 4.4: Register rename signal declaration

```
-- Register Renaming signals
SIGNAL Complete_Flag          : STD_LOGIC_VECTOR (0 to Physical_Registers -1);
SIGNAL Unmap_Flag             : STD_LOGIC_VECTOR (0 to Physical_Registers -1);
SIGNAL Renamed_Registers      : Register_Addresses_Type;
SIGNAL Reg_Counters           : RegCounter_Acccum_Type;
```

The `Complete_Flag` signal stores a bit value of 1 for each renamed register that has been written to. When the Common Data Bus broadcasts the results of the instruction, the renamed destination register's complete status bit is updated to 1. This informs the register renaming mechanisms that a register has completed execution, and dependent instructions within the pipeline have been notified. The initial state of the `Complete_Flag` has all registers set as completed.

When a register is renamed, the original register becomes unmapped: no instructions will refer to this register and all future references are assigned the register's new alias. The `Unmap_Flag` bit corresponding to this register is set to 1. Additionally, all references to the original register in subsequent instructions now reference the new alias register. The renamed register becomes mapped and its `Unmap_Flag` bit is set to 0.

While the mapping status is kept at the `Unmap_Flag`, it is the `Renamed_Registers` signal that has the original-to-alias mapping information. When a register is mapped, its Renamed_Registers value is updated to reflect the new destination. Future instructions which require a value from a register will use the appropriate value in the Renamed_Registers signal, which contains its most recent name.

The final signal in the Register Rename component is the Reg_Counters signal. When instructions request values as their source operands, the processor must ensure that these read requests have been handled before a register is reused for further renaming. Even if the instruction for a register has been completed and has been unmapped, we can not reuse the register until all read requests from it have been serviced.

Figure 4.8, created by Moudgill et al., shows the process of renaming registers with these four signals (for a single scalar processor). The Op-Fetch stage in the figure is equivalent to our Instruction Dispatch, while the Execute Stage corresponds to our Execution Units. The final Write-Back stage is equivalent to our Common Data Bus, where the results of the execution are broadcast.

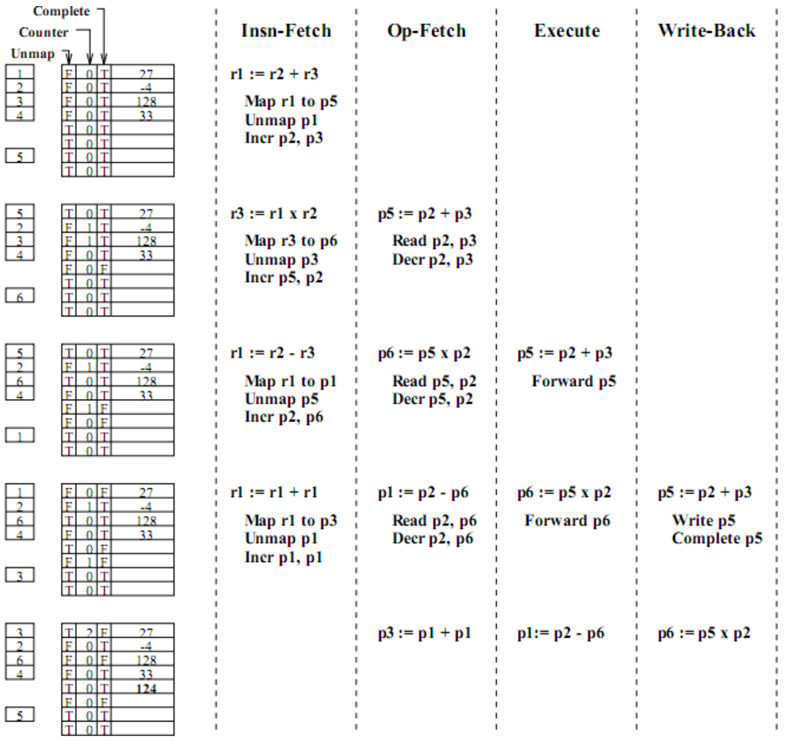


Figure 4.8: Mapping and reclaiming registers - The image shows the process of mapping, unmapping, and reclaiming registers under the techniques implemented above. Image taken from ”Register Renaming and dynamic Speculation: an Alternative Approach” [83]

The combination of an unmapped and completed register with no pending read requests creates a reclaimable register for further register renaming. As long as there are available reclaimable registers, the Register Rename unit will continue doing its job unhindered. Since the search for reclaimable registers must complete within one clock cycle, we had to make sacrifices in the search algorithm. Priority encoders were used to find the first `Instruction_Issue` contiguous reclaimable registers from the pool of architected registers. So, for a system with an `Instruction_Issue` parameter of 4, the priority encoders will locate the first instance of 4-contiguous reclaimable registers.

The Register Rename component is at the heart of many performance tradeoffs in the processor implementation. A large Register File creates a large renaming pool, which aids in preventing stalls due to lack of reclaimable registers. However, that same large Register File degrades clock performance as the priority encoder's search involves more levels of logic and increased delay. Further, because of random access to the registers, fragmentation of reclaimable registers occurs. This might prevent the priority encoders from finding a suitable contiguous block of reclaimable registers, even though the total of reclaimable registers might exceed this need in non-contiguous space. Instead of stalling at this pipeline stage, a backup search is made for a single reclaimable register. If at least a single register is found where a set for contiguous ones failed, then the Register Rename stage will still rename at least one its incoming instructions, effectively preventing a deadlock for the register resources.

The Register Rename implementation complexity also increases depending on the `Instruction_Issue` width. Since dependencies of issued instructions need to be solved in each clock cycle, each extra issue involves increased complexity and delay. Source registers need to be renamed if the previous renamed instructions in that clock cycle has the same destination. Code 4.5 shows the pseudo code for this dependency-solving algorithm. The space complexity is $\mathcal{O}(n^2)$, where n = `Instruction_Issue`.

Code 4.5: Register Renaming Issue Dependencies

```
FOR instruction IN 0 to Instruction_Issue-1 LOOP
    -- Analyze destination and source registers
    Renamed_Destination(instruction)    <= Next_Free_Register;
    Complete_Flag(Renamed_Destination) <= '0';
    Unmap_Flag(Renamed_Destination)     <= '0';
    Unmap_Flag(Original_Destination)    <= '1';
    FOR previous_instruction IN 0 to instruction-1 LOOP
        -- Check if the sources point to a newly renamed destinations
        IF Source_1 = Original_Destinations(previous_instruction) THEN
            Source_1 <= Renamed_Destination(previous_instruction)
        Counter(Source_1) <= Counter(Source_1) +1;
        IF Source_2 = Original_Destinations(previous_instruction) THEN
            Source_2 <= Renamed_Destination(previous_instruction)
        Counter(Source_2) <= Counter(Source_1) +1;
    END LOOP;
END LOOP;
```

Besides renaming destinations and sources, the Register Rename component also has to keep track of outstanding read requests. Each instruction's sources increment the (renamed) source register count by one. Some instructions, like those with immediate addressing modes, only require one register to be accessed, with the other value passed inside the instruction. Other custom instructions might not even require accessing registers. The Register Rename component also monitors read requests made to the Register/Future File by the Instruction Dispatch component. When a register has been read, the counter decrements accordingly. When a register's counter reaches zero, it can be reclaimed if it is unmapped and completed.

Finally, the renamed instructions are sent to their corresponding execution unit in the next stage.

4.3.2.5 Execution Unit Demultiplexer

The Execution Unit Demultiplexer reads the target EU for each instruction. With that information, it routes the instruction to one of its multiple output ports. The number of outputs ports is equal to the variable `Number_of_Execution_Units`, with each output port corresponding to each EU defined in the system.

The Demultiplexer routes each of the issued instructions to its corresponding execution unit, but only one instruction can be routed per EU at a time. When contention for one of the outputs occurs, only the first instruction (in terms of issue index and sequence order) gets routed to the appropriate output. The remaining instructions are saved in a register, and scheduled for routing in the next clock cycle.

Not only can issued instructions try to use the same EU, but each issued instruction can use any execution unit. Figure 4.9 shows how issued instructions are routed to their corresponding execution units. Because an instruction at this component's input can be routed to any of its outputs, the resulting circuit follows the behavior of a demultiplexer. In contrast, the decision about which of the multiple inputs gets each individual output behaves as a multiplexer.

Instead of defining the behavior of this component explicitly in terms of multiplexers and demultiplexers, we used VHDL to express the behavior as a procedural statement. This resulted in a better mapping onto the FPGA LookUp Tables during synthesis.

From this pipeline stage onwards, the area and performance are mostly dependent on the variable `Number_of_Execution_Units`.

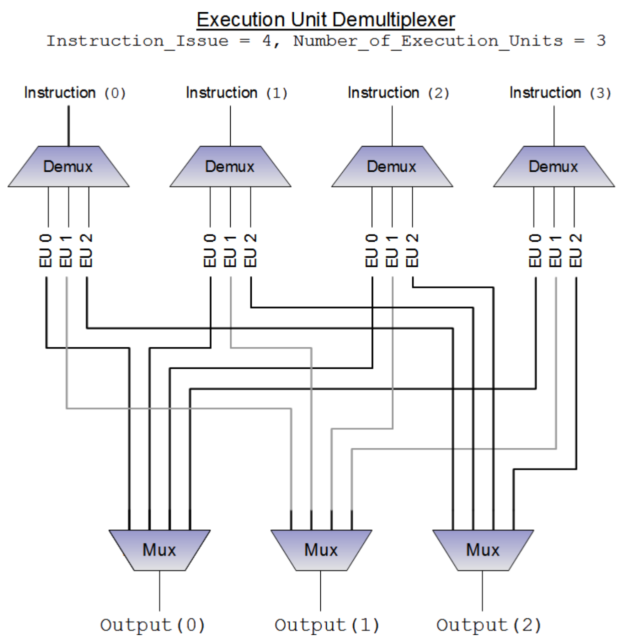


Figure 4.9: Demultiplexing instructions - Each of the four instructions can be sent to any of the three execution units. The decoder sends every instruction to each EU and a multiplexer selects among them.

4.3.2.6 Instruction Dispatch

The Instruction Dispatch stage resolves contention for the memory read ports. Each instruction can request read access to at most two source registers. This potentially creates the need of two read ports per instruction in the Register File. Realistically, we can not satisfy this multi-port requirement in hardware without paying a huge cost in delay and complexity, so we use dual-port memory instead. Howevwer, multiple instructions can request to use the same memory port at the same time, creating a many-to-one relationship. To make matters even more complicated, memory in FPGAs are limited to two ports and a designer must reserve at least one of them as a write port. This leaves the designer in the disadvantageous position of having only a single port of read access. Clearly, multiple memories are needed to allow multiple instructions to access the Register File in a timely fashion.

Our solution was to interleave the Register File, allowing the use of multiple memories to provide extra read (and write) ports, while spreading all read/write accesses among the interleaved memories. Further, we duplicated the contents of the Register File into another memory set, creating copies which supply extra read ports to access the same information, as long as the information is kept coherent among all copies.

The system variable `Register File memory organization` is really composed of two different variables: `Interleaving` and `Copies`. These allow us to set an appropriate interleaving amount and number of copies for the Register File.

The total number of outputs of the Instruction Dispatch stage is set equal to the result of `Interleaving` x `Copies`. Each of these outputs allows the servicing of a read request. We then route each instruction request to its appropriate output, in a fashion similar to what we did in the Execution Unit Demultiplexer stage. We also don't immediately block conflicting port requests. Because we set multiple copies of the Register File, each interleaved port can service a number of requests equal to the value of `Copies`.

A second function of the Instruction Dispatch stage is monitoring each EU's reservation stations. If a reservation station is full, the flow of instructions has to halt until the station becomes available to store new instructions. To minimize the amount of halting due to this conflict, we set the mechanism to halt only when a reservation station is full and an instruction is being dispatched to that station. If both of this conditions are met, the pipeline stalls, otherwise, the instruction flow continues unhindered.

Finally, this stage assesses if a memory port is actually needed by reading the `Complete_Flag` signal from the Register Rename stage. If an instruction has not been completed, it means that its result has not been broadcast and written. Thus, it is not necessary to read the Register File at that location; it contains an obsolete value. Instead, we marked the read request as *in-flight*, postponing the retrieval of the up-to-date value until the reservation station stsge. Due to the high-level of interdependency between instructions, requests for source registers are usually made before previous instructions have finished writing to them. Hence most of the requests will be marked as *in-flight*. This situation is beneficial because it helps reduce contention for the memory ports, which consequently avoids halting our system's pipeline.

4.3.2.7 Register File/Future File

So far, we have used the terminology Register File to indicate the location of values for each register. However, the term Register File is an oversimplification of the multiple locations in which a register's value can be located. Even under a scalar RISC architecture, the value of a register might not have been written to the Register File after execution, yet it is passed to dependent instructions through data forwarding. For our superscalar architecture, whenever an instruction requests to read its source operands, the origin of these values can be read from three different types of instructions and sources:

1. Instructions that have been completed in order. Read requests are made to the Register File.

2. Incomplete instructions that have been executed and whose results have been broadcast. Read requests are made to the Future File.

3. Instructions that have not been executed or are currently being executed. The sources are made available during broadcasting by the Common Data Bus.

The third case is handled by the reservation stations and explained in the following section. For the first case, when the ReOrder Buffer completes instructions in order, we write that value to the Register File. Usually, by the time an instruction's result is written to the Register File, an updated value for the register is moving through the pipeline, and, therefore, the Register File should not be accessed. However, the Register File plays an important role when recovering from precise interrupts. When an interrupt happens, the only location that stores in-order information enabling instruction flow recovery is the Register File. After each interrupt (which can be a branch operation changing the normal instruction flow), we flush the pipeline, but we make the Register File retain the latest in-order values. We then access these values to restart the pipeline's instruction flow.

In-order completion through the Register File is necessary for precise interrupts. However, we are letting instructions execute out-of-order to exploit performance benefits. If dependent instructions have to wait for the results of previous instructions to complete in-order, these benefits are lost. To prevent this, the Future File stores the latest broadcasted out-of-order values for each register and makes these values available for dependent instructions as soon as possible. This satisfies the conditions of the second case, in which an instruction has been executed but not completed. By using a Future File in our system, we avoid implementing a centralized ReOrder Buffer, whose associative lookup circuitry would make our system slow and more complex.

As described earlier, the structure of the Register File is dictated by the system parameters `Interleaving` and `Copies`. Figure 4.10 shows an example structure and how `Interleaving` spreads read requests, while `Copies` increases the number of read ports. We connect together the write ports of all Copies in each interleaved set, so that all write requests affect copies uniformly. This solution maintains value coherence among all copies.

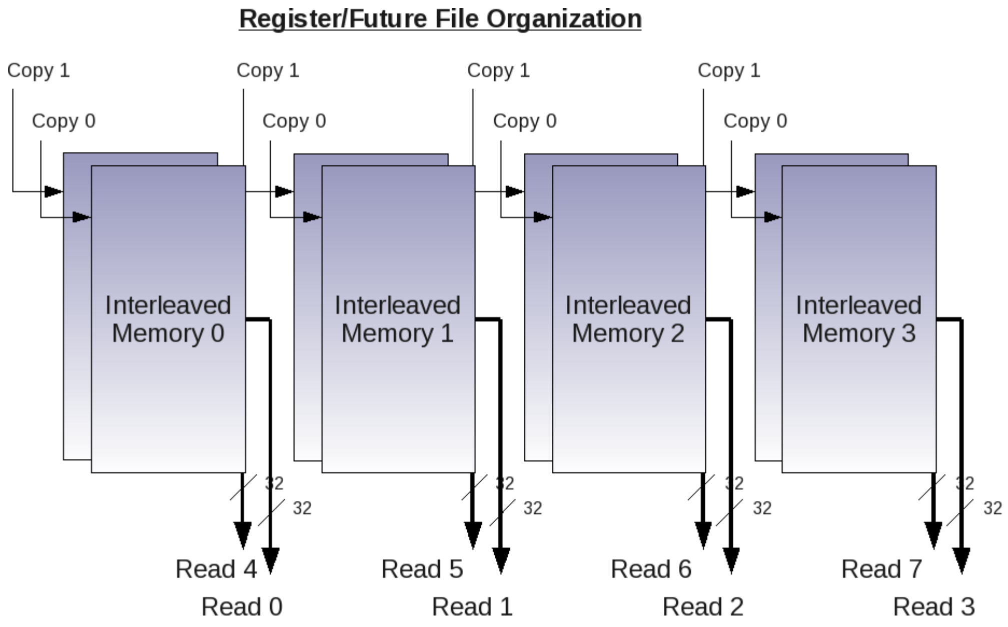


Figure 4.10: Register/Future File Organization - Resulting organization of Register and Future file with `Interleaving`=4 and `Copies`=2.

While the Register/Future File structures were modeled using BlockRAM memories, their small sizes were implemented during synthesis as distributed RAM rather than BlockRAMs. This prevents the use of a fixed-size 16 KB BlockRAM which will only be sparsely filled. Regardless of this implementation, the Register and Future File functionalities remains the same: to service read and write requests for register values. Each register's up-to-date value can reside in either the Register File or Future File, so the results from both of them are multiplexed before being sent to the outputs. The multiplexing is based on which of the two has the most recently updated value, which is updated whenever there is a broadcast or an interrupt. We then forward the results to the reservation stations.

4.3.2.8 Reservation Station Manager

The reservation stations receive instructions from the Instruction Dispatch stage and store the instructions until all of their operands are ready. The Instruction Dispatch stage previously made read requests to the Register/Future File for the instructions' operands which were not *in flight*. The resulting outputs from these requests are checked to match those of the operands, and upon a match, we set the operand as valid. If the instruction contains operands which are *in flight*, then the reservation station waits until the results from previous operations are broadcast. Once a match is made, we set the operand as valid. Figure 4.11 shows the necessary structure of an reservation station entry that stores its current state. When all of the instruction's operands become valid, the instruction is sent to its target execution unit.

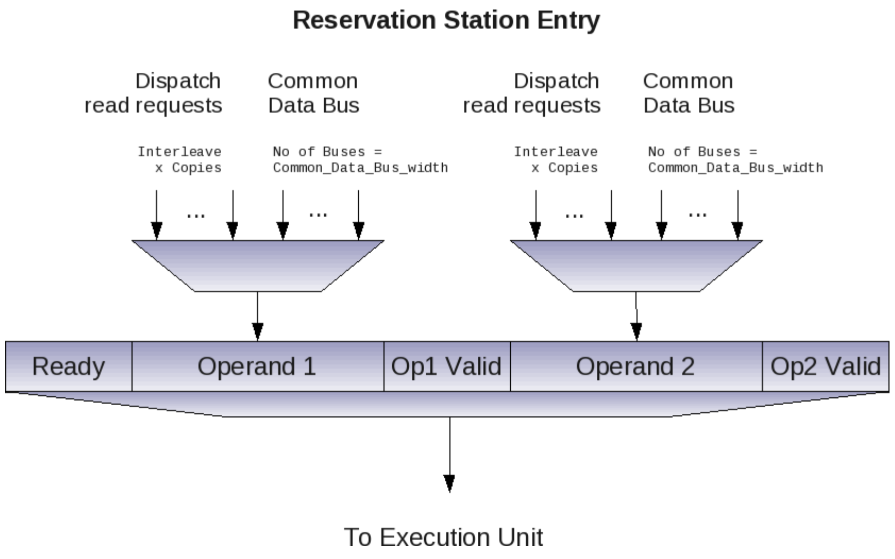


Figure 4.11: Reservation Station Entry - The dispatch read requests and results from the result buses are read until all operands are valid, making the reservation station entry valid and ready to be executed.

A second function of the Reservation Station Manager is instantiating all combinations for the reservation stations. Normal superscalar microprocessors include a set size for reservation stattions for each execution unit. However, our system allows for a lot of flexibility when organizing reservation stations. Hence, a designer can select a specific size for each execution unit's station. For example, a designer might assign a larger reservation station size to the EU for load/store operations than the others, since these operations make up for the majority of instructions in sequential programs. If `EU(0)` is the execution unit for load/store operations, he could assign the size of this EU's reservation station to be 4, while setting the size for the two others to only 2 (for `EU(1)`) and 3 (for `EU(2)`). Code 4.6 shows how that can be achieved in our system through a single line of VHDL code that feeds into the processor architecture definition. This example follows the reservation station organization presented previously in Figure 4.3.

Code 4.6: VHDL reservation station definition and assignment for a system with `Number_of_Execution_Units = 3`

```
TYPE     EU_Reservation_Stations_Type
         IS ARRAY (0 TO Number_of_EUs-1) OF INTEGER RANGE 1 TO Max_RS;
CONSTANT EU_Reservation_Station_numbers :
         EU_Reservation_Stations_Type    := (4,2,3);
```

A potential problem with any reservation station is that it can become full. When this happens, the station is not able to receive newly dispatched instructions for its EU, and consequently they stall the pipeline. A full reservation station can be the result of its small size, long-latency execution units, or a system instruction issue which is too high to be serviced in a timely manner. To avoid penalties, a designer can set reservation station sizes that allow dependencies to be resolved at an appropriate rate. Our flexible system allows this kind of customization.

Multiple instructions within a reservation station can be ready at the same time. That is, they all have their source operands valid. This creates a situation in which the reservation station needs to pick one from among all possible readied instructions; it is a decision that involves complicated non-scalable hardware and increased delay. To solve this, our reservation stations take the simplistic approach of waiting until the first instruction has its operands readied before servicing the next, in FIFO order. This approach only incurs a minor performance hit of about 0.6% for an instruction issue width of 2 and 2% for an issue width of 4 [77]. It is an appropriate solution that prevents us from having to implement complex priority schemes for multi-sized reservation stations that might degrade performance in even larger amounts.

4.3.2.9 Execution Unit Manager

The Execution Unit Manager instantiates all different execution units in the system, allowing multiple EUs to operate in unison and exploit intrinsic instruction level parallelism (ILP). With this component a designer can select which EUs fit best in their particular application, and even add custom logic. It has a standard interface for each

EU, with inputs corresponding to two operand values and useful instruction information like the original destination (to the Register File), the renamed destination (to the Future File) and the instruction's order number. The EU output forwards these signals to the next stage, except that the operands are replaced by the computation result.

Each Execution Unit indicates if it is currently busy or executing an operation through a 'ready' output bit. Simple combinatorial or arithmetic operations like those of an ALU finish in a single clock cycle. Therefore, the ALU is always ready for new instructions and its ready bit is 1 at all times. However, other EUs like floating-point operations can take multiple clock cycles to finish executing a single instruction, and during this time a value of 0 is assigned to the ready bit. When this happens, reservation stations are not able to deliver instructions readied for execution to the EU.

To avoid this potential stalling by high-latency operations, we allow these operations to be implemented in a fully-pipelined manner. If the state of the operation is saved at each pipeline stage, then potentially an EU can receive instructions each clock cycle without waiting for execution completion. A designer is then able to use these techniques to increase throughput, without having to modify the processor at the architecture level. This detachment from the underlying architecture allows a designer to concentrate on the execution units, while abstracting away our processor's implementation details. However, some other EUs might still implement I/O functions or other operations that require high-latency times to access off-processor devices like RAM memory. In these cases, waiting might be the only option.

The load/store/branch execution unit is implemented as a default in our system. This EU handles load/store requests, by instantiating the Data Memory as a component. Requests to Data Memory are abstracted away from the rest of the system and handled only inside this EU, according to the type of memory. This solution gives great flexibility as to how Data Memory is architected. It can be local fast-access BlockRAM memory, or off-chip larger memory like SRAM. Hence, when using single-cycle access BlockRAM the EU can use the memory's data and address input ports directly. Otherwise, when dealing with high-latency SRAM memory, a finite state machine may be implemented so it abides by the SRAM's handshaking protocols while dealing with the non-deterministic communication delays. In addition to load and store requests, the default EU also handles branch instructions. It executes the appropriate operands for the branch, and the result either sets the branch as taken or not-taken. For example a branch-if-equal (BEQ) operation will compare the operand to zero, and if they are the same, will set the branch as taken. The branch result, along with the instruction's order and the target PC address, is sent to the ReOrder Buffer and the Program Counter. The Program Counter will stop pumping new instructions into the pipeline until it receives confirmation from the ReOrder Buffer that all instructions leading to the taken branch have been written to the Register File. The pipeline flushes and the Program Counter restarts issuing instructions from the branch target address, as previously shown in Figure 4.4. If the branch is not taken, the instruction flow continues as usual.

Some instructions do not write to registers. For example, branches are purely control operations and do not produce a value that is used by any other instructions. Store operations write a register's value in Data Memory but do not change or modify any registers. Finally, a custom execution unit can be used for controlling a device, and even though the device operation might produce beneficial system results, the control instructions probably won't produce useful values for the rest of the instructions. In these cases, we can set an execution unit to mark the instruction as a non register-writing instruction. These instructions are logged in the system to ensure program order, but their results (if any) are never written to the registers by the ReOrder Buffer nor broadcast by the Common Data Bus. Our system also capitalizaes on this situation by not having them be part of the contention for resources in the ReOrder Buffer and Common Data Bus stages, and consequently avoiding stalls.

4.3.2.10 Common Data Bus

When instructions finish executing in their EU, they are sent to the Common Data Bus stage. Here, their results are broadcast to all waiting instructions in the reservation stations. As with many of the other pipeline stages, there are only a limited number of outputs, and contentions for them. The system parameter `Common_Data_Bus_width` defines the maximum number of broadcasted instruction results each clock cycle. While it is a variable, we set it to a constant value of 2 buses in our implementations. A designer might naïvely want to increase this number to allow more instructions to be broadcast at a time. However, this involves a heavy penalty in both hardware and place-and-route complexity, since each broadcast must reach every single entry of each reservation station. Pragmatically, the number of buses is best left at a low value. While immediate penalties from control hazards due to contention in the Common Data Bus might seem limiting, the perceived benefits are mostly imaginary. Even at high levels of performance, bus utilization is about 50-55%, and the performance benefit of going from 2 to 3 buses is less than 1% [77].

The Common Data Bus stage solves contention for its limited number of buses, but also adds an extra limitation: the results' destinations must have different interleaving values. By enforcing this limitation in our system, we allow broadcasting and writing to different interleaved input ports in the Future File at the same time. Once a result has been broadcasted, read requests waiting in the reservation stations are fulfilled. Subsequent read requests for braodcasted registers are then made to the Future File.

4.3.2.11 ReOrder Buffer

The broadcast of results is received not only by all reservation stations and the Future File but also by the ReOrder Buffer. This component stores all the out-of-order instruction results and then sorts them by issue order (which was attached to the instruction at the Program Counter) before writing them to the Register File.

Instead of handling a linked list as in software, we instead send insertion requests to all list entries, which we call shift-cells. Each shift-cell has access to the ordering number of its left and right neighbors. Given their neighbor information and the incoming order number of the new value being added, each entry decides autonomously if it needs to grab its left neighbor's values, its right neighbor's values, the new incoming values, or if it needs to keep its current values. Figure 4.12 shows a string of 4 shift cells that compare values against each other to ensure the single-cycle sorted insert into the list.

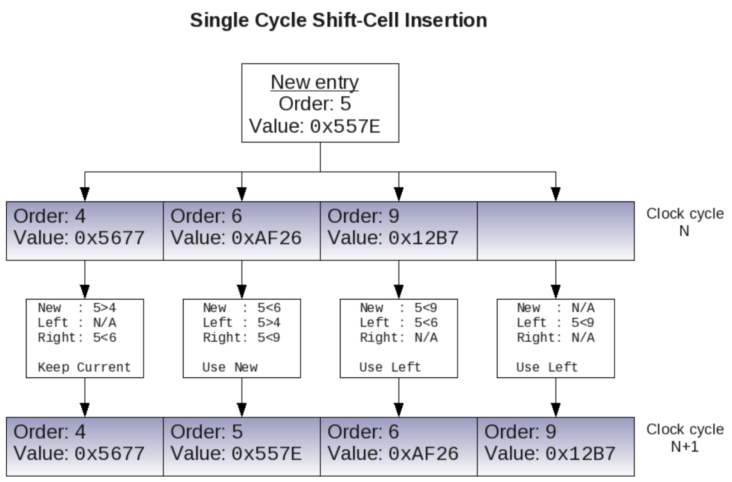


Figure 4.12: Shift Cell Logic - By comparing new entries with their left and right neighbors' contents, each cell maintains the sorted order in the ReOrder Buffer.

Our goal was to avoid having to sort an unordered list, which is definitely not suitable for a single clock-cycle hardware implementation because it is a time consuming task. The above solution makes all inserts into the list be in-order insertions. Therefore, as we receive a broadcasted result, it is inserted into the list at its corresponding place, and the ReOrder Buffer always has access to the earliest instruction. Because dynamic lists can't be created in hardware, we set the system parameter `Reorder_Buffer_size` to dictate how many entries the list will have.

When the ReOrder Buffer finds the next ordered instruction, it outputs it and deletes it from the list. The values for all entries shift to the left (each entry grabs their right neighbor's values) and the left-most entry is written to the Register File. Since the Reorder Buffer can both output a value and receive new values at the same time, the shift logic for each cell becomes slightly more complex. Regardless, sorted inserts still happen since values can shift left and right, or remain in place. Figure 4.13 shows how the ReOrder buffer receives new unsorted instructions and outputs sorted instructions in issue order, just like they came out of the Instruction Memory stage.

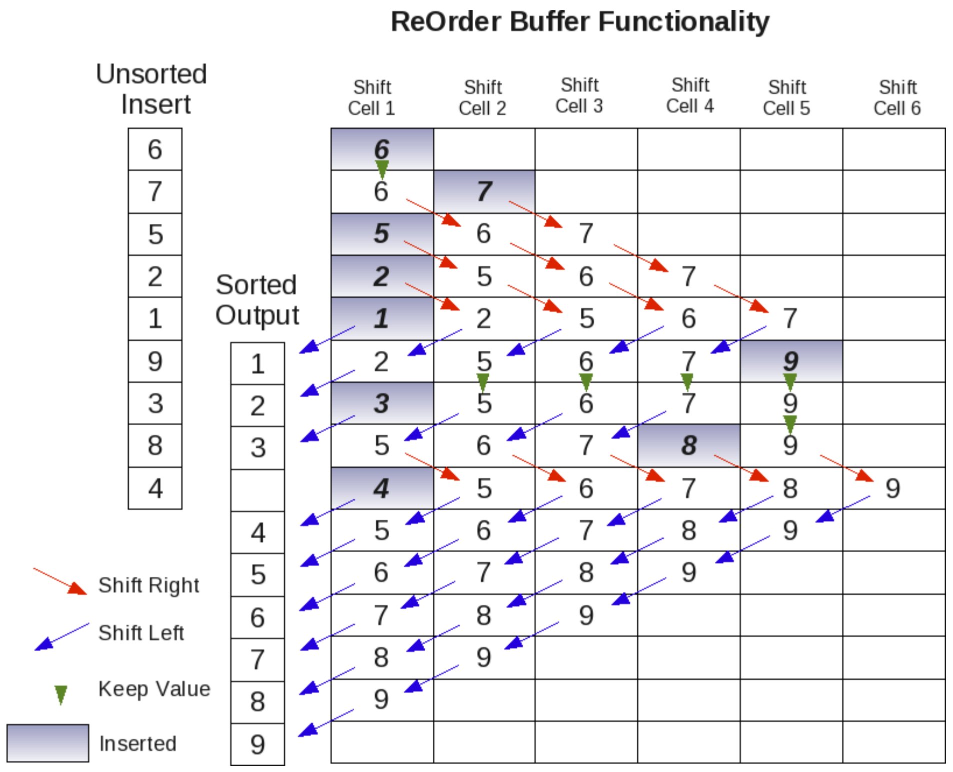


Figure 4.13: ReOrder Buffer Functionality - By examining their neighbors' contents and comparing them against the inserted new entries, each cell helps maintaining the sorted order in the ReOrder Buffer.

The shift-cell ordered queue only allows sorted insert for one instruction per clock cycle, which corresponds to the functionality of a linear sorter [84, 85, 86, 87]. However, we receive an `Interleaving` number of instructions from the Common Data Bus. We solved this problem by creating the same number of queues in the ReOrder Buffer, with each broadcasted instruction being inserted in its particular queue [88]. The outputs do not need to be ordered or arbitrated again because each queue output is directly connected to the corresponding interleaved Register File memory write port and their copies.

The final function of the ReOrder Buffer is to complete branch instructions. For this, it receives information from the Branch execution unit. When this EU processes a branch instruction and determines it is *taken*, we sent the branch instruction's order number to the ReOrder Buffer. In this final stage of the pipeline, the ReOrder Buffer writes all instructions leading to the taken branch. When it services the *taken* branch instruction, a reset signal is sent to all components (except the Register File) and the target address is sent to the Program Counter. The reset signal flushes all intermediate speculative instructions and values from the pipeline, while the Program Counter restarts instruction flow at the target address. Our solution keeps the Register File with all in-order completed instructions and allows the system to restart the instruction flow normally.

4.3.3 Interconnect

All the components we have described are connected in a pipeline manner as shown in Figure 4.3. The connections between each stage are implemented as registered outputs. The reason for this is that we need the ability to hold output values in case of a control hazard, that is, if a stage further down the pipeline is unable to process new instructions. This situation is a natural consequence of contention for the stage's resources or outputs, which forces new instructions to wait until unserviced ones have been dispatched from a stalled stage.

4.3.3.1 Pipeline Control

Each stage in which there is contention for resources can produce a control hazard. Resolution of control hazards generally involves saving blocked requests for resources in registers and then servicing those requests during the next clock cycle, repeating this process until all saved requests are serviced.

The list of all possible hazards locations and reasons are shown next.

- Jump Decoder - This stage does not stall the pipeline. However, upon detecting a jump instruction, it sets the Program Counter accordingly and sets a flag to ignore any obsolete incoming instructions. The Jump Decoder's output resumes when the jump instruction is retrieved from the Instruction memory.

- Register Rename - Hazards occur when trying to rename more instructions than the amount of available free registers. If there was at least one free register, the hazard still goes off, but one stalled instruction is renamed.

- Execution Unit Demultiplexer - Only one instruction can be forwarded to each execution unit output. Otherwise a hazard is created.

- Instruction Dispatch - Unserviced instructions due to contention for the Register/-Future File memory ports create a memory hazard.

- Reservation Station Manager - The reservation stations can get full, prompting a hazard to occur when a dispatched instruction is sent to them.

- Common Data Bus - Only `Common_Data_Bus_width` number of instructions can broadcast their results. Hazards occur when additional instructions complete their execution at the same time.

- Reorder Buffer - Hazard occurs when at least one of the shift cell sorting queues is full and it receives a new value.

The only remaining source of control hazards are branch hazards. These occur when a branch is taken, prompting the flushing of any subsequent intermediate signals. The Pipeline Control stage sets the appropriate reset signals for all intermediate stages except for the Register File, which contains the completed in-order instructions necessary to restart the pipeline.

To arbitrate multiple concurrent hazards and stall only necessary stages, we created a control system which prioritizes hazards depending on the dependencies between stages and the type of hazard.

4.3.3.2 Design Hierarchy

We initially implemented the prioritization of pipeline hazards in a distributed manner. Every stage had access to other dependent stages' hazard signals, using it to assess whether they could process new instructions. Our initial implementation was not an optimal solution because it recreated the same combinatorial logic multiple times at different stages. However, it proved to be a scalable solution while each stage of the processor was being implemented. Eventually, this implementation was scrapped and replaced by a single Pipeline Control component, which was a centralized control scheme in which a priority hierarchy was assigned to each stage.

The hierarchy for pipeline hazard stalls was mostly anti-symmetrical in comparison to the pipeline stage's depth. Deeper stages had priority, because a hazard meant that probably every single stage preceding it would have to stall until the hazard was solved, preventing instruction flow. Table 4.1 shows stage-hazard dependencies. Stages that can create hazards are located on the top of the chart while stages stalled by these hazards are located on the left. The Register File stage is never stalled since this stage does not have any contention for resources (they were solved during Instruction Dispatch).

		ReOrder Buffer	Common Data Bus	Reservation Station	Instruction Dispatch	EU Demultiplexer	Register Rename
1	Program Counter	X	X	X	X	X	X
2	Instruction Memory	X	X	X	X	X	X
3	Jump Decoder	X	X	X	X	X	X
4	Register Rename	X	X	X	X	X	
5	EU Demux	X	X	X	X		
6	Instruction Dispatch	X	X	X			
7	Register File						
8	Reservation Station	X	X				
9	Execution Units	X	X				
10	Common Data Bus	X					
11	ReOrder Buffer						

Table 4.1: Hazard Hierarchy Label Pipeline stages (left) stalled by hazards (top)

Each individual stage sends hazard signals to the Pipeline Control component to be prioritized accordingly. Then, our centralized system replies with necessary halt signals to all the dependent stages, making them halt their execution.

4.3.4 Protocols

Communication between the processor's stages has to happen at a resolution of one clock cycle. Because this setup only allows for simple combinatorial logic, communication protocols cannot be readily implemented in our system. This limitation is why our processor relies on hazard and stall signals from the centralized Pipeline Control component. Within this component, a simplified preemption system prioritizes hazard signals and makes active pipeline stages stall their predecessors.

Our processor's default data storage system is single-cycle latency memory implemented in BlockRAMs, which does not need communication protocols to control it. However, BlockRAMs can only hold a limited about of memory, and a larger amount of memory in SRAMs might be needed for larger applications. An SRAM controller involves a handshaking protocol with acknowledgments to requests. The requests are quite possibly serviced with non-deterministic latency if the requests go over a standard bus which requires arbitration. To handle data read and write requests, a designer can implement this protocol in the load/store execution unit through a finite state machine.

The presence of other protocols in our system is possible through custom custom execution units. These EUs might interface with off-processor devices or even other processors. Due to the application-specific nature of such systems, the designer must optimize such protocols for their specific system.

4.4 Processor assessment

Once the processor structure, components and connections were assembled, the processor functionality was assessed through software execution and synthesis. This process involved multiple simulations followed by debugging, which led to a more robust assessment of the processor's capabilities, limitations and reconfigurability. Hephaestus' instruction set architecture exposes this functionality to software, serving as an interface between a program's instructions and the processor's underlying semi-custom hardware.

4.4.1 Instruction set architecture

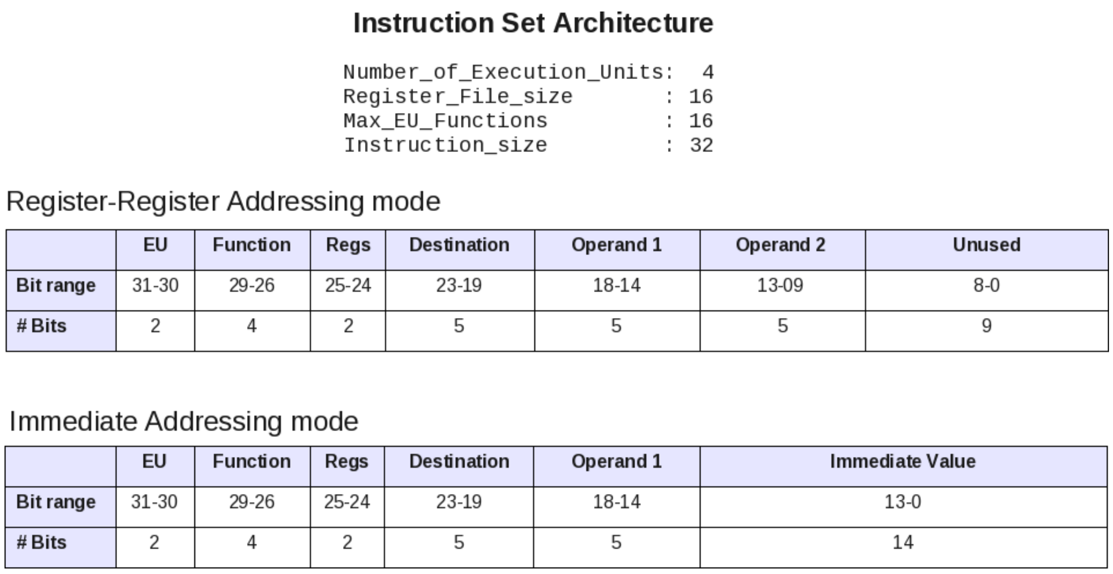
Our extendable superscalar architecture provides the benefits of mainstream optimized general-purpose processors and custom hardware specialization for a variety of computational paradigms. However, this flexibility means that we cannot adhere completely to a standard instruction-set architecture (ISA). Because we allow custom execution of instructions, it necessarily follows that our ISA should be flexible enough to incorporate encodings for these custom instructions. Even though RISC-style ISAs are becoming a poor match for rapidly changing underlying fabrication technology [89], it would be unproductive to require switch to a completely different ISA . In order optimize the architecture, we created a solution that uses the most common instructions and addressing modes in the MicroBlaze processor [15]. The default processor configuration implements an ISA which includes register and immediate addressing modes for ALU and Load/Store operation. However, the ISA is flexible enough to accommodate new or additional custom instructions.

To differentiate among addressing modes, we reserved two bits in the instruction width. Each of these bits corresponds to each of the instruction's operands. The bits for not-needed operands are sent unchanged through the pipeline; thus, hard-coded data values can be attached to the original instruction for immediate addressing. Jump instructions don't need any of their operands, so the whole instruction is left unchanged and operands are concatenated with the immediate bits to create a larger immediate value.

The target execution unit and the functions within the EU are encoded in necessary instruction bits. There is no global method for doing this because of custom EUs. However, we explicitly define the maximum number of functions per EU to automatically set the encoding width to the minimum number of bits.

The remaining bits in the instruction are left for encoding the access registers for the destination and operands. Depending on the user-set width for the instruction, an appropriate number of bits will be padded to the end of the instruction, comprising immediate value bits. Figure 4.14 shows the structure of a 32-bit instruction with encoding for four execution units and a maximum of sixteen functions for each EU, and a register file size of sixteen registers (with thirty-two physical registers).

Even though the locations and sizes of the ISA components change, they are directly correlated to the parameters that a designer chose when creating the semi-custom system. However, the designer can also override these automatically calculated minimum-



Instruction Set Architecture

```
Number_of_Execution_Units:  4
Register_File_size       : 16
Max_EU_Functions         : 16
Instruction_size         : 32
```

Register-Register Addressing mode

	EU	Function	Regs	Destination	Operand 1	Operand 2	Unused
Bit range	31-30	29-26	25-24	23-19	18-14	13-09	8-0
# Bits	2	4	2	5	5	5	9

Immediate Addressing mode

	EU	Function	Regs	Destination	Operand 1	Immediate Value
Bit range	31-30	29-26	25-24	23-19	18-14	13-0
# Bits	2	4	2	5	5	14

Figure 4.14: Instruction Set Architecture - The number and location of bits are dependent on four system parameters that use minimum binary encoding length.

length binary codes with larger widths to provide a more uniform instruction structure across multiple configurations, allowing a simpler encoding of software instructions.

4.4.2 Software execution

The initial selection of software programs to execute in our processor included programs like Dhrystone and other benchmarks. Dhrystone is a synthetic integer benchmark for assessing performance characteristics. However, we quickly found out that synthetic benchmarks could not provide a good assessment of our application-based processor, and that Dhrystone was not appropriate as it does not mimic modern workloads within specialized processors [90]. Other more robust application-based benchmarks were not used since we did not have a compiler for our processor. Compilers generally target sequential execution in time (software) and not parallel execution in space (hardware). The necessary work to create a compiler for a polymorphic architecture provides an interesting tangential research topic in reconfigurable computing, but it is beyond the confines of this dissertation. Unfortunately, the lack of a compiler heavily hampered our ability to evaluate our processor through more robust methods such as performance benchmarks and complex applications.

Matrix multiplication was used as the testing software in our processor. Due to the complexity of writing custom assembly code, the application complexity was kept at a minimum. The processor executed the multiplication of two matrices with sizes 2×3 and 3×4 for a resulting matrix of size 2×4. The result of the matrix multiplication is shown below:

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \end{bmatrix}$$

where

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31}$$

and in general with $m = 2$, $n = 3$ and $p = 4$, $1 \leq i \leq m$, $1 \leq j \leq p$,

$$C_{ij} = \sum_{r=1}^{n} A_{ir} \times B_{rj}$$

The execution of this program requires 24 multiplications and 16 additions, not counting the extra operations necessary to keep track of indexes and address location offsets. There are multiple dependencies for calculating each value, making it an appropriate program to run an test in a soft-core processor. Its lack of custom instructions also makes it suitable to compare our implementation against that of a MicroBlaze. Table 4.2 shows the results obtained when executing the program in our processor and in a MicroBlaze system for the same Xilinx Virtex 5 board.

	MicroBlaze	Hephaestus
Frequency(MHz)	200	137
Execution (clock cycles)	377	208
Execution time (μs)	1.89	2.04

Table 4.2: Soft-core processor execution comparison. Results obtained when running matrix multiplication in Hephaestus and MicroBlaze

The execution results show that while competitive, our processor is outperformed by the MicroBlaze in overall timing. This is to be expected as MicroBlaze is a mature product that has undergone many revisions. However, MicroBlaze's optimization relies on using particular board resources (which hamper portability) and utilizing an optimized and immutable ISA (which prevent customization). On the other hand, Hephaestus' structure is independent of device-specific components, allowing portability at the expense of performance tweaking. Additionally, our flexible ISA allows custom instructions and execution units, making application performance gains through local parallel execution possible. The performance results reflect the performance for general-purpose multiplication and not a customized processor with for example dedicated cryptographic or digital signal processing instruction cores.

4.4.3 Simulation and debugging

ModelSim [76] was used to test all the components and stages. Initial testing began in simulation, with test benchmarks set up to stimulate the system with a flow of instructions.

The Instruction Memory component was modified to initialize its values from a text-file in both simulation and implementation, which resulted in a merging of our simulation and implementation processes. From that point on the only difference between the two was that the simulation file stimulated the clock signal while the implementation connected it to the device's global clock.

The simulation in Figure 4.15 shows the instruction flow from an initially empty system. These instructions traverse the length of the pipeline from one stage of the pipeline to the next. This traversal is represented visually in the horizontal time axis, with the instructions moving left to right in time while crossing each pipeline stage. The numeric data values shown in the simulation look esoteric because they are the encoded values used by the processor.

The simulation is the result of feeding the processor the instructions for a matrix multiplication of two matrices with sizes 2×3 and 3×4. The matrices were defined as

A=

$$\begin{bmatrix} 1 & 3 & 4 \\ 2 & 0 & 1 \end{bmatrix}$$

B=

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 2 & 1 & 4 \end{bmatrix}$$

Resulting in a multiplication matrix with values

C=

$$\begin{bmatrix} 19 & 16 & 13 & 23 \\ 5 & 6 & 7 & 6 \end{bmatrix}$$

To compensate for the lack of an appropriate compiler, a python script was used to convert assembly language for the processor to the instruction set architecture, using the minimum bit encoding lengths explained in section 4.4.1. The instruction and data

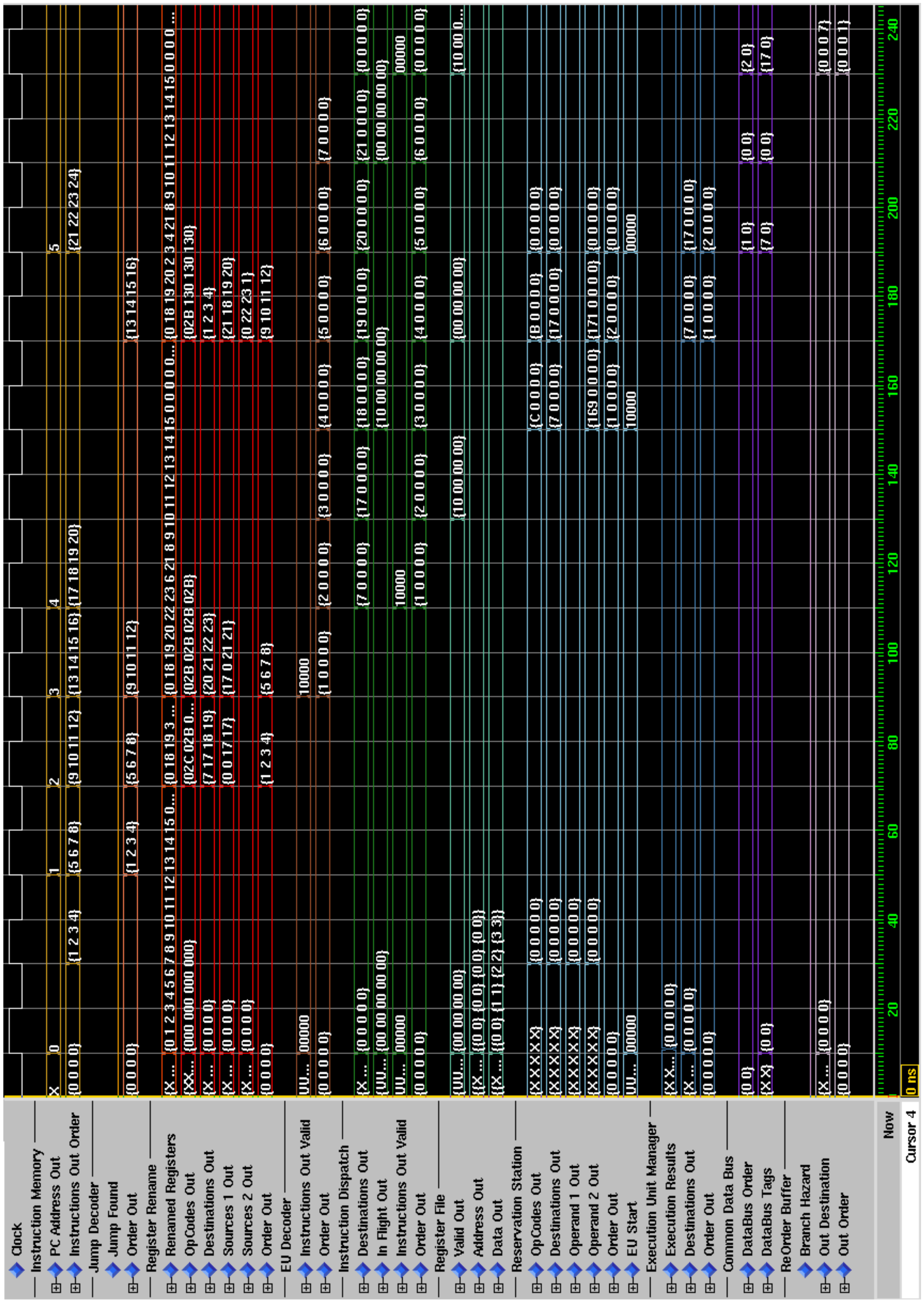
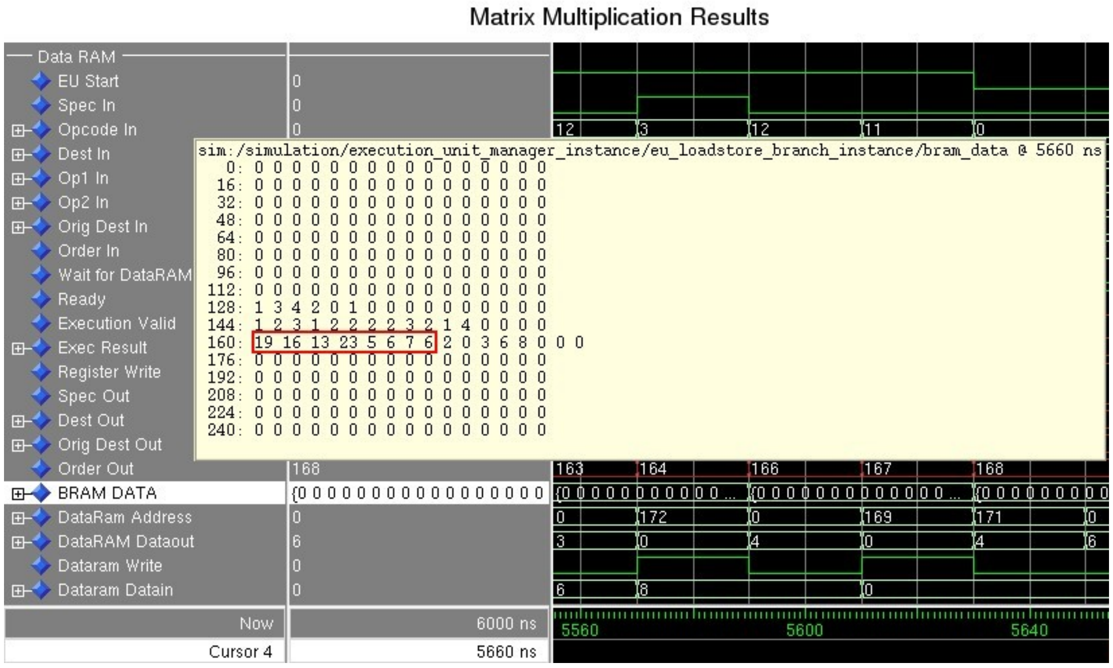


Figure 4.15: Pipeline Simulation - Pipeline traversal of instructions through all processor stages.

memories were initialized with the appropriate series of instructions and values before simulation and implementation. Locations for the matrices data values were assigned arbitrarily but ensuring there was enough allocated memory space to hold all the results and variable values. The resulting data written to memory for the space allocated to the C matrix were the correct matrix values from multiplying matrices A and B. Figure 4.16 shows the results that have been written in BlockRAM, after program execution, corresponding to a linear memory array implementation of the matrix C values 19, 16, 13, 23, 5, 6, 7, 6.



```
sim:/simulation/execution_unit_manager_instance/eu_loadstore_branch_instance/bram_data @ 5660 ns
  0:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 16:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 32:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 48:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 64:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 80:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 96:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
112:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
128:  1  3  4  2  0  1  0  0  0  0  0  0  0  0  0  0
144:  1  2  3  1  2  2  2  2  3  2  1  4  0  0  0  0
160: 19 16 13 23  5  6  7  6  2  0  3  6  8  0  0  0
176:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
192:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
208:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
224:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
240:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figure 4.16: Matrix Multiplication Results - The box shows the C-matrix data written to main memory after the matrix multiplication. The matrix values are stored linearly; A starting at address 128, B at 144 and C at 190.

The corresponding C code for the same array multiplication program in a MicroBlaze processor can be seen in Code 4.7. The execution of the compiled program results in the output illustrated on Code 4.8, thus achieving the same result as expected from both analytical methods and execution in our processor.

While a lot of time was spent on implementation and simulation of the processor, even more time was used for debugging the system during synthesis. The debugging process was mostly comprised of testing, simulating and synthesizing the functionality of individual components, and subsequently refining requirements, implementation and

Code 4.7: C code for Array Multiplication

```
for (i=0; i<m; i++) {
    for (j=0; j<p; j++) {
        for (k=0; k<n; k++) {
            C[i][j] += (A[i][k] * B[k][j]);
        }
    }
}
```

Code 4.8: Results from MicroBlaze Execution

```
Total execution clock cycles = 377

C[0][0] = 19
C[0][1] = 16
C[0][2] = 13
C[0][3] = 23
C[1][0] =  5
C[1][1] =  6
C[1][2] =  7
C[1][3] =  6
```

interfaces to fix errors. It was during this process that the tradeoff between ideal and applicable implementation defined the structure of our processor.

4.4.4 Synthesis

Once the main hardware implementation is done, synthesis was used for determining the maximum and minimum FPGA slot areas that the Hephaestus processor configurations can take. The synthesis process on Xilinx ISE 8.2.2 was set with an optimization goal for speed and a high optimization effort. Since many of the more complex processor configurations might not fit in our Virtex-II Pro evaluation board, the synthesis target device was a XC5VLX220 Xilinx Virtex 5 board, with a Configurable Logic Block (CLB) array of 160 rows x 108 columns, 138240 slices and 6,912KB of BlockRAM (192 36KB-BlockRAMs).

The main techniques used for synthesis involved the use of system-level parameters listed in section 4.1. These parameters were defined as a VHDL package file, which all processor components included. Through the use of this package, the system parameters were propagated to all structural, combinatorial and processing hardware-description language statements. Because of the high level of customization of our processor, great care was used to make sure all parameters could interact with each other in unison. For example, the implementation of the Register Rename component involves the use of ten system parameters defined at synthesis time.

1. `ISA_Registers`: How many registers can be addressed by instructions.
2. `Physical_Registers`: The size of the Register File, including rename registers.
3. `Reg_address_size`: Bits required for minimum binary encoding of register number.
4. `Number_of_EUs`: Number of execution units in the system.
5. `EU_Functions`: Maximum number of operations per execution unit.
6. `Opcode_Size`: Bits required for minimum binary encoding of EU, EU function and addressing mode.
7. `Immediate_Size`: Bits used for immediate addressing mode.
8. `Instruction_Size`: Total bits allocated for instruction, which include op-code, destination and two source registers, and immediate bits.
9. `Interleaving`: Number of write ports to the Register/Future File.
10. `Instruction_Issue`: Number of instructions issued per clock cycle.

Figure 4.17 shows the relationships between these ten parameters. They all contribute to define the generation of structures, loop limits, signal width and signal assignments for the Register Rename component described in section 4.3.2.4.

We limited the synthesis process to our specific design only. Hence, we did not make our processor a component of a larger system, nor did we attach it to a bus. Because there was no need to create a fully embedded system, we did not use Xilinx's Embedded Development Kit (EDK) software, which allows for the addition of user-defined IP-cores into a system. Instead we used their Integrated Software Environment (ISE) which focuses more on HDL-based synthesis. All processor communication was kept local, including instruction and data memories. This default design would synthesize directly onto an FPGA board without the need for other functional blocks. These other functional blocks used in embedded systems include MicroBlaze processors, processor local buses (PLB) and on-chip peripheral buses (OPB). The presence of multiple masters in the buses would add a non-deterministic amount of delay for communication over the bus. Even if no arbitration were needed, a minimum of five clock cycles would be used in bus communication delay.

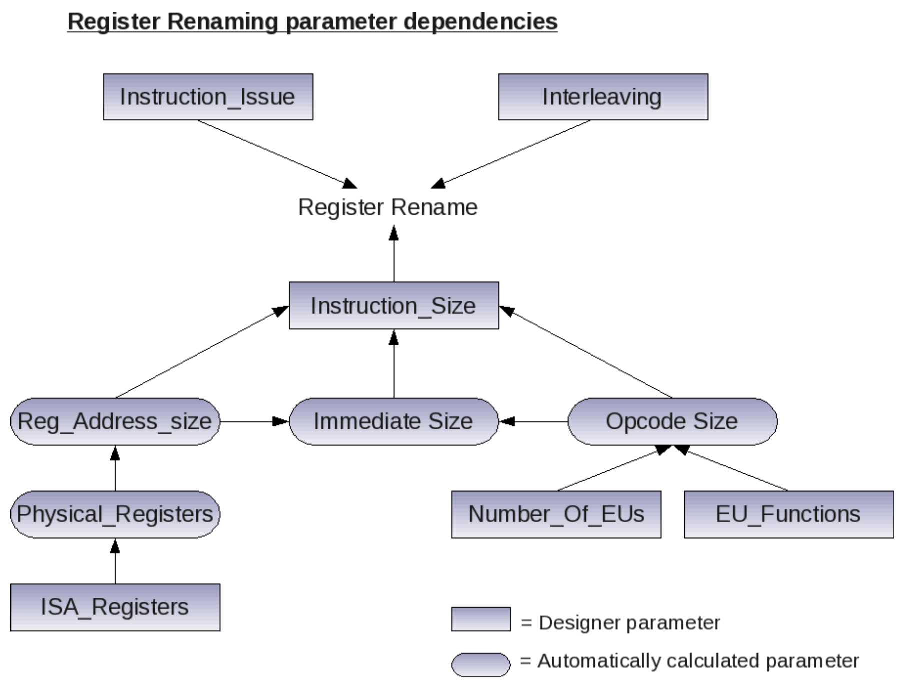


Figure 4.17: Synthesis parameter dependencies - The figure shows all parameters, assigned and calculated, necessary for register renaming.

Instead, we provide a fully autonomous and portable HDL processor solution which can be dropped and used as needed by a designer. The processor can be used for stand alone applications, multi-threaded hardware-based systems, and embedded systems. The HDL description of the processor might not provide the optimal synthesis in a specific device, but it does allow synthesis in multiple devices and FPGA families. This capability is consistent with current reconfigurable computing programming models, where threads can migrate their location from software to a hardware-based solution in multiple reconfigurable devices. Customization of a Hephaestus processor can act as a computational thread executing on synthesized hardware, enabling coarse grain thread-level parallelism (TLP) in a reconfigurable system. Multiple instances of identical synthesized processors can lead to symmetric multiprocessing (SMP), while synthesis of heterogeneous processors customized for specific tasks creates an asymmetric multiprocessing (ASMP) system.

Our solution additionally allows a designer to provide synthesizable reconfigurable custom logic, which can hone-in on exploitable application-level code that more generic processing cannot compensate for.

4.4.5 Reconfigurability

Another contribution of our project is providing processor reconfigurability through the use of high-level system parameters, in essence abstracting away the implementation details from a system designer. Our solution allows the designer to tailor the processor architecture to the particular application at hand, a solution not available with standard microprocessors. Even other soft-core processor implementations are limited to a rigid architecture. Instead, by weighing the costs and gains of a flexible Hephaestus processor, a designer is able to create an optimal solution, even in a confined area within an FPGA.

Area confinenment is important in reconfigurable computing because it can enable partial run-time reconfiguration. This type of reconfiguration not only reduces the overhead of having to reprogram the whole FPGA at once but also to reconfigure an area of the FPGA while the rest is still active. This ability is the key point of current research in reconfigurable computing [24, 91, 92, 93, 94, 95, 96, 97, 98], yet few researchers have addressed the issue of the costs of parallelism and performance in terms of reconfigurable area [19, 73, 99].

A global view of reconfiguration analysis shows a lack of fitting criteria to determine the overall costs of a system. Our processor provides a flexible solution for parallel performance gains through an optimized processor architecture, and also establishes a performance criterion based on deterministic area use (Chapter 6). This deterministic area, a requirement needed for partial run-time reconfiguration, limits the maximum area that can be used by a partially reconfigurable module. It is useful if such modules were to be a processing elements, and better yet independent processor cores like ours.

We can force all of our processor configurations in a multi-processor system to occupy the same amount of fabric space and geometry. This ensures that each processor can be replaced by any of its different instances, with minor disruption to the system. Because of the high dependency between processor configurations and their associated area costs, we delay the discussion of area constraints until Chapter 5, where we have establish what constitutes the area of a processor and how it can be used for slotted synthesis.

4.4.6 Limitations

Hephaestus is a very flexible processor architecture, but this flexibility imposes some limitations in our system. Communication support for microprocessor bus architectures like the Processor Local Bus (PLB) or the On-chip Peripheral Bus (OPB) is not an integral part of the processor design. If needed, a designer can use a PLB or OPB wrapper core to connect the processor to other devices attached to a bus.

Like the MicroBlaze, our processor connects to local memories for data and instruction. The sizes of such memories are often too small to store programs with large data sets. To solve that, a hierarchical set of memories is often used, and the accesses to memory are handled by a memory management unit (MMU). This is a solution which offers virtual memory, address translation and memory protection. However, many embedded systems do not need this level of memory management, and memory policies can be created by a designer to best fit its system. We previously mentioned that our data memory component was configurable and could be customized to an application, even enabling support for data cache. Additionally, even though our instruction memory initializes its values during synthesis, it also allows instructions to be written, like a normal memory unit.

The final limitation is the lack of a status register. Usually, this register stores information like arithmetic overflow, rollover accumulations and zero-result operations. However, since our processor executes out of order, access to this information might not reflect the in-order state of instruction execution, making the information ambiguous. The most common use of a status register is reading the zero-flag to determine the result of a *branch-if-equal* (BEQ) or *branch-if-not-equal* (BNE) instructions. Our processor instead reads the actual result when determining the result of a branch operation, followed by a comparison. Because we broadcast this result as soon as it is available, this generally result in no extra delay.

4.4.7 Complexity

Detrimental consequences of our processor included increased hardware fabric area use, as anticipated. Evaluation of these costs will be examined in detailed in Chapter 5. Additionally, when doing partial reconfiguration, area is bound to be wasted. To allow a constant area for this process, we need to account for our maximum processor area constraint, and force all configurations to use it. Hence, fabric area is bound to be wasted in simpler or heterogeneous processor configurations due to less extensive use of active resources, which is not mirrored in its use of bounded area.

The increased processor complexity also resulted in increased place-and-route complexity and time when fitting each processor configuration into the allocated area. It is clear to see how increasing the parallelism of the processor creates and increased need for resources. The next chapter evaluates not only this area increase but also the associated performance gains.

4.5 Summary

We used the VHDL hardware description language to define a vendor-independent and portable processor architecture. This superscalar processor allows the issuing of multiple instructions from memory while providing execution units for parallel execution of instructions. The processor architecture's structure is controlled through configuration variables, and structural details are automatically implementated in reconfigurable FPGA fabric, and hidden away from the programmer for ease of use.

The processor organization consists of an 11-stage pipeline, with in-order issue, out-of-order execution and in-order completion of instructions. Hazards created by pipeline stages were handled in a hierarchical centralized control unit. It solves contention for limited resources by stalling the pipeline, and multiplexing such resources in time. We set our interface with software with a RISC-like ISA that accomodates extensions for custom logic and user-defined execution units.

The simulation and synthesis runs showed the correct behavior of our processor. However, Hephaestus' flexibility also creates ambiguity on the performance gains acquired by sacrificing FPGA area.

Chapter 5

Evaluation

In order to analyze the area requirements of our superscalar processors, we synthesized configuration permutations with respect to the system's independent variables `Instruction_Issue`, `Number_of_EUs`, `Interleaving`, `Register_File_Size` and `Data_Width`. These configurations provided the full selection criteria that a designer might need for their particular system. Because performance for semi-custom circuitry, as with fully-custom circuitry, is based on the specific application, many performance metrics like integer and floating point benchmarks did not provide an adequate performance assessment. In many instances, such benchmarks were not even applicable to the circuit. Instead of using them, we generalized our performance metric to encompass the average potential case for a well-designed hybrid hardware/software program. User logic was not evaluated, but its restrictions and liberties when adhering to a synthesized system are discussed as part of our results.

Each configuration's synthesis resulted in a minimum time delay for correct synchronous signal propagation, thus setting the system's maximum processor clock frequency. Additionally, the synthesis process specified the total FPGA area and resource use. With this information, we evaluated our independent variables' impact in resource consumption, and analyzed their relationship to provide a deterministic assessment of area, a useful evaluation for partial reconfiguration.

5.1 Area use

As mentioned previously, our synthesis tool Xilinx ISE 8.2.2 was set with an optimization goal for speed and a high optimization effort for a XC5VLX220 Xilinx Virtex 5 board. This setup provided the largest processing performance while sacrificing FPGA area to achieve it. The processor was initially synthesized for the FPGA device and the default place-and-route setup (with no constraints) was used.

5.1.1 Synthesized area

Much of area used to synthesize our design resulted from low quality optimization by the synthesis and the place-and-route algorithms. Historically, area-inefficient designs result from the high complexity needed when mapping a system to finely granular hardware [100]. Because the capacity of modern FPGAs surpass millions of equivalent ASIC gates, routing becomes the dominant factor in the final performance of the design [101].

To reduce the synthesized area and consequently reduce routing, Xilinx provides relational placements macros (RLOCs), which specify the placement of particular elements with relationship to each other. RLOCs can potentially increase performance between 30% and 50% for common applications with high device utilization. However, this performance increase depends on the careful placement of static circuitry within a regular design structure [102]. Because of our processor's flexibility, its underlying structure varies wildly and is therefore not a good candidate for RLOC constraints. The usefulness of RLOC techniques is generally limited and we instead used the default automatic tool placement with competitive results. Even with larger processor areas, we could still constrain these areas for our purposes.

5.1.2 Area constraints

The potential benefit of our research is providing an area assessment for hardware processing, as part of ongoing field efforts to better use dynamic partial reconfiguration within FPGAs. Partial reconfiguration has multiple advantages, since it minimizes the reconfiguration overhead from a full reconfiguration, and allows static parts of the FPGA to continue running. To benefit from this technique, designers must use the aforementioned constraints to confine the synthesizable hardware area into a fixed synthesized area. For our project, we defined the area a processor core takes and the placement boundaries for the processor. This facilitates the placement of multiple processor instances (with possibly different configurations) within the FPGA, to exploit thread-level style parallelism (TLP). Additionally, the fixed area per processor allows the swapping of processor configurations in and out of the FPGA with minimum overhead through partial reconfiguration methods.

To start evaluating the area needs of processor configurations, we synthesized, placed and routed a processor with `Instruction_Issue` = 2, `Number_of_EUs` = 2, `Register_File_Size` = 16 and `Data_Width` = 16 bits. This configuration has its system variables near the average processor complexity, and as an average case, is a good indicator of area requirements for processing. Figure 5.1 shows the area used by this processor configuration in the target device, with color-coded information for the different area used per pipeline stage. This area reflects an automatic placement scheme with no area constraints by the user.

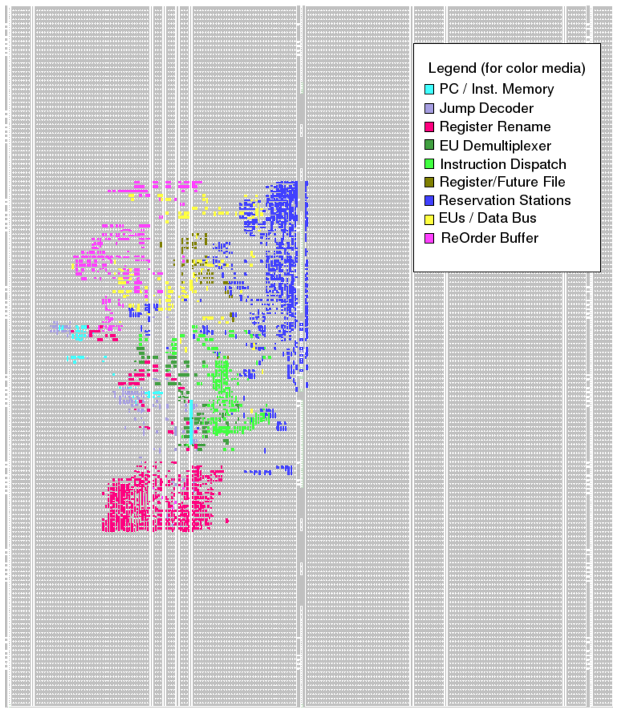


Figure 5.1: Unconstrained Area - The area used by the processor is a result of automatic placement by Xilinx's place and route tools.

Xilinx's synthesis tools shape the area of a particular design through user constraints. Our user constraints file (UCF) used `AREA_GROUP` constraints to fit our design within specific area boundaries for the targeted device. `AREA_GROUP` constraints are used for partitioning the design into different regions for mapping, packing, placement and routing [103], but we used them to affect the location and maximum area of our processor as a whole. By placing suitable area constraints during synthesis, all processor configurations occupy the same amount of fabric space, thus making reconfiguration of the system possible. This approach closely follows modern trends that seek performance increases through multiprocessor architectures running multi-threaded applications, with each thread executing in different reconfigurable locations.

According to current reconfiguration techniques, dynamic partial reconfiguration requires a convoluted manual implementation [104, 105, 106]. The requirements for this implementation detailed by Mermoud in [105] involve:

- *Applying area constraints for each module to be implemented*: The module areas must follow some strict guidelines:

  1. They must have a four-column minimum width.
  2. Their width is always a multiple of four columns (e.g. 4, 8, 12, ...).
  3. They are always the full height of the device.
  4. The boundary between two modules is placed on an even column (e.g. C19-C20).
  5. The area groups defined in the .ucf file are in the .ucf file with MODE=RECONFIG.

- *The floorplanning of all IOBs*: Each IOBs has to be wholly contained within the columnar space of their associated reconfigurable module.

- *The floorplanning of all global logic*: There must be no unconstrained top-level logic.

- *Constraining bus macros position*: LOC constraints are inserted for each bus macro into the .ucf file. Location of the bus macro is in the boundary between the modules forming the communication bridge. Each bus macro will occupy a 1-row by 8-column section of TBUF site space.

- *Check for pseudo logic*: Pseudo logic, created when a net connects one module to another is strictly forbidden in dynamic partial reconfiguration.

Following these requirements, we created the .ucf file in Code 5.1. The constraints declare everything within a design to be part of the `AREA_GROUP` AG_Implementation, and then set the area for this group to encompass all the slices from (column 0, row 0) to (column 51, row 159), hence using 160 rows (Y axis) and 52 columns (X axis).

Code 5.1: Fixing the area to 52-column width and full height of device

```
# Start of Constraints extracted by Floorplanner from the Design
INST "/*/" AREA_GROUP = "AG_Implementation" ; # This encompasses the whole design
#Y ranges from 0 to 159, full height of device
AREA_GROUP "AG_Implementation" RANGE = SLICE_X0Y0:SLICE_X51Y159 ;
AREA_GROUP "AG_Implementation" MODE  = RECONFIG;
```

Since the height of the XC5VLX220 is 160 rows, we fulfill the full height constraint, and the 52 columns fulfill the four-slice column boundary in an even column requirements. After applying these constraints to the processor in Figure 5.1, the placing and routing of the processor elements constraint the area to the left hand side of the device, as shown in Figure 5.2. The area in the middle of the FPGA indicates that some of the inputs and outputs were routed in non-columnar IOBs. To solve this situation, one would manually add more constraints for each input/output bit of each different processor configuration (and location).

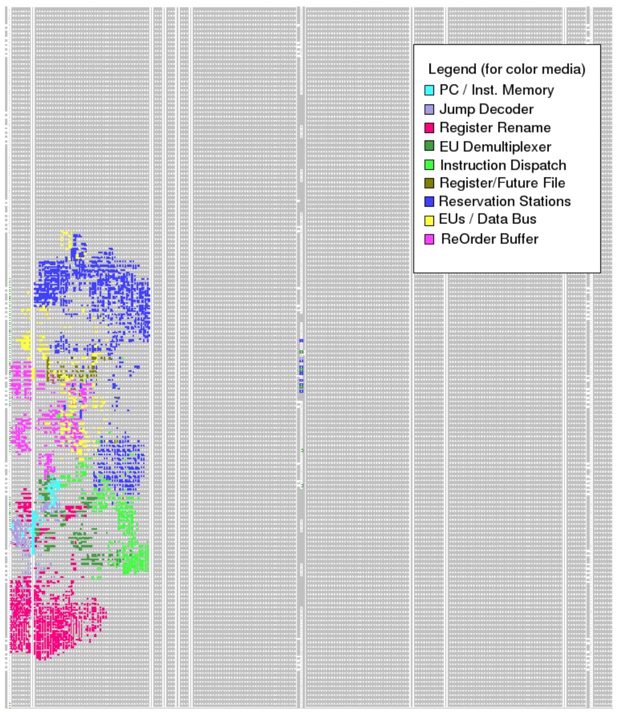


Figure 5.2: Constrained Area 1 - Processor with area constraints in the first 60 columns of the device.

The area in this processor was constrained to 160 rows $\times$ 52 columns, starting at row 0, column 0. However, we can also force a different location for the processor with the area remaining constant. We used Code 5.2 to force the location of the processor to the adjacent slot to the right of the original location. The result of this placement is seen in Figure 5.3.

Code 5.2: Fixing the location to start at column 52

```
# Start of Constraints extracted by Floorplanner from the Design
INST "/*/" AREA_GROUP = "AG_Implementation" ; # This encompasses the whole design
#Y ranges from 0 to 159, full height of device
AREA_GROUP "AG_Implementation" RANGE = SLICE_X52Y0:SLICE_X103Y159 ;
AREA_GROUP "AG_Implementation" MODE  = RECONFIG;
```

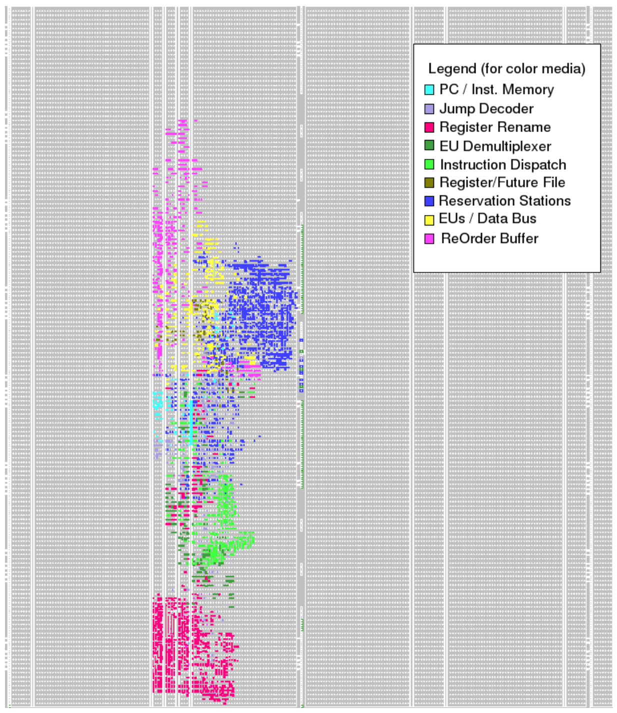


Figure 5.3: Constrained Area 2 - Processor with area constraints in the second 60 columns of the device.

To emulate a system which exploits TLP, we need multiple processors. Presumably, each processor would adopt a computational thread to run, and there would be some central module, which received the results from thread computations. A sample system would implement this functionality directly, synthesizing different processor configurations as needed. Figure 5.4 shows the implementation of a homogeneous double processor system. No area constraints are used, thus each processor is automatically placed and routed by the synthesis tools.

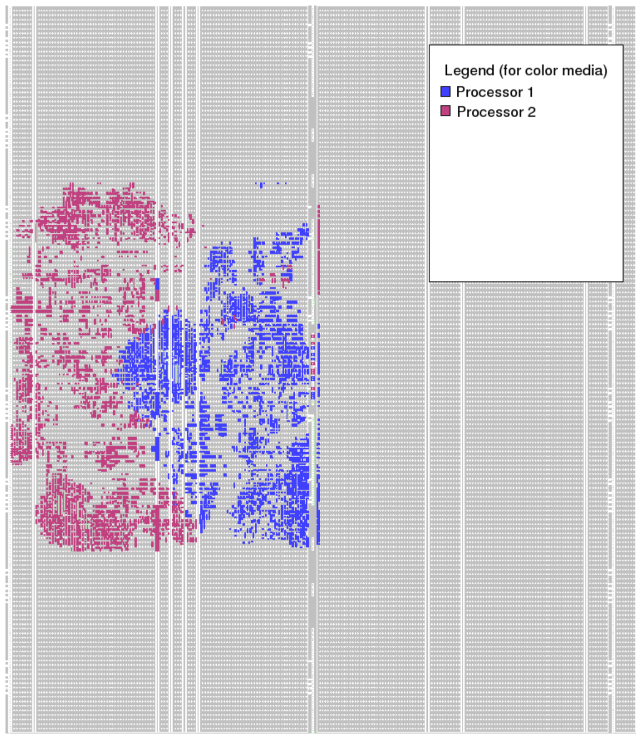


Figure 5.4: Dual-processor system - Implementation of a double processor system.

This implementation is adequate, and shows the FPGA's flexibility to implement different system configurations, depending on the design requirements. However, this setup does not capitalize in the potential for partial reconfiguration. Instead of synthesizing different configurations for every new design, one can create the central static module and make the different processors into dynamic modules. Hence, different computational threads can be mapped into the dynamic modules, execute their tasks, and be replaced by active threads afterwards through partial reconfiguration. This system setup requires us to once again constrain the area of dynamic modules to static boundaries and locations. By adhering to these requirements, processors can be inserted, removed or swapped in order to time-multiplex the resources of the FPGA, while letting other modules continue running simultaneously.

A sample UCF file with corresponding area constraints for the double processor system is shown in Code 5.3. Each processor occupies the same 160 rows $\times$ 52 columns area as before, but their left boundary locations start at column 0 and column 104.

Code 5.3: Double-processor User Constraints File

```
# Start of Constraints extracted by Floorplanner from the Design
INST "Processor_1_Instance/*" AREA_GROUP = "AG_Implementation_1";
INST "Processor_2_Instance/*" AREA_GROUP = "AG_Implementation_2";

 #Y ranges from 0 to 159, full height of device
AREA_GROUP "AG_Implementation_1" RANGE =    SLICE_X0Y0: SLICE_X51Y159;
AREA_GROUP "AG_Implementation_2" RANGE = SLICE_X104Y0:SLICE_X156Y159;
AREA GROUP "AG_Implementation_1" MODE=RECONFIG;
AREA GROUP "AG_Implementation_2" MODE=RECONFIG;
```

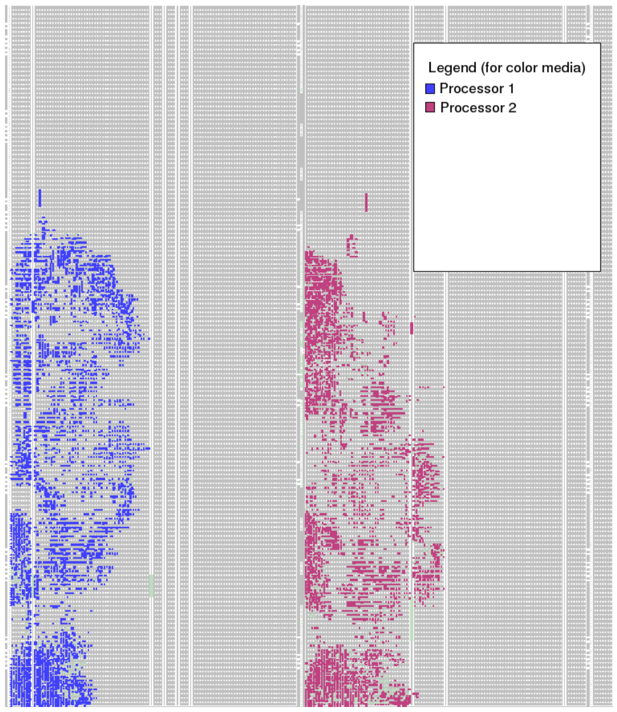


Figure 5.5: Dual-processor floorplanning - Each processor occupies the same area but is located in different sections of the FPGA.

We intentionally left a middle portion of the FPGA unused to reflect where a static module for the system could be placed. Figure 5.5 shows the new placement and area constraints for the processors, reflecting the dynamic setup explained previously. A designer would enable partial reconfiguration by swapping pre-synthesized processor configurations with the required constraints. The reconfiguration overhead would be reduced to a fraction of the full partial reconfiguration costs, namely the constraint area divided over the total FPGA area. However, our area constraints have been arbitrary in nature up to this point. With the UCF and partial reconfiguration techniques and requirements in place, we still need to evaluate how much performance is obtained per area unit of the FPGA, and which FPGA resources are being used.

5.1.3 FPGA resources

FPGA resources consumed by our design were measured by the total number of logic elements used. Logic elements comprise the majority of the FPGA real estate, and are implemented as a two dimensional mesh. For Xilinx devices, each logic cell contains a Look-Up Table (LUT) and a D-type flip-flop. FPGA slices include LUTs combined with multiplexers and some arithmetic logic like adders and carry-chains. Finally, a configurable logic block (CLB) is comprised of slices and routing resources. The combinations for logic cells, slices and CLBs have changed with the evolution of Xilinx's FPGA families. Table 5.1 shows the logic arrangement for Virtex II families and Table 5.2 for Virtex 5 families.

Virtex II Pro		
CLB	Slices	4
Slice	LUTs	2
	1-bit registers	2
	Multiplexers	2
	Adders	2
	Carry chain	1
Logic Cell	LUT inputs	4

Table 5.1: Virtex II-Pro components: CLBs, slices and logic cells.

Virtex 5		
CLB	Slices	2
Slice	LUTs	4
	1-bit registers	4
	Multiplexers	3
	Adders	2
	Carry chain	1
Logic Cell	LUT inputs	6

Table 5.2: Virtex 5 components: CLBs, slices and logic cells.

The CLB's LUTs, registers, adders and multiplexers are efficiently used by the synthesis tools to create storage, arithmetic and logic elements. Storage elements comprise RAM, ROM, registered memories and latched memories. Arithmetic elements include adders, counters and accumulators. Finally, logical elements include comparators, multiplexers and priority encoders.

Storage elements for a pipelined design usually involve multi-bit flip-flop registers, but the registers can also be configured to act as latches. Additionally, LUTs provide combinatorial bit storage in the form of Read-Only-Memory (ROM) logic truth tables. Outside the CLB scope, we can use the embedded Block RAM components for larger storage, like we did for the Instruction and Data memories of our processors.

Arithmetic elements in our design are mostly used by the execution units to calculate results and address offsets. Most of the counters and accumulators were used in the Register Rename stage for counting outstanding register read requests.

The logical elements were used extensively throughout each stage of the design. Due to the polymorphic nature of our processor, a carefully designed ISA bit-encoding scheme was not possible. Instead, we needed to use many comparators at every desicion point in the pipeline. These points included testing for jump and branch instructions, comparing source registers to previously renamed destination registers and comparing each broadcasted result in every reservation station. Multiplexers were used for appropriate input selection to outputs, especially during the Execution Unit Demultiplexer and Instruction Dispatch stages.

5.1.4 Area cost metrics

FPGA slices, which can be configured for storage, arithmetic and/or logic, were our unit of measurement for design area costs. However, the use of FPGA-specific elements like adders, comparators and multiplexers were also evaluated. Specifically, we collected the bit-level aggregates required for all of our processor configurations. All these were cross-tabulated with architecture configurations for different issue widths, execution units, register files, data sizes and memory interleaving. This spanned processors with `Instruction_Issue` $\in$ [1,4], `Number_of_EUs` $\in$ [1,5], `Register_File_Size` $\in \{08, 16, 32\}$ and `Data_Width` $\in \{8, 16, 32\}$ bits and `Interleaving` $\in \{1, 2, 4\}$ for a total of 540 different synthesized processor configurations.

While FPGA slices are a good resource measurement unit, it remains difficult to compare resource use among different FPGA families. We used CLBs, the smallest discrete configurable elements in the FPGA, for our device comparisons. Table 5.3 shows the CLB density for Virtex II and Table 5.4 for Virtex 5 devices. Stating that Virtex 5 has the same computational density of its predecessors just because of its registered bits would be an understatement. To be more objective, for the aggregate number of bits used for all 540 processor configurations, the average use of slices as

Virtex II Pro CLB density		
Slice	4-input LUTs	2
	1-bit registers	2
Slice density	LUT bits	32
	Register bits	2
CLB density	LUT bits	128
	Register bits	8

Table 5.3: Virtex II-Pro CLB density: LUT and register bits per CLB.

Virtex 5 CLB density		
Slice	6-input LUTs	4
	1-bit registers	4
Slice density	LUT bits	256
	Register bits	4
CLB density	LUT bits	512
	Register bits	8

Table 5.4: Virtex 5 CLB density: LUT and register bits per CLB.

LUTs was 72% while register slices use was 28%, with less than 2% variation. When we applied these percentages to the CLB density bit types, we obtained $0.72 \times (512/128) + 0.28 \times (8/8) = 3.16$ density increase per CLB. Our evaluation for area costs can thus be scaled up and down according to the device family.

Additionally, if the area cost were to be evaluated in terms of physical real estate, we would use the slice cost results for the appropriate family (or even other vendor FPGAs) and then normalize the area for the fabrication technology [19]. This fabrication technology λ, expressed in nanometers, reflects the average half-pitch of a memory cell. For Virtex 5 devices, $\lambda = 65$ nm, while for Virtex 4 devices and earlier $\lambda = 90$ nm. For real state calculation, we would normalize each device by λ^2. The physical density increase for Virtex 5 devices, normalized by (65nm^2), compared to Virtex 4 devices normalized by (90nm^2), is then 6.058 density increase per CLB.

With our area cost assessment, we are able to scale our FPGA resource-use to an appropriate device. Depending on the necessary area constraints for partial reconfiguration, designers can now chose the appropriate processor complexity to maximize their system performance.

5.2 Performance gains

The main dependent variables for our configurations were f_{clk} (the system's clock frequency) and reconfigurable fabric area. The clock frequency has an important defining role for overall performance of the superscalar processor, and hence the synthesis tool was set with an optimization goal for speed and a high optimization effort to maximize f_{clk} and minimize delay.

5.2.1 Clock frequency limitations

During the design process, we identified the Register Rename and Instruction Dispatch pipeline stages as the performance bottlenecks for our processors. These stages were trimmed as much as possible to minimize their delay. However, the complex processes of solving instruction dependencies and arbitrating register port requests are still the main bottlenecks of the system.

Register renaming is dependent on `Instruction_Issue`, and as stated before, its processing complexity increases quadratically with larger issue widths. However, larger issue widths increase the chances of finding ILP and executing multiple instructions simultaneously. On the other hand, the Instruction Dispatch stage is dependent on the `Number_of_EUs` parameter, where multiple instructions heading to each individual EU request read ports from the Future/Register file in the next stage.

5.2.1.1 Register Rename delay

To aleviate some of the Register Rename complexity and associated delay, a designer is able synthesize a system with a smaller `Register_File_Size`. This reduces the number of entries to examine when searching for reclaimable registers during the register renaming process.

Like we described earlier (section 4.3.2.4), a large `Register_File_Size` creates a large renaming pool. This pool prevents hazards from lack of reclaimable registers, but creates a large delay. A small `Register_File_Size` improves the f_{clk}, at the expense of added clock cycles spent waiting for reclaimable registers from a small renaming pool. In general, our superscalar processor will extract most of the ILP from the instruction stream, and the decision for an appropriate `Register_File_Size` is mostly based on maximum area requirements.

5.2.1.2 Instruction Dispatch delay

An unexpected finding was the effect that the Register and Future File parameter `Interleaving` had on the Instruction Dispatch stage (described in section 4.3.2.6). To service requests, this stage assigned each instruction with its necessary read ports. But our processor alleviated this need by broadcasting results, thus safely ignored port requests for *in-flight* values. This reduced need for ports proved beneficial, and reduced the amount of memory `Interleaving` needed, making the logic for port assignment easier, smaller and faster. Table 5.5 shows the amount of instructions *in-flight* and *in-registers* for a program running matrix multiplications in Hephaestus processors with `Interleaving` = 4. The *in-flight* instructions comprise most of the instructions, immensely relaxing the port requirements.

Interleaving 4	
In-flight	72%
In-register	28%
Cycles	282
R.Buffer	4x4

Table 5.5: LUT and register bits per CLB.

Interleaving 2	
In-flight	72%
In-register	28%
Cycles	286
R.Buffer	2x8

Table 5.6: LUT and register bits per CLB.

Interleaving 1	
In-flight	73%
In-register	27%
Cycles	305
R.Buffer	1x20

Table 5.7: LUT and register bits per CLB.

Table 5.6 shows minimal costs for slashing the number of available register ports in half, with only 4 clock cycles (1.4%) of timing penalty. On average, for 180 processors with varying issue widths and number of execution units, a reduction of the `Interleaving` parameter to 2 yielded an 8.9% increase in f_{clk} and 10.2% reduction in area.

`Interleaving` can be further reduced to one. This means that the total number of read ports available is equal to the value of `Copies`. Even though reducing this value would reduce the Instruction Dispatch delay and area considerably, it will also put an upper limit to the number of instructions able to be written to the Register File by the ReOrder Buffer. Table 5.7 shows an 8.2% clock cycle penalty for this setup. Even though no hazards were found during the Dispatch Stage, there were extra execution clock cycles needed at the ReOrder Buffer stage, slowing overall performance. Because only one instruction can be written per clock cycle when `Interleaving` = 1, the Reorder Buffer's number of logic cells has to be increased to accomodate a larger number of *in-flight* instructions waiting to be written. The total number of logic cells as described in section 4.3.2.11 is equal to `Interleaving` $\times$ `Reorder_Buffer_size`. This value was empirically found to be 16 for `Interleaving` = 4, 16 for `Interleaving` = 2 and 20 for `Interleaving` = 1 in the above example. The `Interleaving` parameter played an important role in our results for performance density, as smaller values can enhance run-time performance but at the cost of increased area.

5.2.2 Performance metrics

Standard performance metrics like instructions completed per second (IPS) or instructions per cycle (IPC) are not suitable for a computationally-intensive hardware based systems. In semi-custom hardware systems, performance is achieved by ensuring that execution units are actively performing computations or parts of computations. Further, the use of specialized and custom units diverge from the norm of single-operation instructions. Instead, to make maximum use of the available hardware, a single instruction could start a series of operations within a specialized execution unit, taking multiple clock cycles while keeping the execution unit busy. Consequently, it was not sensible to count all these operations as a single instruction for performance purposes.

We defined our basic unit of performance as the instruction execution rate. It is similar in nature to instructions per second, but also deals with the ambiguity of not knowing the throughput and latency composition of a semi-custom system's functional units a priori. To approximate the instruction execution rate, we used a value between the minimum and maximum of N_{issue} (the value of `Instruction_Issue`) and N_{eu} (the value of `Number_of_EUs`). The rationale for this was that neither the minimum nor maximum values for N_{issue} and the N_{eu} might reflect an accurate representation of the system. A system with a high N_{issue} and small N_{eu} might send a large number of instructions into the processor pipeline, but these instructions will stall while competing for the execution units. Likewise, a system with a small N_{issue} width but large N_{eu} will be underused due to the limited number of instructions being issued into the EUs. By consolidating these two parameters into a single value, we take into account the stalling from over-issuing and EU disuse from under-issuing, and also the added instruction parallelism from an increased number of execution units. Thus this value was an accurate approximation of the total number of instructions executing each clock cycle. This was an optimistic approximation for single-cycle instruction systems, but the possibility of multi-cycle custom instructions in a system makes this value more realistic.

The maximum clock frequency f_{clk}, which normalizes the processor implementations, reflects the performance costs of complex processor configurations. However, the performance gains are still not clearly defined. The reconfigurable execution units can handle any custom logic, from ALU combinatorial logic to complex subsystems involving sets of instructions. Assuming that the latter is the norm for hardware-based designs, we can conclude that the execution units will be heavily used, even with minimal instruction issue. Consider two cases of superscalar processor configurations with a single instruction issue and the functional units outlined in Table 5.8. The different functional units represent a typical setups for floating point operations. Even though a single instruction is issued per clock cycle (cc), the three floating point EUs can execute in parallel because of their extended execution latency, which is the time delay between starting and finishing a single instruction execution. Conversely, a large instruction-issue width aids in keeping instruction execution active in low-latency EUs. Table 5.9 shows a different system setup for integer computations, where multiple instruction issuing is needed to benefit from parallel execution in the system's low-latency EUs.

Table 5.8: System Setup 1

EU	Type	Latency
1	ALU	1 cc
2	FP Adder	6 cc
3	FP Multiplier	5 cc
4	FP Divider	15 cc

Table 5.9: System Setup 2

EU	Type	Latency
1	ALU	1 cc
2	INT Adder	1 cc
3	INT Multiplier	1 cc
4	Custom	3 cc

Although we have presented single-instruction EUs in our example, the number, types and latencies of the instructions contained within an EU can vary according to the custom logic that they implement.

The processor's instruction parallelism can be expressed in terms of the instruction-issue width and the number of EUs. The example using Table 5.8 shows that a system with a single instruction issue is capable of executing in parallel a number of instructions equal to N_{eu}, the number of EUs. Table 5.9, however, shows that N_{issue}, the instruction-issue width, can also set the number of instructions executing in parallel. We establish the lowest of these two values as the number of instructions that can be executed in parallel per clock cycle. Added parallelism resulting from the difference of the instruction-issue width and the number of EUs can be scaled by a coefficient α. The resulting value is normalized by the maximum clock frequency f_{clk} for the processor configuration, thus defining the Instruction Execution Rate (IER) for parallel execution within a reconfigurable processor framework:

$$\begin{aligned}\mathrm{IER} = f_{\mathrm{clk}} \times (&\mathrm{MIN}(N_{\mathrm{issue}}, N_{\mathrm{eu}}) + \\ &\alpha \times \Delta(N_{\mathrm{issue}}, N_{\mathrm{eu}}))\end{aligned} \tag{5.1}$$

This metric approximates the added performance from custom, high-latency and/or multi-cycle instructions, which are the main draw when migrating a system to specialized hardware.

5.3 Performance/area evaluation

The performance/area tradeoff was established in three ways. The first one is the instruction-execution density, the second one, bit-execution density, and the third one is the maximum instruction execution per area.

5.3.1 Instruction-execution density

The instruction-execution density defines the number of instructions that can be achieved per area unit, and is calculated for each configuration by dividing the IEPS over the synthesized processor area. The resulting value then indicates how many instructions are executed per second in each FPGA slice.

This approach shows some similarities to DeHon's computational density metric. In [19], DeHon suggests that an FPGA CLB roughly equates one ALU bit processor operation. He added that FPGAs gain computational density by executing multiple operations in parallel, while processors loose computational density due to added instruction overhead. However, this assessment of FPGA peformance is largely theoretical since it addresses raw computational power. Most non-trivial, instruction inter-dependent computations will require additional overhead for mapping an instruction stream into an FPGA board.

Our approach when calculating instruction execution density takes into account the instruction overhead and storage needs. This setup matches the requirements of traditional microprocessor systems. This makes it more accurate representation of the computational power per area unit within an FPGA.

5.3.2 Bit execution density

An advantage of reconfigurable computing is that it can match the data width requirements of a particular system. In the matrix multiplication example in section 4.4.2, the data results were all integers that could be represented in less than 5 bits. A 32-bit hard-processor executing such a program would have 85% of its available data bandwidth wasted in unnecessary bits. Even soft processors like MicroBlaze do not allow modifications to their 32-bit data path width. However, our processor modifies its data path width to ease the system's area requirements and increase performance.

The ability to fine-tune the data granularity is quite powerful. In terms of code size, Hennessy and Patterson showed that using 16-bit instruction width can produce up to 40% code size reduction when compared to 32-bit wide instruction words [79]. In terms of area, there will be a proportional reduction in use for all data storage locations and signals. Consequently, the reduced area increases the instruction-execution density of the system.

5.3.3 Maximum instruction execution

This method for analyzing performance and area tradeoffs relates directly to partial reconfiguration. Given a fixed area requirement, we can use our results to maximize the instruction execution within fixed space boundaries. The assesssment of which processor configuration is most suitable to maximize performance is made through performance/area graphs. When treating area as an independent variable, the graphs show which configurations fit within the area limitations, and of those configurations, which one yields the highest instruction-execution rate.

When using this method, it is important to follow partial reconfiguration schemes, like aligning dynamic module boundaries to 4-column increments and using the full height of the device [106].

With the evaluation for instruction-execution density, bit-execution density and maximum instruction execution, we can speculate the area requirements for a particular application and also define the performance gains per area unit. These evaluations greatly aid in defining the performance/area tradeoff within FPGAs when optimizing dynamic systems for coarse-grained parallelism.

5.4 Summary

The myriad of processor configurations that are possible with Hephaestus provide ample opportunities for design optimization and performance tweaking. To evaluate the costs associated with performance, we collected area and performance values for 540 different processor configurations over a range of superscalar system parameters. These results will be the central focus of the next chapter.

The synthesis of each processor showed its resource requirements in terms of FPGA low-level resources like slices, adders, multiplexers and comparators. We used slices as our main area cost metric, since they act as storage, arithmetic and/or logic. Each processor can be shaped through slotted synthesis into a fixed area, which allows partial reconfiguration.

The system's clock frequency degrades with increasing processor complexity, and is set by the longest delay in the pipeline's bottleneck stage (usually Instruction Dispatch or Register Rename). However, increased processor complexity allows a larger number of instructions to execute in parallel. To approximate multi-cycle instructions typically found in custom systems, we used the values for the instructions issued and available execution units. This was a conservative approximation for the number of instructions executing each clock cycle. When normalized by the clock frequency, we obtained the processor's instruction execution per second, used as our performance metric.

We combined the area and performance metrics to provide three methods for performance/area evaluation for FPGAs: instruction-execution density, bit-execution density, and maximum instruction execution. The later one can be directly used for maximizing performance in fixed area slots for partial reconfiguration, while the others enable a robust assessment of performance compromises in FPGAs. Chapter 6 presents the results of our area and performance evaluation.

Chapter 6

Results and Discussion

The previous chapter justified our instrumentation, area and performance metrics, and data collection methodology. With the evaluation framework in place, we then collected data on performance and resource use. Our results showed two important characteristics for each processor. The first one was what level of parallel execution performance could be achieved with a particular configuration, and the second one was the amount of resources needed to achieve this parallelism. Even though the individual characteristics of each processor configuration provided useful information, the aggregate values proved to be more valuable when examining general trends in the performance and area trade offs.

Performance mostly depended on the N_{issue} and N_{eu} parameters, which allowed increased issuing and parallel execution of instructions, respectively. The system parameters `Register_File_Size` and `Interleaving` make indirect contributions to the instruction execution rate, and their impact is analyzed in section 6.3.4. The final system parameter, `Data_Width`, also did not contribute to the instruction execution rate directly, but predictably, had a deep impact in area use and performance density.

6.1 Performance

6.1.1 Clock frequencies

As stated previously, semi-custom system performance was expressed as the instruction execution rate, IER. This instruction execution rate was calculated from the independent variables N_{issue} and N_{eu}, which conform the basic superscalar processor structure. The dependent variable f_{clk} expressed the logic and delay complexity of the synthesized circuit, imposing an upper limit to performance in terms of maximum pipeline speed.

After synthesizing all of the 540 processor configurations, we obtained every implementation's clock frequencies and calculated each IER. The processor's clock frequency mean value was 73.02 MHz with a standard deviation of 29.58 MHz, exposing the high clock-frequency variance in our system. This was expected, as the logic time delay for pipeline stages increased drastically in complex processors. This is in contrast with the hand-optimized static Microblaze processor pipeline from Xilinx, which has a top speed of 200 MHz on a Virtex-5 Xilinx FPGA.

Table 6.1 shows the clock frequencies f_{clk} for processors with a register file size of 16 registers, an instruction interleaving of 2, a 16-bit datapath width and a varying number of issue widths and execution units.

Table 6.1: Clock Frequencies (MHz) for processor subset.

Issue Width,	Execution Units, N_{eu}				
N_{issue}	1	2	3	4	5
1	163.6	126.3	82.2	60.7	48.8
2	89.8	94.6	90.2	63.6	51.7
3	65.4	66.5	64.5	61.6	49.7
4	55.3	54.4	54.2	54.8	51.5

Clock frequency tables for processors with different datapath widths, memory interleaving and register file sizes provide similar trends. Table 6.2 shows the averaging of f_{clk} over all twenty-seven different combinations of these three variables. It evaluates the change with respect to N_{issue} and N_{eu}, normalized over a scalar processor implementation ($N_{\text{issue}} = 1$ and $N_{\text{eu}} = 1$).

The clock frequency rapidly decreases with increased processor complexity. However, these clock frequencies results only expose the timing delay costs, not the parallel

Table 6.2: Normalized Clock Frequencies.

Issue Width,	Execution Units, N_{eu}				
N_{issue}	1	2	3	4	5
1	1.00	0.84	0.57	0.42	0.34
2	0.63	0.63	0.57	0.42	0.34
3	0.45	0.45	0.45	0.41	0.34
4	0.40	0.40	0.40	0.38	0.33

performance that can be attained by these systems.

6.1.2 Logic and routing delays

There are two types of timing delays that occur in our processor: logic and routing delay. Logic delays are the result of the propagation times needed for signals to travel from the inputs of logic gates to their output, and increases accordingly with larger amounts of logic levels. Routing takes into account interconnection delays among the different FPGA logic cells, and is a function of circuit size, fanout of the net and routing congestion [107]. It is important to discern the type of delay so that then they may be used for targeted optimization. Our superscalar processor framework did not provide much logic processing, as data processing was left for the customized executions units. However, it routed the instructions and data to their appropriate destinations and pipeline stages' outputs. On average, the logic delay was 21.3% and the routing delay was 78.7% of the total delay for all processor configurations. This coincides with current assessments that processing data is not difficult, but moving it is [108]. However, because our data is highly localized within the processor framework, our data routing cost is minimal when compared to multi-core, multi-threaded and multi-processor systems.

There is room for improvement in our processor by combining some pipeline stages. Figure 6.1 shows each pipeline stage's time delay, divided as logic and routing types, for a sample processor with $N_{\text{issue}} = 2$ and $N_{\text{eu}} = 2$. Three main bottleneck stages were identified: the Register Rename, Instruction Dispatch and ReOrder Buffer stages. Unfortunately, due to the linear nature of the operations of these components and the need for single clock cycle latency, further further pipelining is not possible.

The memory `Interleaving` for Figure 6.1 was set to two. Consequently, the Instruction Dispatch and ReOrder Buffer stages needed additional time for their rout-

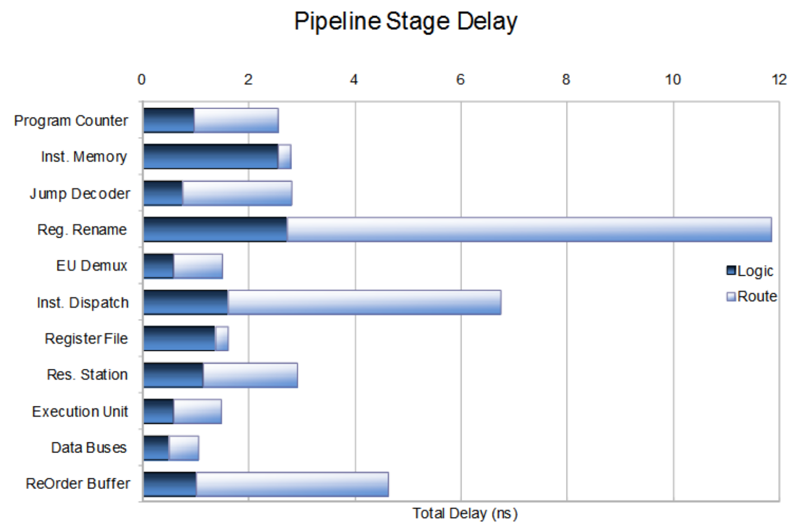


Figure 6.1: Pipeline Stage Delay - Total delay in each of the eleven pipeline stages, by type.

ing computations. However, this time was not as lengthy as the 11.8 ns required for register renaming for register renaming. This high value is explained by the large `Register_File_Size` value of 32 registers.

While the combination of memory interleaving and the register file size played fine-tuning roles in subsequent pipeline delay, the three main pipeline bottleneck stages remain fairly constant throughout all configurations. Hence, it is easy to see how in the future we could combine some of the adjacent pipeline stages like the Register File and the Reservation Stations, or the Execution Units the Common Data Buses. This merging would potentially reduce the area needed for intermediate registers between the stages, while decreasing the latency of instructions traveling through the processor.

The main culprits driving the system's time delay were still the N_{issue} and N_{eu} variables, which set the an appropriate processor configuration for a particular application. Figure 6.2 shows the logic delay result after averaging together all processor configurations with respect to N_{issue} and N_{eu}. For simple configurations, N_{issue} was the main contributor for logic delay due to register renaming.

However, as more and more execution units are added, the largest delay shifted to the Instruction Dispatch stage, where a larger amount of read port requests have to be arbitrated. Eventually, the delay caused by the increased number of execution units dominated the overall system delay, adding 0.72ns of logic delay for each extra EU. This was also true for routing delay, but with a higher delay cost. Figure 6.3 shows a more

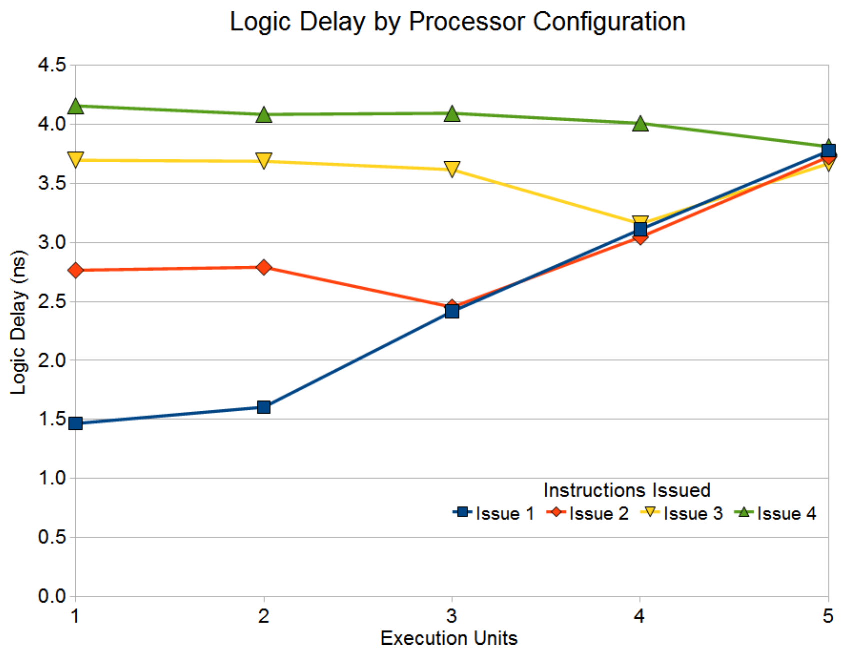


Figure 6.2: Logic Delay by EU - Logic delay increased linearly with respect to the number of execution units.

pronounced 3.29ns route delay increase for each additional EU.

The linear nature of these logic and routing delay results showed that the costs of time delay were predictable. Consequently, the clock frequency f_{clk} of a system could also be predicted depending on the number of execution units present, and considering that each extra EU will add 4.01ns of total delay to the system. Of course, the negative effects of extra execution units were counter-balanced by the positive increase in the instruction execution rate from additional opportunities for parallel execution.

6.1.3 Instruction Execution Rate

The extent to which the potential benefits of added hardware benefited performance were gauged by the independent parameter α. We reiterate the equation for IER, the Instruction Execution Rate (previously shown in Section 5.2.2), for clarity:

$$\begin{aligned} \text{IER} = f_{\text{clk}} \times (\text{MIN}(N_{\text{issue}}, N_{\text{eu}}) + \\ \alpha \times \Delta(N_{\text{issue}}, N_{\text{eu}})) \end{aligned} \tag{6.1}$$

We used Equation 6.1 to calculate the instruction execution rate for the five hundred and forty synthesized permutations of processor configurations. The processors' IER

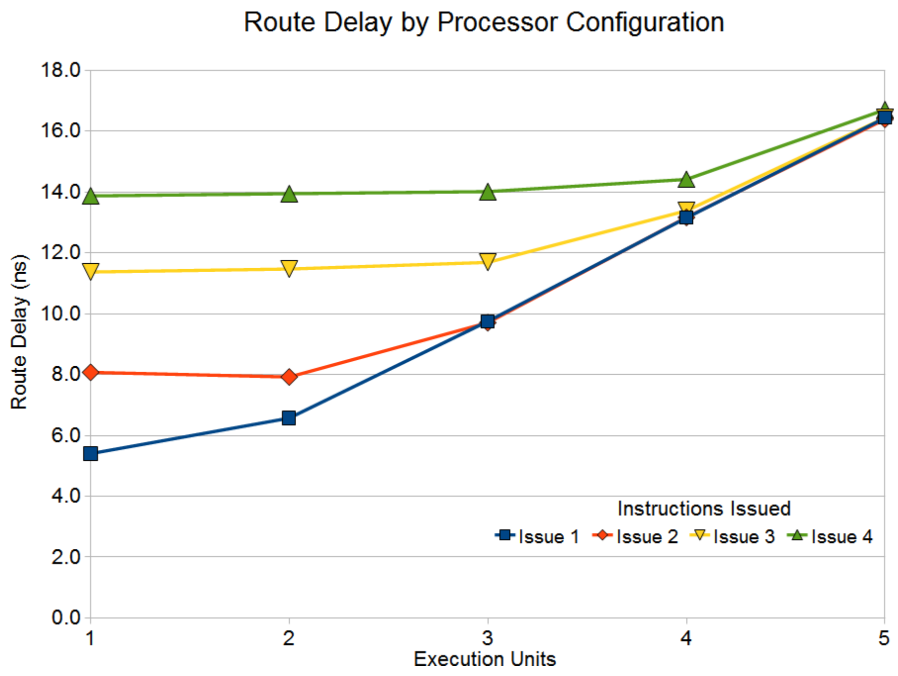


Figure 6.3: Route Delay by EU - Route delay increased linearly with respect to the number of execution units.

values were averaged and grouped together by the number of instructions issued to determine the effect of the coefficient α in the final execution rate values. Figure 6.4 shows this effect, with α ranging from 0 to 1. The IER generally tends to increase as more instructions are issued. When $\alpha = 0.0$, the minimum of the instruction-issue width and the number of EUs is used to scale the clock frequency. At $\alpha = 1.0$, the IER clock frequency is scaled using the maximum of the two values. $\alpha = 0.5$ corresponds to scaling the resulting processor's clock frequency by the average of the two values.

The same process was used to group the processors according to the number of execution units they implemented. Figure 6.5 shows the effect of adding more EUs to a system (also with a variable coefficient α). The decrease in IER is caused by the bottleneck in the instruction-dispatch stage, which must solve contention for the register file's memory ports for instructions arriving at their respective execution unit. Therefore, all operands for each EU must be examined sequentially, creating a long delay and consequently degrading the clock frequency and ultimately the processor's IER.

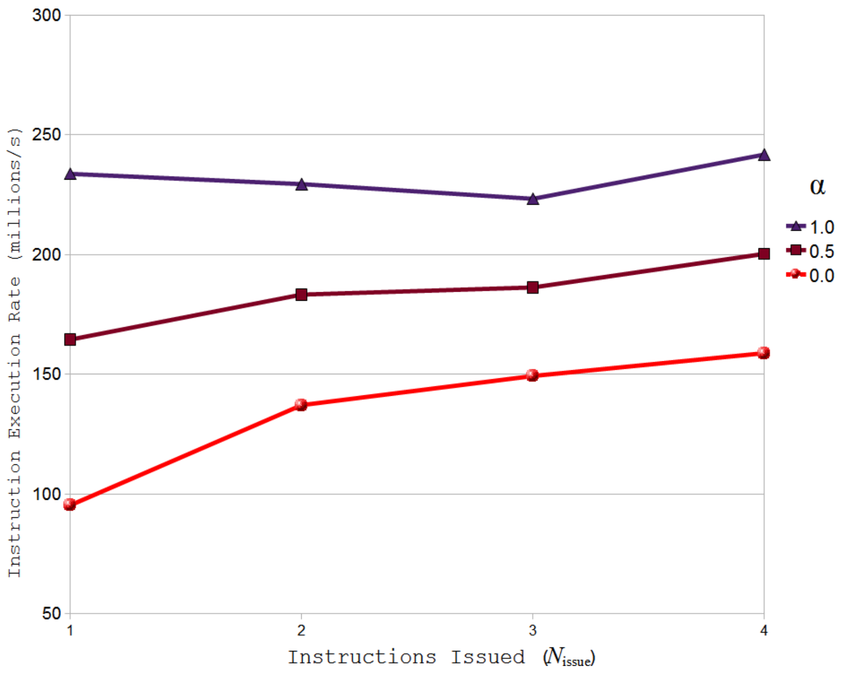


Figure 6.4: Instruction Issue IER - IER, grouped by instruction-issue width.

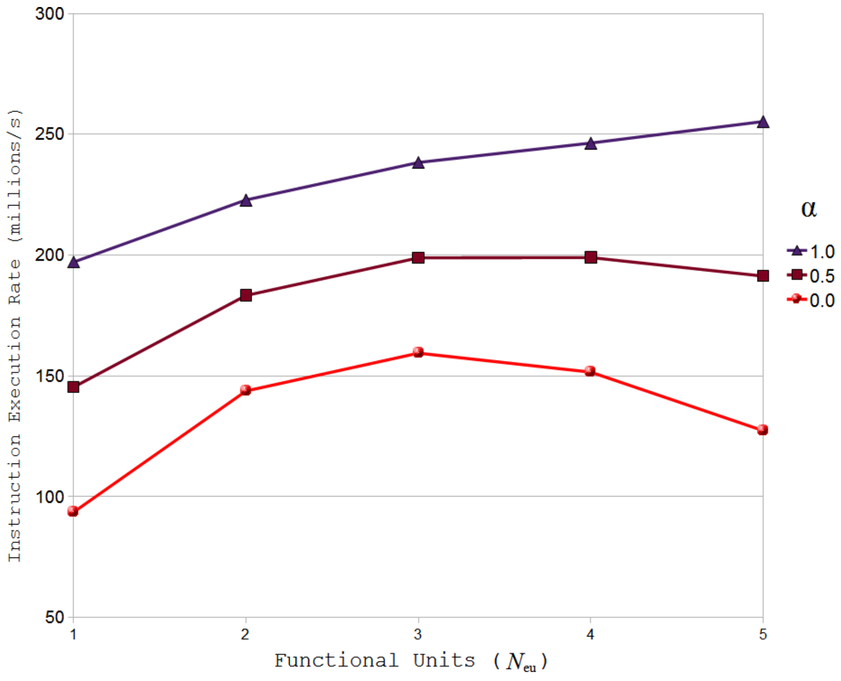


Figure 6.5: Functional Units IER - Multiple levels of IER, grouped by execution units.

We restricted our focus for the value of $\alpha = 0.5$ for IER calculations, which corresponds to an adequate level of added parallelism, without pessimistic or optimistic assumptions on the processor configuration. The IER equation was applied to the same system presented previously in Table 6.1, a processor architecture with a 16-bit datapath width and register file size of 16 registers. The resulting IER, shown in Table 6.3, shows the potential instruction execution rates for different processor configurations that enable parallel execution.

Table 6.3: Maximum IER (millions/second)

Issue Width,	Execution Units, N_{eu}				
N_{issue}	1	2	3	4	5
1	163.6	189.5	164.5	151.6	146.3
2	134.7	189.3	225.6	190.9	181.0
3	130.8	166.1	193.6	215.5	198.7
4	138.3	163.2	189.6	219.2	231.8

At the aggregate level, the analysis of all processor configurations shows that issuing an additional instruction increased the IER by 7.3%, and including additional execution units increased it by 7.9%. Note that these values do not necessarily represent application speedup, but the number of expected instructions executing in parallel within the reconfigurable processor architecture. With these values the best processor setup to implement a custom system can be chosen.

The instruction execution rate increased as more instructions were issued and more execution units were available, even with slower clock frequencies resulting from complex synthesis. The IER aggregate for all the synthesized processors was 182.0 (millions of instructions/second) with a standard deviation of 47.7M/s. The point of diminishing returns was reached after adding a fifth execution unit, when the system's clock frequency degradation outweighs parallel gains.

6.1.4 Processor configuration performance

We grouped together all processors with the same instruction execution configurations, regardless of register file size, datapath width and memory interleaving. This resulted in twenty-seven processor configurations sharing the same instruction-issue width and execution units, for a total of twenty different groups.

Figure 6.6 shows the aggregate instruction execution rate for these different groups, arranged by instruction issue and execution units. We see that for a constant (color-coded) execution unit, the largest IER was attained when N_{issue} closely matched the N_{eu} in the system. However, for systems with two or fewer execution units, the clock degradation resulting from issuing multiple instructions also decreased the instruction execution rate.

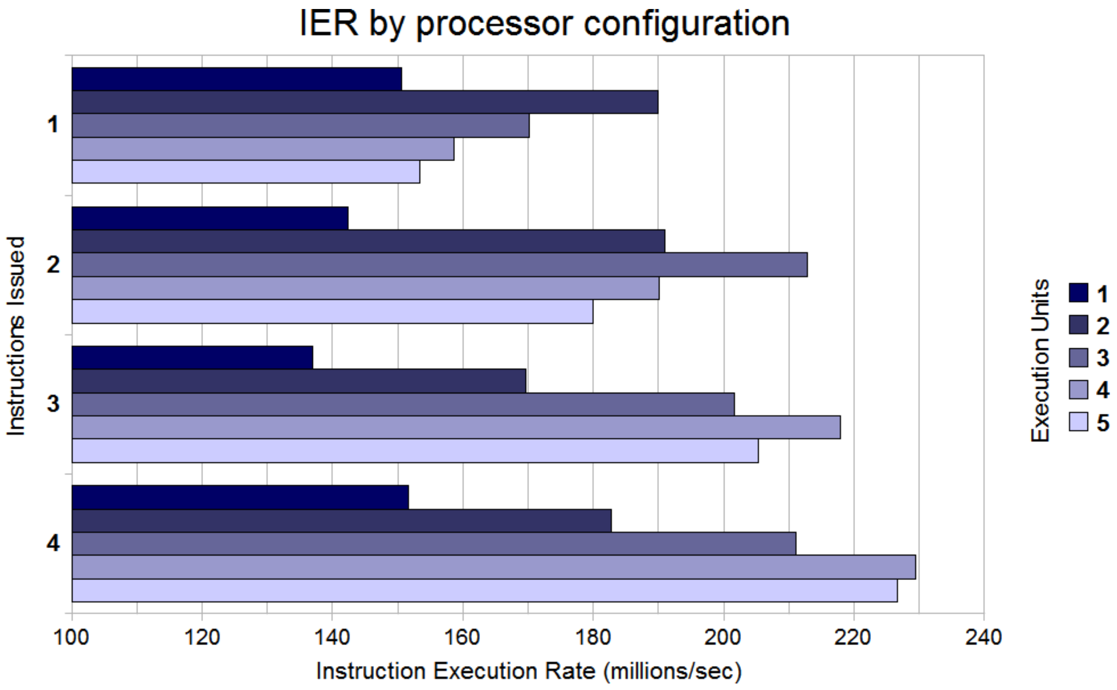


Figure 6.6: Figure: Instruction Execution Rate - Aggregate IER by instruction-issue width and execution units.

On average, the IER went up by 7.0% when increasing the instruction-issue width by one, with the largest increase (11.5%) happening when increasing the instructions issued simultaneously to two. When adding execution units, the IER average increase was 7.6%, with a decrease of −4.2% occurring when adding a fifth execution unit.

Overall, our configuration flexibility introduced a hybrid solution that implements custom logic inside a general-purpose processor. By tailoring our processor to a particular application setup, we increased the IER for the system. The configurations with the largest amounts of IER occurred when N_{eu} was slightly larger than N_{issue}. When this happened, the logic and route delays introduced into two different pipeline stages (which each depend on one parameter exclusively) closely matched each other. As a result, there was no bottleneck pipeline stage, but instead we had a system with a more uniform delay per pipeline stage.

The results for the IER outline the potential benefits that Hephaestus provides to an application by means of increased instruction execution rate [109]. A Hephaestus system is more flexible than vendor soft-core processors, and achieves increased parallel execution for superscalar configurations while enabling custom logic in its datapath without disrupting it.

Even though we provide the framework for increased parallel execution, application speedup is still reliant on custom logic provided by the system designer. Floating point operators are suitable candidates for execution units, since they contain mostly logic and little routing, as opposed to our processor framework were the routing is a limiting factor. A sample synthesis for a six-stage pipelined floating point adder shows that the delay for the critical path (29 logic levels) is comprised of 76.5% logic and 23.5% routing delay (3.08ns logic, 0.95ns route), allowing for a clock frequency of 248.3 MHz. This of course, is so fast when compared to our the processor's clock frequencies, that it borders on waste. A seasoned designer might take unorthodox approaches to fully benefit from the underlying reconfigurable architecture. For example, we can use asynchronous floating point operators for FPGAs to implicitly scale down the number of pipeline stages a floating point operator uses [110]. Another way is to use parameterizable floating point operators that allow us to explicitly set the number of pipeline stages, like with the FloPoCo HDL floating point library [111]. Hence, if our system clock is slow, our execution units can pack more logic in each of their pipeline stages, reducing clock cycle latency. Designers can use the same logic-packing approach can be used in custom execution units.

6.2 Area

The area was evaluated through the synthesis of all configured processors into a XC5VLX220 Virtex 5 device. This FPGA contains 17,280 CLBs, equivalent to 34,560 slices. Each Virtex 5 FPGA slice has four LookUp Tables (LUT) and four single-bit registers. These slice resources provide a clear indication of the minimum necessary area requirements to implement the design's logical functions and storage.

6.2.1 Costs of Parallelism

Area costs were evaluated by the number of slices used. Post-synthesis results provided us with the minimum necessary registers and LUTs as two separate values. However, a Virtex5 slice accommodates four LUTs and four registers, making it necessary to merge these two values for an accurate slice count. The maximum count value for LUTs or registers dictates the minimum number of slice resources; in our design, LUTs were consistently more numerous. Thus we divided the maximum LUT count by four to obtain the minimum slice count.

The synthesis slice requirements were modified to fit the actual implementation requirements. A factor of two was factored in to our calculations due to congestion routing and interconnect resources during place-and-route [108]. Moreover, it is too optimistic to assume that all CLBs will be used to their maximum potential, even with high optimization efforts by the synthesis tools. Thus, we further relaxed the place-and-route efforts by assuming that on average only 75% of the slices within each CLB are used. Area slice use after the place-and-route implementation proved this synthesis approximation method to provide a fairly accurate final area slice count. Thus our slice area utilization was calculated as follows:

$$\begin{aligned}\text{Slice count} = {} & \text{MAX (Slice LUTs, Slice Registers) Synthesized Slices}\\ & \times 2 \text{ Slices / Synthesized Slices}\\ & \times 1 \text{ CLB / 4 Slices}\\ & \times 75\% \text{ Slices / CLB} \qquad (6.2)\end{aligned}$$

6.2.2 Total area

Using Equation 6.2 we evaluated the FPGA slice area costs for the synthesized processors by grouping them by their processor configurations (N_{issue} and N_{eu}). We used aggregate results from our synthesized processors to quantify the necessary area for increased issuing and execution units. Table 6.4 shows the average slice utilization for all processor configurations:

Table 6.4: FPGA Slice Use

Issue Width,	Execution Units, N_{eu}				
N_{issue}	1	2	3	4	5
1	2389	3374	4393	5565	6599
2	3017	4049	5147	6231	7273
3	3870	4918	6037	7185	8312
4	5338	6431	7625	8757	9908

These same results are depicted in graphical form in Figure 6.7. We see that the area increases dramatically but predictably with complex processor configurations.

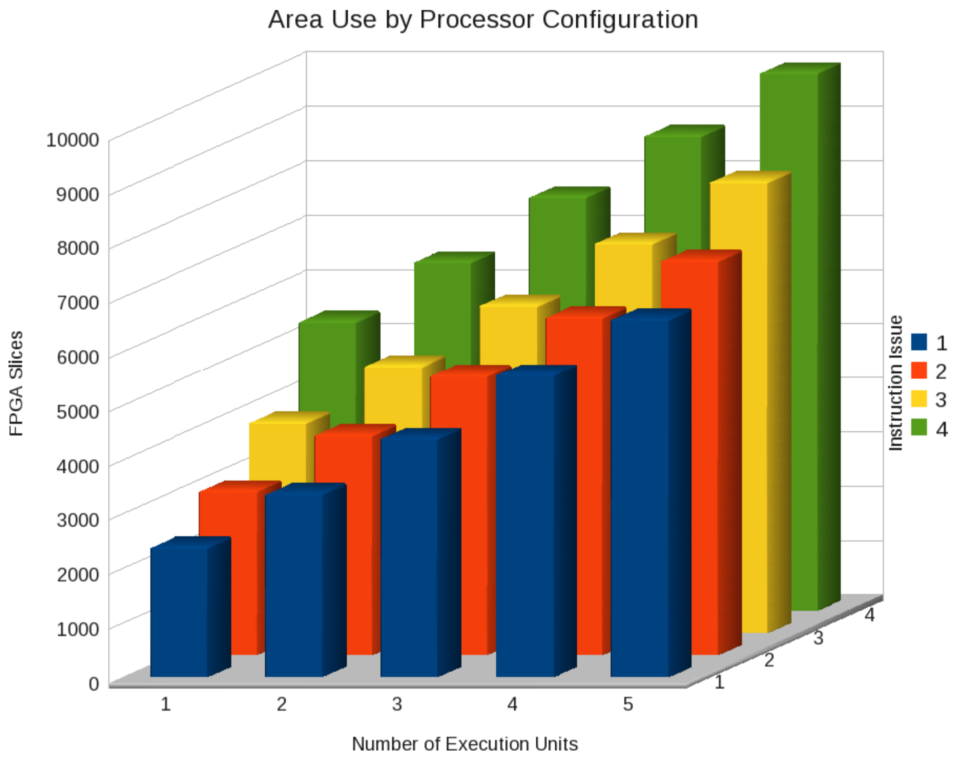


Figure 6.7: Total Slice Count - Total slice count by processor configuration.

The area increase was separately assessed by its N_{issue} and N_{eu} variables, to observe the percent area increase from a scalar configuration. Figure 6.8 shows the linear increase in area with respect to N_{eu}.

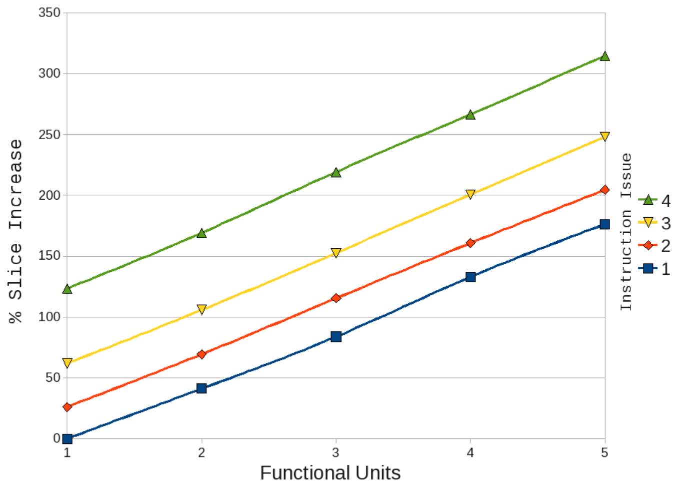


Figure 6.8: Area Increase per Execution Unit - Slice increase from a scalar system.

Increase of the instruction-issue width N_{issue} caused a quadratic effect on area overhead [13]. However, the issue-width overhead further caused a quadratic reduction in the clock frequency f_{clk} (also expressed as a quadratic delay increase). The area increased linearly with respect to the number of reconfigurable execution units N_{eu}, with an average 48.2% for each additional EU. Figure 6.9 shows the effect (again represented as area percent increase from a scalar system) when increasing the instruction-issue width for processors with different EUs.

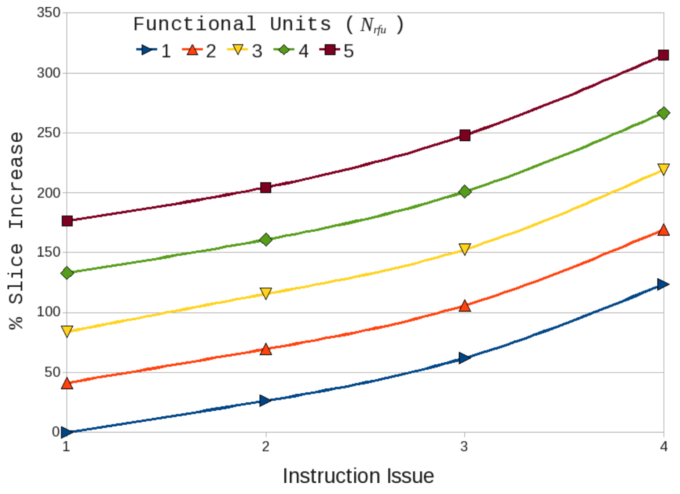


Figure 6.9: Area Increase per Instruction Issue - Slice increase from a scalar system.

The overall area-percentage increases from a scalar system are shown in Table 6.5, and summarize the effect of adding parallel execution opportunities to the slice area cost.

Table 6.5: Percent Area Increase by Processor Architecture

Issue Width,	Execution Units, N_{eu}				
N_{issue}	1	2	3	4	5
1	0%	41%	84%	133%	176%
2	26%	69%	115%	161%	204%
3	62%	106%	153%	201%	248%
4	123%	169%	219%	267%	315%

The slice count increased an average of 23.5% with each increment of N_{issue}, but this value does not represent the quadratic area increase. The total values for the FPGA slice count for the different systems in terms of N_{issue} and N_{eu} reflect the polynomial nature of area expenses. For example, a system with 16 registers, 16-bit datapath width and memory interleaving of two was curve-fitted to find the polynomial which predicted its area in slices, obtaining the following area equation:

$$\begin{aligned}\text{Slice count} = 314 &+ (1185 \times N_{\text{eu}}) \\ &- (68 \times N_{\text{issue}}) \\ &+ (172 \times N_{\text{issue}}^2)\end{aligned} \tag{6.3}$$

The number of slices per increase of N_{eu} (1185 slices) correspond to the area requirements of an ALU, a reservation station with three entries, and the combinatorial logic to receive instruction result broadcasts. This value will change depending on the type of custom EU implemented by a designer.

Area also depends on the system parameters `Register_File_Size`, `Interleaving` and `Data_Width`. By generalizing Equation 6.3 we accounted for the effect each different combination had on the slice values:

$$\begin{aligned}\text{Slice count} = \alpha &+ (\beta \times N_{\text{eu}}) \\ &+ (\gamma \times N_{\text{issue}}) \\ &+ (\delta \times N_{\text{issue}}^2)\end{aligned} \tag{6.4}$$

The variables α, β, γ and δ were then evaluated for all combinations of N_{reg} (the value for `Register_File_Size`), $N_{memport}$ (the value for memory `Interleaving`) and N_{data} (the value of `Data_Width`). Table 6.6 shows all the parameters to get the area for any possible processor configuration.

Table 6.6: Area Equation Values by System Parameters

N_{data}	N_{reg}	$N_{memport}$	α	β	γ	δ
8	8	1	967	375	-262	183
8	8	2	658	629	64	104
8	8	4	963	847	31	116
8	16	1	584	566	19	165
8	16	2	500	833	114	161
8	16	4	875	1033	94	169
8	32	1	1279	576	-243	362
8	32	2	994	919	-21	317
8	32	4	1550	1075	46	305
16	8	1	1841	460	-376	208
16	8	2	816	990	-229	171
16	8	4	1256	1072	48	120
16	16	1	849	809	76	157
16	16	2	314	1185	68	172
16	16	4	1425	1161	198	113
16	32	1	1471	829	-152	349
16	32	2	1159	1115	12	311
16	32	4	1930	1308	-8	316
32	8	1	996	796	-255	186
32	8	2	1448	1314	-154	157
32	8	4	1631	1633	-53	136
32	16	1	551	1002	49	158
32	16	2	914	1606	-66	195
32	16	4	1547	1810	9	184
32	32	1	848	1080	-205	353
32	32	2	1606	1541	-61	334
32	32	4	2393	1821	-65	323

This reconfigurable area assessment is particular to the Virtex 5 FPGA family. However, the area requirements may be scaled appropriately to other devices, using slice information among FPGAs with different families and transistor sizing [112].

Equation 6.4 can be used by a system designer in conjunction with Table 6.6 to predict the total number of slices in a system configuration, with all the required system parameters. This is an important consideration when area is a limited resource due to partial reconfiguration or multiprocessor system designs.

6.2.3 Area Constraints

Partially reconfigurable and multi-processor systems must use area constraints in their coarse grain components to benefit from parallelism. Reducing the synthesized area allows allocation of additional components. FPGA area-allocation algorithms may be used to achieve optimal reconfigurable component placement in dynamic systems [113], once the task area is known.

To determine the area of a particular processor configuration, we used Equation 6.4 to calculate the necessary slices. For example, in a system with $N_{data} = 16$, $N_{reg} = 16$, and $N_{memport} = 2$, we can use the simplified Equation 6.3 to calculate the slice count for a processor configuration of $N_{\text{issue}} = 2$ and $N_{\text{eu}} = 2$. The calculated slice count is 3926 slices. Without any area restrictions, the actual slice use can be higher, but using constraints increases the place-and-route effort, thus reducing the number of slices.

Xilinx's partial reconfiguration techniques specify that area boundaries must be located at 4-column increments using the full height of the device. The XC5VLX220 device has 160 rows and 108 CLB columns, and being a Virtex 5 FPGA, has 2 slices per CLB. This means the area increases by 1280 slices every four CLB-columns. At the 12 CLB column boundary, the slot area is 3840 slices, which is *not* enough to map our predicted 3926-slice superscalar processor. Figure 6.10 shows multiple physical FPGA implementations of this processor, with varying degrees of area constraints.

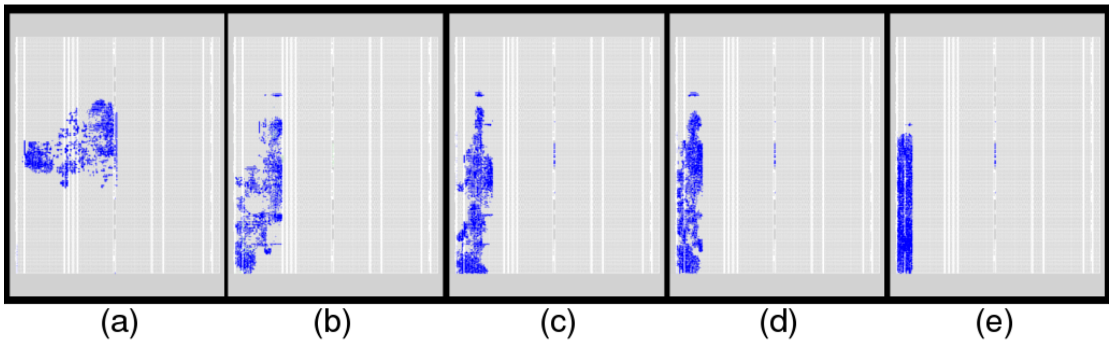


Figure 6.10: Floorplanning - Physical implementation in (a) all columns (b) 52 columns (c) 40 columns (d) 28 columns (e) 16 columns.

Attempts to place-and-route the processor in 12 CLB-columns failed from lack of space, as our calculations predicted. These calculations allowed us to predict the number of CLB columns needed to implement a reconfigurable processor architecture before implementation, cross verifying our analytical model. However, there are situations in which the area is fixed, and the processor configuration can vary. Section 6.3.1 shows the parallel performance evaluation of a processor that fits in a limited area.

We can determine the area use for a multi-processor system by following the same guidelines we outlined earlier. Each instantiated processor's area can be predetermined by using Equation 6.4 with the desired parameters, given the same or different target configurations to perform the tasks of that particular processor. Figure 6.11 shows a Symmetric Multiprocessor System (SMP) implementation with seven processors configured identically. For visual clarity, each processor was constrained to a separate area slot in the FPGAs.

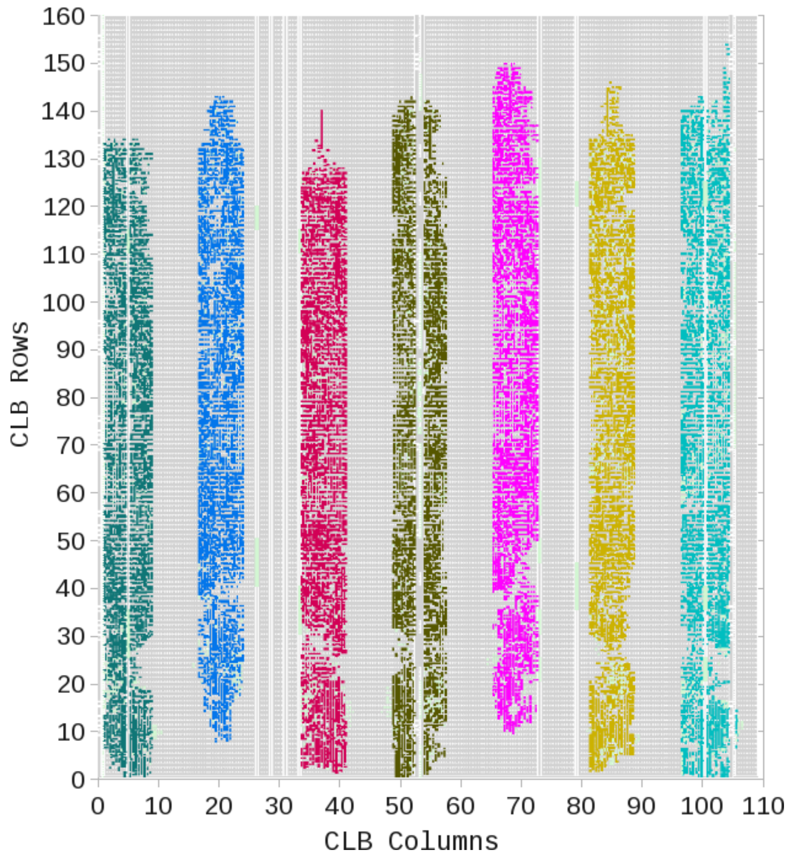


Figure 6.11: Seven processors 16 columns - SMP system with seven processors.

6.2.4 Area Use Reduction

Datapath minimization is not offered in commercial FPGA soft processors, even though FPGAs provide bit-level granularity. Our extra level of adaptability for custom designs helped reduce the area for smaller designs. The reduction comes as a the result of trimming data bits, usually the unnecessary highest significance bits for low-precision arithmetic. In addition, our processor aids in fine-tuning the datapath width to match the requirements of a particular application while decreasing the use of FPGA resources.

The reduction in area was demonstrated by synthesizing a processor's datapath at bit-level resolution up to the standard 32 bits. The selected processor configuration had two instructions issued simultaneously and two execution units. Its register file size was sixteen registers, and memory interleaving was set to two memory banks. The resulting IER was 189.5M/s, close to our aggregate average of 182.0M/s encompassing all configurations. In summary, we used an average processor implementation representative of the rest of the configuration permutations.

As the datapath width of the processor was reduced bit by bit, so did the area that it consumed in the FPGA. Predictably, the percent reduction in slice utilization was a linear function inversely proportional to the number of datapath bits, as shown in Figure 6.12.

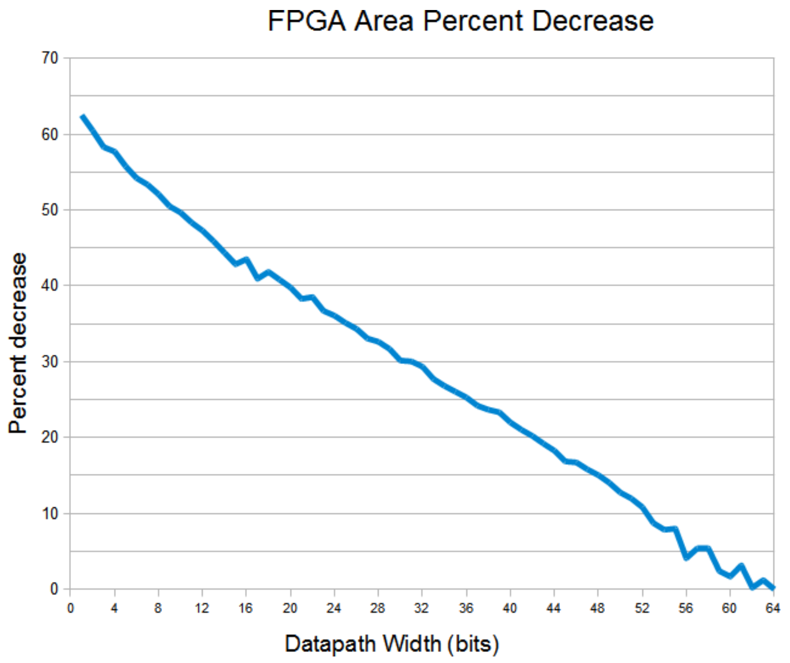


Figure 6.12: Datapath Area Use - Area reduction, normalized over 64-bit datapath implementation.

The processor's IER was not greatly affected (± 4%), but we achieved a reduction in area of up to 63% for single-bit computations. A less trivial datapath of 8 bits reduced the area by 52%. The architecture research group at Hewlett Packard demonstrated how an average reduction of 49% in total gate count can be achieved through datapath optimization for integer operations [114]. Their analysis however, was applicable to non-programmable hardware accelerators. For our programmable semi-custom processors, we obtained a 29% average FPGA-slice reduction for datapath width optimization for a range of 1 to 64 bits.

6.2.5 Synthesis time

With less bits and area to map onto the FPGA, the synthesis tools took less time to implement the design. This is particularly useful when judging the best processor configuration for a specific application, since reduced datapath widths also result in reduced prototyping times.

Time reduction also predictably followed a linear relationship to the datapath width. Over a range of 1 to 64 bits, average synthesis time savings averaged 46%. Figure 6.13 shows the time percent reduction for variable datapath width in single-bit granularity, normalized over a 64-bit implementation, with a maximum of 81% reduction in synthesis time for a single data bit and 73% reduction for an 8-bit implementation.

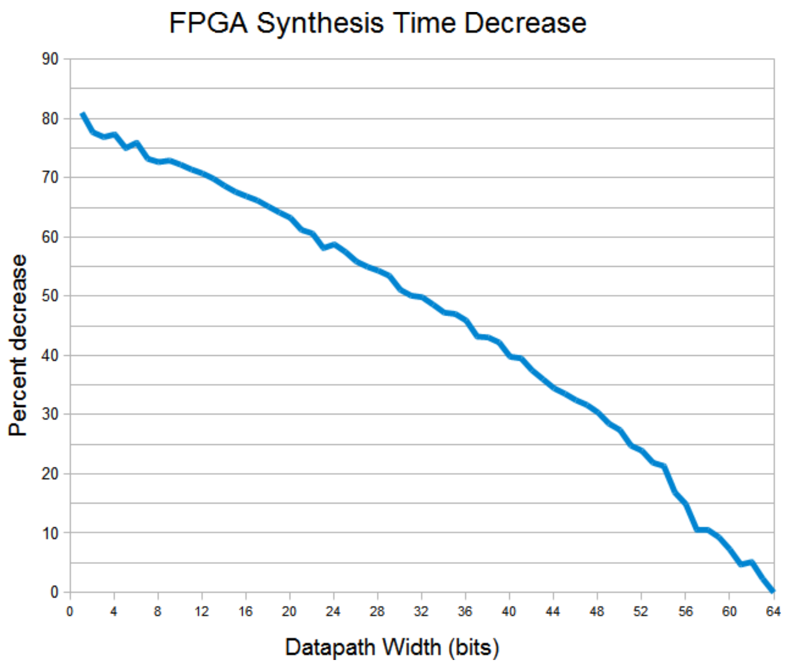


Figure 6.13: Datapath Synthesis Time - Synthesis time reduction, normalized over 64-bit datapath implementation.

6.3 Performance/Area trade offs

So far, we have evaluated the performance and area cost for processor configurations independently from each other. However, the two are tightly coupled and must be evaluated together. For example, designers planning to incorporate multiple processors for parallel execution within a single FPGA must often follow area specifications that enable them to do this. Partially reconfigurable areas must follow strict guidelines, and multiprocessor systems must balance the number of processors with the capacity of the FPGA. Additionally, parallel performance needs to be maximized. While the computational density research by DeHon gives us the maximum theoretical performance that can be achieved in a fully custom FPGA solution [19], we focused on the achievable parallelism within the confines of reconfigurable processor architectures.

6.3.1 Maximum execution rate

Because we can alter the processor architecture by specifying the parameters N_{issue} and N_{eu}, we can optimize the instruction execution rate in constrained areas, like those for partial reconfiguration or multi-processor systems. Figure 6.14 shows the IER for different processor configurations, encompassing all the permutations of N_{issue} and N_{eu}.

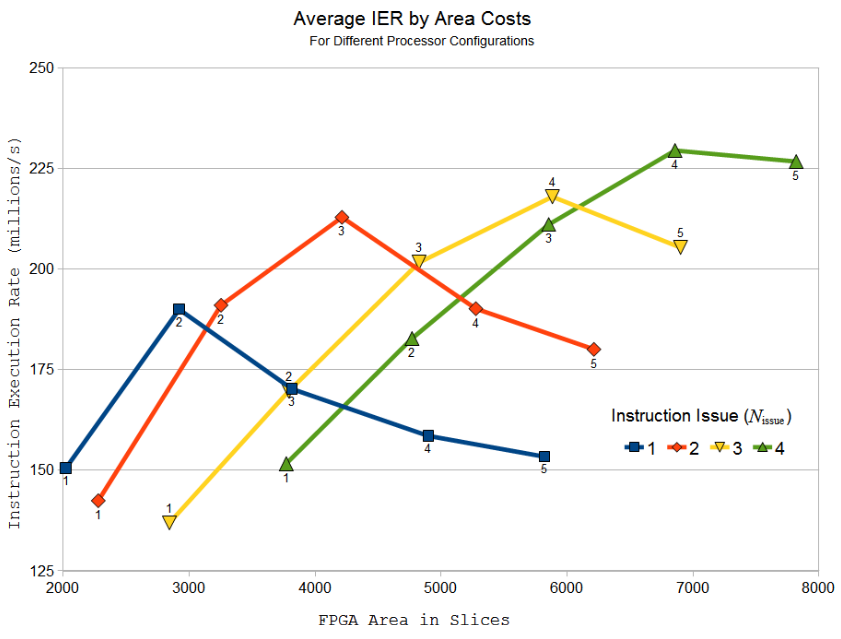


Figure 6.14: IER by Area Costs - IER and area use by processor structure. Data labels represent execution units.

The figure's horizontal axis shows the area cost with units of FPGA slices, while the vertical axis indicates the corresponding IER performance. The numbers under each data point correspond to N_{eu} for that instruction-issue width. We can see that each curve shows a maximum execution rate that then declines under the weight of increased logic and routing delay from added execution units. For systems with larger instruction-issue widths, the maximum IER becomes asymptotically larger with the IER maximum at 232M/s. The FPGA area is increased when increasing either N_{issue} and N_{eu} as described in the previous section.

It must be noted that Figure 6.14 represents values grouped by N_{issue} and N_{eu}, with the IER and slice area averaged with regard to N_{data}, N_{reg}, and $N_{memport}$ variables. Section 6.2.4 showed the area reduction from smaller datapaths, and section 6.3.4 will show how the area used is also affected by the latter two variables. However, the curve shapes for the system's IER remain fairly constant, but scaled vertically and horizontally. Figure 6.15 also shows the IER area costs, but averaged only for 32-bit datapath width systems.

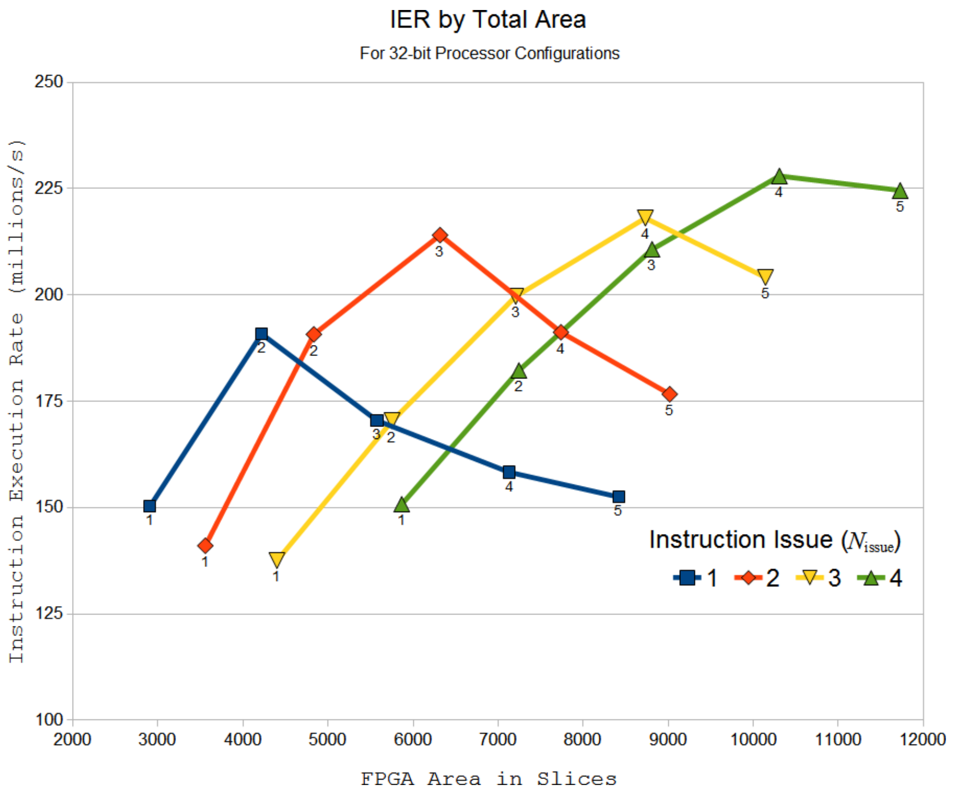


Figure 6.15: IER by Area Costs - 32 bits - IER and area use by processor structures with 32-bit datapaths. Data labels represent execution units.

The area costs for the resulting IER are greatly increased due to the wider datapath of 32 bits. This area reduction can have a deep impact in the IER density, which is the amount parallel execution obtained per area unit. Pragmatically, rather than using the IER density information to fit area constraints, a system designer would instead use the information depicted in IER/area graphs to best match their system.

We can choose the highest level of parallelism from IER/area graphs by selecting the processor configuration with the maximum IER and with a smaller number of slices than the targeted area size. For example, an $N_{\text{issue}} = 1$, $N_{\text{eu}} = 2$, $N_{data} = 32$, $N_{reg} = 16$, and $N_{memport} = 2$ partially reconfigurable system might require to use an extra EU ($N_{\text{eu}} = 3$) for added functionality. Using Equation 6.4 and Table 6.6 results in a calculated area of 5861 slices for this system. To follow partial reconfiguration methods, this system needs at least 20 CLB-columns, which encompass a total of 6400 slices, shown in Figure 6.16 as a grey box. The $N_{\text{issue}} = 1$, $N_{\text{eu}} = 3$ system is labeled as “Minimal Configuration”.

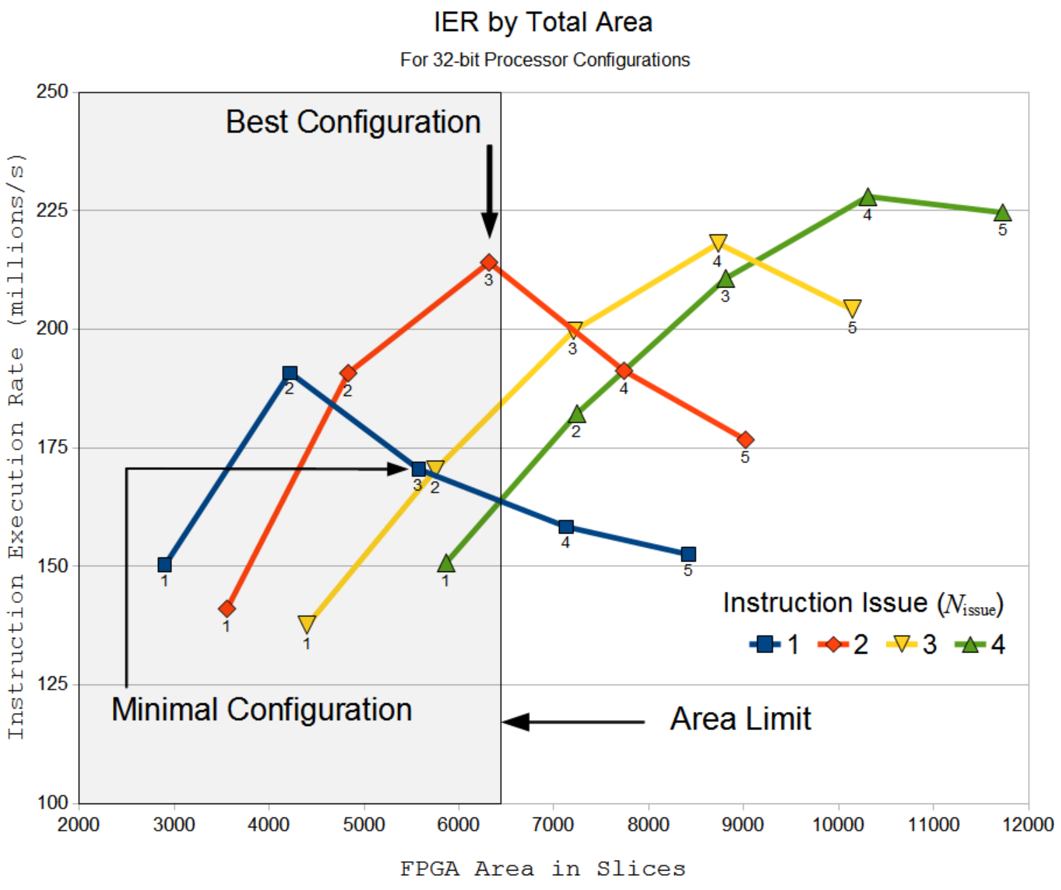


Figure 6.16: Best IER Configuration - 32 bits - Optimal IER in Limited Area. Data labels represent execution units.

Figure 6.16 also shows that a larger amount of parallelism can be achieved within the confines of that area by increasing to instruction issue to $N_{\text{issue}} = 2$, labeled as "Best Configuration" . The instruction execution rate is consequently increased from 170M/s to 214M/s with the new processor configuration, while still fitting the required execution units in the same confined area of 6400 FPGA slices.

Our method of assessing performance/area trade offs thus provides a two-pronged analysis of the maximum instruction execution rate. Not only does it answer the question of how much area is needed for a target parallel execution rate, but also how much parallelism can be obtained for a confined area [115]. These two questions can be merged by analyzing the results for the instruction execution density in FPGAs.

6.3.2 Instruction execution rate density

We have seen that different instruction execution rates can be achieved with different processor combinations taking different amounts of area. A more generic analysis of performance/area trade offs would be to use the instruction execution rate density. By itself, it specifies the individual contribution to parallel execution that is obtained for each additional area unit.

The aggregate value for the IER density is 34,630 instructions/second/slice. This means that each individual FPGA slice added to any particular configuration augments the parallel execution rate by 34,630 instructions per second in a reconfigurable superscalar processor system. However, the IER density varies per processor configuration, so a linear approach for assessing average contribution per slice is not appropriate.

The processor's instruction execution rate value can not be simply increased by adding individual slices, but instead is done by changing processor configurations. These processor configurations also have an effect on IER density. Figure 6.17 shows the IER density for processor with different instruction-issue widths, and Figure 6.18 shows the same density with respect to execution units.

Both figures show that the parallel performance per area in larger processors decays as the processors get more complex. Thus, even though the processors use a larger amount of area to achieve higher performance, the efficiency of this additional area quickly diminishes from the initial value of 63,000 instructions/second/slice. Moreover, it asymptotically approaches a density value of 20,000 instructions/second/slice as more

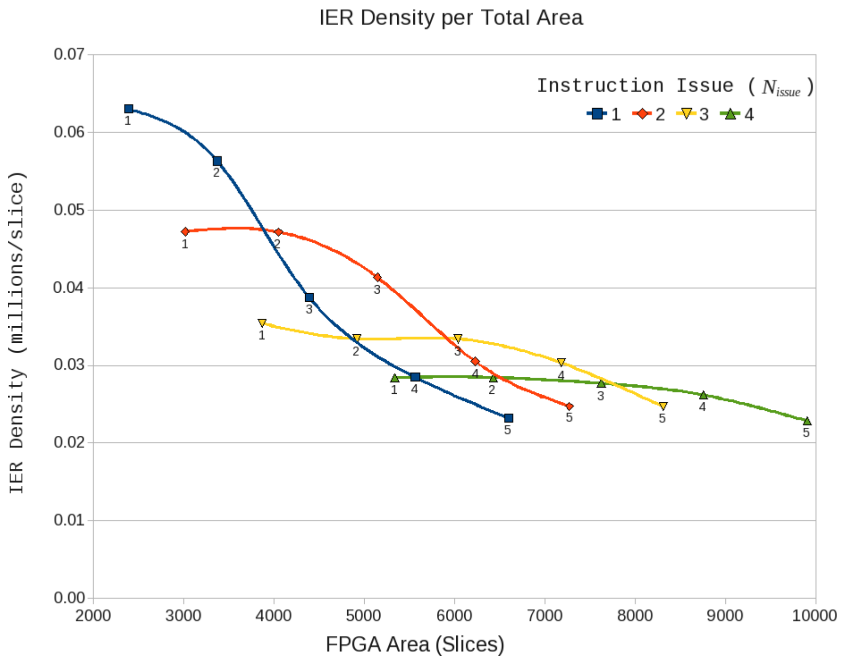


Figure 6.17: IER Density by Instruction Issue - IER performance per slice over total area used, categorized by instruction issue. Data labels represent execution units

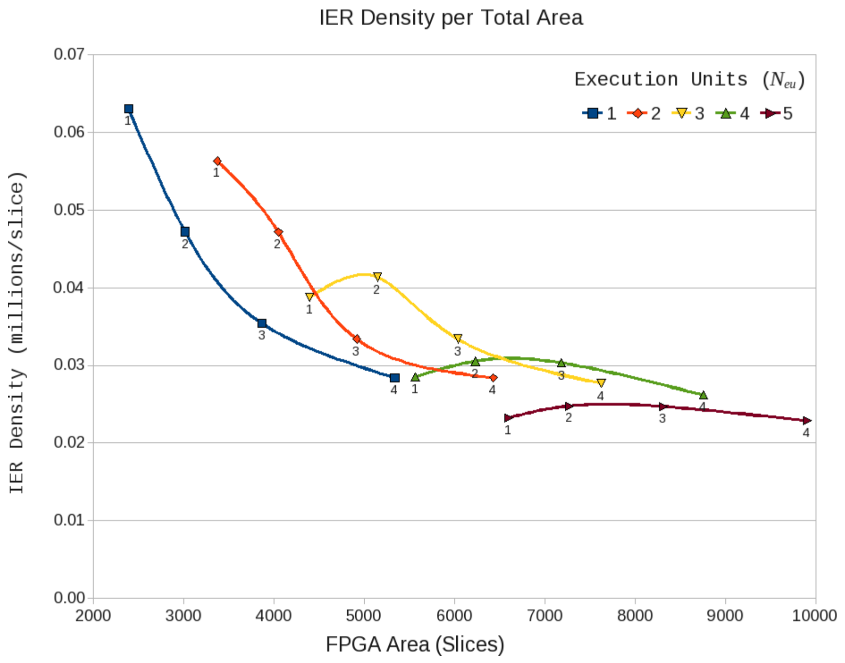


Figure 6.18: IER Density by Execution Units - IER performance per slice over total area used, categorized by execution units. Data labels represent instructions issued

area is used. This non-zero contribution to parallel execution identifies the minimal performance/area trade off for FPGA-based processors with reconfigurable architecture. By contrast, the performance execution density of a MicroBlaze processor implemented on the same Virtex-5 system was a constant 102,828 instructions/second/slice. The 1.6 increase in performance density against our maximum performance density can be attributed to the hand-optimization of MicroBlaze processors combined with their lack of architecture customization, which results in a more efficient structure. Our processor allows not only for architecture customization, but also datapath customization. Because the area changes abruptly with the datapath width, it is also convenient to express this performance density in terms of the particular number of data bits that are being executed per instruction.

6.3.3 Bit execution density

Our processor's ability to reduce area through datapath minimization increased the Instruction Execution Rate density in the FPGA. Since our performance unit, IER, and the processor framework are based on the number of instructions executed, reducing the size of the executed data increases the density of execution within this area. Thus, the opposite is also true: the larger the datapath size needed, the smaller the execution density per area, as Figure 6.19 depicts.

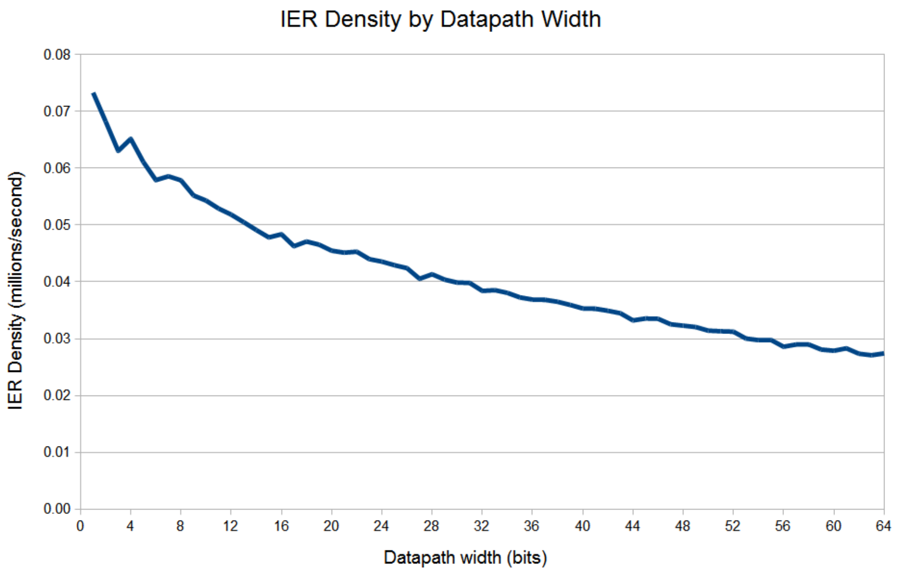


Figure 6.19: IER Density per Data Bit - IER performance per slice per data bit shows better instruction execution density for smaller datapaths.

Though the results are not necessarily encouraging for larger datapath widths, they only correspond to the number of instructions being executed, not the number of bits being executed upon. A more data-centric approach is to instead use the bit-density of execution. Figure 6.20 shows the rate at which bits are being executed per second per slice, over a wide range of datapaths.

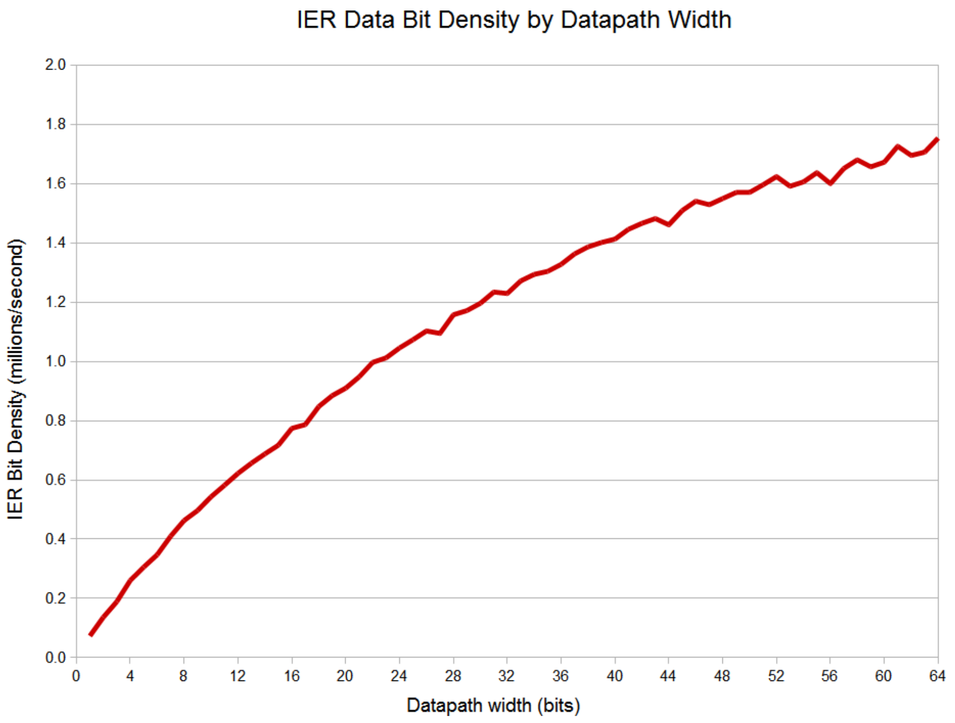


Figure 6.20: IER Data Bit-Density by Datapath Width - Bit-level IER density increases rapidly for larger datapaths, but decreases as more register area is needed.

We can see that the larger the datapath width, the more number of bits that are executed per area unit. This occurrs because for small datapaths, the data storage area is small compared to the instruction storage and architecture area overhead. As the datapath length increases, the necessary storage area for data bits also increases, reducing the bit-level IER density. Eventually, the data area becomes large enough to no longer have a positive effect in density, since the datapath's required area overshadows the bit-level performance increase.

6.3.4 Secondary Independent Variables

We have mostly focused on N_{issue} and N_{eu}, but the other independent system variables also have a profound effect on performance. N_{reg}, the value for `Register_File_Size`, dominates the logic bottleneck during register renaming, and $N_{memport}$, the value for the memory `Interleaving`, influences the bottleneck during the dispatching of memory ports. These two variables do not directly contribute to the number of instructions that can execute in parallel, that is, execution in space. Rather, they contribute to reducing impediments when executing in time.

The evaluation of time/space computations depends on the specific application. Therefore, a system designer must evaluate if increasing or decreasing the sizes for N_{reg} and $N_{memport}$ will be beneficial or detrimental. Large sizes prevent choke points while small ones increase speeds. For the memory interleaving, a low $N_{memport}$ value would not affect an application with high levels of instruction-level parallelism, as most of the dependent values will be in-flight and would subsequently be broadcast to waiting instructions. On the other hand, applications that have high data-level parallelism will benefit from extra memory access ports for independent data.

The register file size has the largest impact on performance besides the instruction issue and number of execution units. A small register file can be used for control-based applications with low amounts of register writes, while a larger register file benefits more scientific computations. Figure 6.21 shows the performance and area trade offs for different register file sizes, with instruction execution rate on the vertical axis and area on the horizontal one.

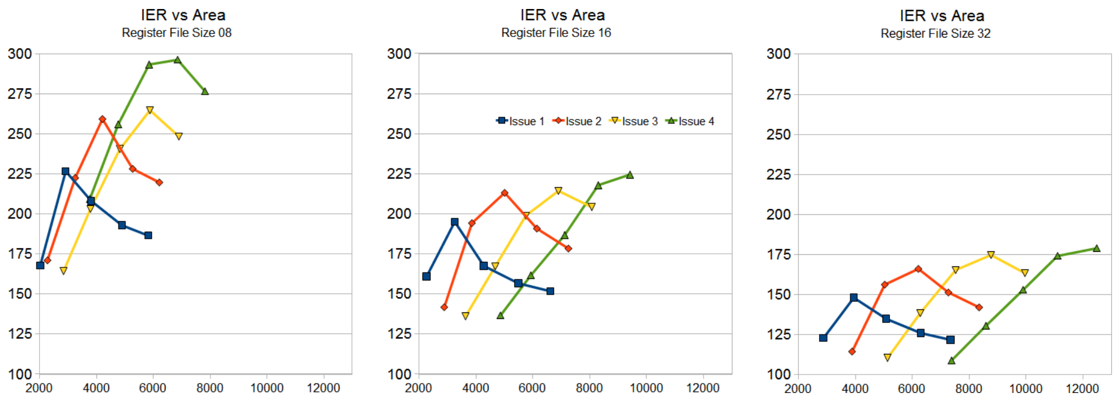


Figure 6.21: IER/Area per Register File Size - Small register files show a better performance and area use.

It is evident that larger register files increase area use and decrease the IER. However, the run-time performance of an application can suffer immensely due to the stall when there are no available free registers to do register renaming, so small is not necessarily better. Fortunately, our flexible processor allows the seamless modification of register file size to better suit performance or area concerns: a small register file can be selected for systems with space constraints, while a larger one enhances execution of applications with time constraints.

6.4 Summary

The results of synthesis techniques for performance evaluation in reconfigurable processors showed that superscalar processor configurations excel in parallel execution opportunities by increasing total performance on complex processor configurations. The cost for these opportunities are decreasing clock frequencies for complex processors, as the logic and routing delay start to dominate the performance. However, this speed decay occurs at a rate which still allows benefits from parallel execution of instructions. To evaluate these benefits, we presented a parallel performance metric, the instruction execution rate, which takes into account the advantages of added execution units and number of instructions issued while normalizing them over the resulting clock frequencies. The processor configurations that can be selected to maximize the instruction execution rate for particular applications. The maximum parallel execution rates happened when the instructions issued closely matched the number of execution units. In these instances, the delay differential of the bottleneck pipeline stages for register renaming and instruction dispatching was minimal.

The synthesis optimization efforts were focused on speed and not area, meaning the costs of parallelism in our system were subsidized by a larger FPGA area use. The metric for this area use were slices, which conveniently pack LookUp Tables (LUTs), registers, and arithmetic functions. An empirical formula was derived from synthesis and place and route area use, involving routing and interconnect resources, sub-par logic mapping, FPGA family and processor configuration variables. Area increased linearly with respect to the number of execution units, and quadratically with respect to the number of instructions issued. The shape of this deterministic area can be modified for situations like multi-processor and partial reconfiguration systems, which require constrained areas. An advantage of our system is the configurable datapath width, which reduces the area used by an average of twenty percent. This ability, in turn, lets instructions execute with a smaller area footprint.

Our joint performance and area evaluation shows the trade offs that can be achieved by synthesizing processor configurations with different instruction-issue widths and execution units. The instruction execution rate was maximized when these two parameters matched each other. Larger parameter values generally resulted in more parallel execution, but systems with more than five execution units result in diminished performance.

Within confined areas, we can either use our empirical equations or the instruction execution rate graphs to maximize performance through the selection of an appropriate processor configuration. Regardless of what these configurations are, the larger the area used, the less effective performance mapping becomes. The instruction execution rate per unit area, or IER density, rapidly decreased from over sixty thousand instructions per second per slice to about a third of this initial value, identifying the minimal performance/area trade off for FPGA-based reconfigurable processors. Performance density gains can be further achieved by changing other aspects of the superscalar processor like the datapath width. Additionally, changing the register file size and the number of memory ports allowed space/time execution trade offs.

Chapter 7

Conclusions

In this dissertation we have presented the implementation and methodology to evaluate the available parallelism and subsequent required area in a reconfigurable superscalar processor architecture. This area and performance evaluation is critical for understanding the benefits and costs of the paradigm shift towards reconfigurable computing systems. Through the use of synthesis techniques to design, synthesize and implement superscalar processors onto reconfigurable fabric, we were able to monitor, evaluate, predict and control area resources. Thus this evaluation aided us in the search for parallel spatial execution through different processor configurations and area costs. Consequently, our findings benefit the reconfigurable computing community by providing means to calculate and quantify the maximum parallel-execution rates in a confined area for partially reconfigurable and multiprocessor systems.

Reconfigurable computing creates an arena where the higher performance of application-specific circuitry constantly battles with general-purpose and standardized solutions with lower non-recurring engineering costs. In the middle of the turmoil are a multitude of reconfigurable architectures that enable different amounts of datapath granularity. Among them, the FPGA's single-bit fine granularity makes it the most custom solution, while soft-processors create the most general purpose solution. Vendor soft-processors lack flexibility to exploit instruction and data-level parallelism, a limitation which we overcame by creating a configurable hybrid solution allowing custom parallel logic within a standard processor architecture framework. Our processor, named Hephaestus, achieves a balance between the flexibility offered by a general-purpose programmable processor and the benefits of custom circuits. It is an adaptable, reconfigurable, superscalar architecture that supports multiple issuing of instructions and execution units executing in parallel.

The structure of a Hephaestus processor is defined in VHDL with system-level parameters. The two main parameters were the instruction-issue width and the number of execution units, both of which enhance the parallel execution of instructions. Three other important parameters included the register file size, the number of memory ports, and the datapath width. We used HDL techniques to automatically synthesize the appropriate components, interconnect and communication protocols for any permutation of these parameters. The system was implemented in a Xilinx FPGA as an eleven-stage superscalar pipeline with register renaming for added performance. The execution units remained autonomous and independent entities, meaning that any custom user logic can be placed in them without affecting the rest of the system. To enable custom instructions, we set our software interface with a RISC-like ISA that accommodates extensions for custom logic and user-defined execution units.

By abstracting away custom circuits and implementing them as reconfigurable execution units, we were able to assess the number of instructions that could execute concurrently in a superscalar system. The instruction-execution rate was thus a function of the instruction-issue width, the number of parallel execution units, and the maximum clock frequency for that configuration. The area for the different processor configurations was evaluated by the total count of multi-purpose FPGA slices. This area had to follow explicit design rules for partial reconfiguration, so we subjected our processor to constrained areas following these rules. As for performance, the instruction execution rate metric was defined to merge the clock frequency and increased parallel execution opportunities for larger issue widths and increased execution units. The performance/area trade offs were evaluated by combining this performance and the area assessment.

Our results answered the fundamental questions driving this research: 1) What performance can be attained for this area? and 2) what area is needed for this performance? We provided an empirical equation for calculating the total FPGA slices used by a processor configuration, which was a polynomial function of the instruction-issue width and the number of reconfigurable execution units. By quantifying the area utilization necessary for parallel execution gains, we can also assess the maximum parallel performance in constrained areas.

The benefits of this assessment are two-fold: the area of a processor configuration can be calculated, and the optimal processor architecture can be selected for a delimited area within an FPGA. Different flavors of this assessment include the maximum execution rate per area block, the instruction-execution density per area unit, and the bit-execution density per area unit. The instruction-execution density showed that on average 34,630 instructions/second/slice can be achieved in a reconfigurable processor, with 20,000 instructions/second/slice as a minimum. Alternatively, the bit-execution density results showed that the apex of data execution occurs at a datapath width of 51 bits, with larger values no longer contributing to higher execution densities. These metrics can be used to evaluate partial reconfiguration costs, assisting in the automation of the partial reconfiguration process. In essence, we can now better understand the implementation compromises that await us during the upcoming paradigm shift towards reconfigurable parallel computing.

7.1 Limitations and Future Work

It is true that most general-purpose application can not benefit from architectures that exploit instruction-level parallelism. Of course, such applications are not the target of reconfigurable computing methods. Our processor not only allows ILP for computation-intensive applications, but also allows a customized ILP framework. Therefore, our solution can execute multiple instructions in parallel, and it can further execute sets of instructions within each custom execution unit.

Our processor can not achieve the performance levels of application-specific integrated circuits, nor it can match the speed and integrated application environment of vendor-dependent commercial processors. However, the implementation costs with our reconfigurable solution are greatly reduced, by avoiding the time and effort needed to modify fully-custom circuits. And even though our processors achieves comparatively slow clock frequencies, we noted that this detriment can be offset by packing more complex custom logic inside each execution unit to reduce latency.

One alternative to our processor solution is to use the vendor-specific soft-processors. These are obviously optimized for their companion vendor devices, and not surprisingly performed better than simple configurations of our processor. In addition, the configuration of these processors allowed the automatic addition of peripherals, and memory hierarchy control. While our research does not integrate peripherals and design automation for specific devices, it does provide vendor independence and design portability. Our focus was not on system integration, but in flexibility at the architecture level, a feature that vendor processors do not provide. To better compete with alternative solutions, performance needs to be enhanced and area reduced. By merging pipeline stages that are not bottleneck stages, we can decrease the execution latency and intermediate storage requirements, thus alleviating routing complexity to reduce delay.

Reducing the complexity and size of the processor will also reduce the system's place-and-route complexity. Due to the procedural nature of the hardware-description language code defining the processor's behavior, the place-and-route procedure needs an excessive amount of system memory to synthesize the processor. Such requirements can be relaxed by defining structural (rather than procedural) components to implement the register renaming logic. We believe this will give us more control over the register renaming logic and reduce delay, fomenting a faster, cleaner processor design.

Finally, to extend the implications of this research work beyond the devices currently in existence, we can scale our area assessment to different FPGA families, and eventually, different FPGA vendors. We have already provided a methodology to do this for different slice configurations within Xilinx families, but ideally, our vendor-independent approach should be evaluated in other FPGA platforms like those from Altera or Actel.

7.2 Concluding Remarks

This dissertation showed the implementation of a competitive superscalar processor core which is fit to produce speedups for a variety of applications. By itself, it allows the evaluation of many superscalar trade offs which might have only been possible by software simulation rather than implementation. Flexibility was the focus of the implementation, and we have not analyzed all the system variables to their full extent. Further analysis of the variables controlling the common data buses, instruction size, register file organization, memory addressing, branch prediction and reorder buffer is possible, but beyond the confines of this dissertation.

The impact of our results confirm that soft-core processors can be expressed as relocatable objects for reconfigurable architectures and tread level parallelism exploitation. Our analysis further assisted the field of reconfigurable computing by providing genuine metrics for the consequential costs of dynamic reconfiguration.

References

[1] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”, *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 2533, 1967.

[2] Gerald Estrin, “Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer”, *IEEE Ann. Hist. Comput.*, vol. 24, no. 4, pp. 3–9, 2002.

[3] J. R. Hauser and J. Wawrzynek, “Garp: a MIPS Processor with a Reconfigurable Coprocessor”, in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, Washington, DC, USA, 1997, p. 12, IEEE Computer Society.

[4] Miljan Vuletic, Laura Pozzi, and Paolo Ienne, “Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing”, in *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference*, Washington, DC, USA, September 2004, pp. 339–351, IEEE Computer Society.

[5] Miljan Vuletic, Laura Pozzi, and Paolo Ienne, “Seamless Hardware-Software Integration in Reconfigurable Computing Systems”, *IEEE Des. Test*, vol. 22, no. 2, pp. 102–113, 2005.

[6] Member-Herbert Walder and Member-Marco Platzner, “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”, *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1393–1407, November 2004, Christoph Steiger.

[7] R. Enzler, C. Plessl, and M. Platzner, “System-Level Performance Evaluation of Reconfigurable Processors”, *Microprocessors and Microsystems*, vol. 29, no. 2-3, pp. 63–73, 2005.

[8] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ed Komp, Ron Sass, and David Andrews, “Enabling a Uniform Programming Model Across the Software/Hardware Boundary”, in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2006, pp. 89–98, IEEE Computer Society.

[9] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden, “Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link”, *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[10] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp, “The Case for High level Programming Models for Reconfigurable Computers”, in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Proceedings of the*, June 2006.

[11] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee, “CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit”, in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, New York, NY, USA, June 2000, pp. 225–235, ACM.

[12] S. Hauck, TW Fry, MM Hosler, and JP Kao, “The Chimaera Reconfigurable Functional Unit”, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 2, pp. 206–217, 2004.

[13] John Reid Hauser, *Augmenting a Microprocessor with Reconfigurable Hardware*, PhD thesis, University of California, Berkeley, 2000.

[14] Xilinx, Inc, *Embedded System Tools Guide*, October 2003.

[15] Xilinx, Inc, *Microblaze Processor Reference Guide*, January 2008.

[16] F. Plavec, “Soft-Core Processor Design”, Master's thesis, University of Toronto, 2004.

[17] A.P. Ambler, “Hardware Accelerators for CAD”, *Computer-Aided Engineering Journal*, vol. 6, no. 3, pp. 77–81, June 1989.

[18] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood, “Hardware-Software Co-Design of Embedded Reconfigurable Architectures”, in *Proceedings of the 37th conference on Design automation*, New York, NY, USA, 2000, ACM New York, NY, USA, pp. 507–512, ACM.

[19] André DeHon, “The Density Advantage of Configurable Computing”, *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[20] J. Greco, B. Holland, I. Troxel, G. Barfield, V. Aggarwal, and A. George, “USURP: A Standard for Design Portability in Reconfigurable Computing”, in *IEEE Symp. on Field-programmable Custom Computing Machines (FCCM), Napa Valley, CA, Apr*, 2006, pp. 24–26.

[21] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, “Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs”, in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'07)*, 2007.

[22] H.C. Freitas, D.M. Colombo, F.L. Kastensmidt, and P.O.A. Navaux, “Evaluating Network-on-Chip for Homogeneous Embedded Multiprocessors in FPGAs”, *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pp. 3776–3779, May 2007.

[23] Altera Corporation, *Nios II Processor Reference Handbook*, November 2008.

[24] Javier Resano, Daniel Mozos, Diederik Verkest, and Francky Catthoor, “A Reconfiguration Manager for Dynamically Reconfigurable Hardware”, *IEEE Des. Test*, vol. 22, no. 5, pp. 452–460, 2005.

[25] James E. Thornton, “Design of a computer: the CDC 6600”, *Scott and Foresman, Glenview, Ill*, 1970.

[26] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., “Architecture of the IBM System/360”, pp. 17–31, 2000.

[27] Raul Camposano, “Will the ASIC Survive?”, in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, New York, NY, USA, 2004, pp. 5–5, ACM.

[28] J. Villasenor and B. Hutchings, “The Flexibility of Configurable Computing”, *Signal Processing Magazine, IEEE*, vol. 15, no. 5, pp. 67–84, 1998.

[29] Ahmed Nabil and Darren Zacher, “ASIC Prototyping with FPGAs”, Technical Publication TECH7430-w, Mentor Graphics Corporation, April 2007.

[30] Bill Troxel, Brendan Cremen, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, and Paul S. Zuchowski, “A Hybrid ASIC and FPGA Architecture”, *Computer-Aided Design, International Conference on*, vol. 0, pp. 187–194, 2002.

[31] Scott Hauck, "The Future of Reconfigurable Systems", in *Keynote Address, 5 thCanadian conference on field programmable devices, Montreal*, 1998.

[32] Inc Nikkei Business Publications, "FPGAs Leveling with ASICs, ASSPs", `http://techon.nikkeibp.co.jp/article/HONSHI/20080924/158406/`.

[33] LinuxDevices, *Embedded Processor and System-on-Chip Quick Reference Guide*, November 2006.

[34] Jürgen Becker, "Configurable Systems-on-Chip (CSoC)", in *SBCCI '02: Proceedings of the 15th symposium on Integrated circuits and systems design*, Washington, DC, USA, 2002, p. 379, IEEE Computer Society.

[35] Philippe Magarshack and Pierre G. Paulin, "System-on-Chip Beyond the Nanometer Wall", in *DAC '03: Proceedings of the 40th conference on Design automation*, New York, NY, USA, 2003, pp. 419–424, ACM.

[36] Wayne Wolf, "A Decade of Hardware/Software Codesign", *Computer*, vol. 36, no. 4, pp. 38–43, 2003.

[37] Jorge Ortiz, "Hardware/Software Co-design of Schedulers for Real Time and Embedded Systems", Master's thesis, The University of Kansas, May 2004.

[38] Tom Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design", in *Design and Test. IEEE Transactions on*, August 1984, pp. 21–39.

[39] M. Franzmeier, C. Pohl, M. Porrmann, and U. Ruckert, "Hardware Accelerated Data Analysis", in *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*, January 2004, pp. 309–314.

[40] G. Russel, "The Anatomy of Hardware Accelerators for VLSI Circuit Design", *Computer-Aided Engineering Journal*, vol. 6, no. 3, pp. 82–91, 1989.

[41] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective", in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, Piscataway, NJ, USA, 2001, pp. 642–649, IEEE Press.

[42] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh, "A Quick Safari Through the Reconfiguration Jungle", in *DAC '01: Proceedings of the 38th conference on Design automation*, New York, NY, USA, 2001, pp. 172–177, ACM.

[43] Eduardo Sanchez, Jacques-Olivier Haenni, Jean-Luc Beuchat, André Stauffer, Andrés Perez-Uribe, and Moshe Sipper, "Static and Dynamic Configurable Systems", *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 556–564, 1999.

[44] Ian Page, "Reconfigurable Processor Architectures", *Microprocessors and Microsystems*, vol. 20, no. 3, pp. 185–196, 1996.

[45] In-Stat Press Release, "FPGA Market Will Reach $2.75 Billion by Decade's End", `http://www.instat.com/press.asp?Sku=IN0603187SI&ID=1674`.

[46] D. Langen, J.C. Niemann, M. Porrmann, H. Kalte, U. Rückert, and G. Paderborn, "Implementation of a RISC Processor Core for SoC Designs–FPGA Prototype vs. ASIC Implementation", in *Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002, pp. 27–34.

[47] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications", in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, New York, NY, USA, February 1999, pp. 135–143, ACM.

[48] A. Dehon and E. Mirsky, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, 1996, pp. 157–166.

[49] MB Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams", in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, 2004, pp. 2–13.

[50] Marlene Wan, Hui Zhang, Varghese George, Martin Benes, Arthur Abnous, Vandana Prabhu, and Jan Rabaey, "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System", *J. VLSI Signal Process. Syst.*, vol. 28, no. 1-2, pp. 47–61, 2001.

[51] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", in *ARVLSI '99: Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, Washington, DC, USA, 1999, p. 23, IEEE Computer Society.

[52] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas, "Managing Pipeline-Reconfigurable FPGAs", in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, New York, NY, USA, 1998, pp. 55–64, ACM.

[53] Samuel K. Moore, "Multicore Is Bad News For Supercomputers", Technical Publication Spectrum 11-08, IEEE, November 2008.

[54] Cray, Inc, *Cray XD1 Datasheet*, January 2006.

[55] K. Bondalapati and V.K. Prasanna, "Reconfigurable Meshes: Theory and Practice", in *Reconfigurable Architectures Workshop. International Parallel Processing Symposium on*, April 1997.

[56] Roman Lysecky and Frank Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning", in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, Washington, DC, USA, March 2005, pp. 18–23, IEEE Computer Society.

[57] Lattice Semiconductor Corporation, "Open and Easy Microprocessor Designs Using the LatticeMico32", A Lattice Semiconductor White Paper, February 2008.

[58] FPGA and Programmable Logic Journal, "Embedded Microprocessor Trends and Challenges", `http://www.fpgajournal.com/whitepapers_2008/q1_embedded_lattice.htm`.

[59] Ralph D. Wittig, "OneChip: An FPGA Processor With Recongurable Logic", Master's thesis, University of Toronto, 1995.

[60] Arnaud Lagger, "Self-Reconfigurable Platform for Cryptographic Application", Master's thesis, Swiss Federal Institute of Technology Lausanne, 2006.

[61] Xilinx, Inc, *MicroBlaze v7 FAQ*, July 2008.

[62] Altera Corporation, *Nios II Custom Instruction User Guide*, May 2008.

[63] Lattice Semiconductor Corporation, *LatticeMico32 Processor Reference Manual*, November 2008.

[64] Lattice Semiconductor Corporation, *LatticeMico32 Tutorial*, October 2008.

[65] Gaisler Research, *GRLIB IP Library User's Manual*, August 2008.

[66] Norman P. Jouppi and David W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", in *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 1989, pp. 272–282, ACM.

[67] David W. Wall, "Limits of Instruction-Level Parallelism", in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 1991, pp. 176–188, ACM.

[68] Kees A. Vissers, "Parallel Processing Architectures for Reconfigurable Systems", *Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 10396, 2003.

[69] Altera Corporation, *Creating Multiprocessor Nios II Systems*, Dec 2007.

[70] Jorge E. Carrillo and Paul Chow, "The Effect of Reconfigurable Units in Superscalar Processors", in *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, New York, NY, USA, 2001, pp. 141–150, ACM.

[71] A. Romer, R. Enzler, D. Cottet, and G. Troster, "Reconfigurable FPGA Processor", Tech. Rep., Swiss Federal Institute of Technology, 2000.

[72] Steve Trimberger, "Xilinx, FPGA's, and Moore's Law", `http://www.skatelescope.org/US_SKA_Technology_Day06/Trimberger.pdf`, 2005.

[73] David Wentzlaff and Anant Agarwal, "A Quantitative Comparison of Reconfigurable, Tiled, and Conventional Architectures on Bit-Level Computation", in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 289–290, IEEE Computer Society.

[74] Marco Di Natale and Enrico Bini, "Optimizing the FPGA Implementation of HRT Systems", in *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, Washington, DC, USA, 2007, pp. 22–31, IEEE Computer Society.

[75] GateRocket Dave Orecchio, "Mammoth FPGAs Require New Tools", `http://www.gaterocket.com/device-native-verification/bid/7966/Mammoth-FPGAs-Require-New-Tools`, 2009.

[76] Mentor Graphics Corporation, *ModelSim SE Users Manual, v6.5d*, May 2009.

[77] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[78] J.P. Shen and M.H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Science/Engineering/Math, 2005.

[79] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, May 2002.

[80] Xilinx, Inc, *Synthesis and Simulation Design Guide*, April 2006.

[81] James E. Smith and Gurindar Singh Sohi, "The Microarchitecture of Superscalar Processors", *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, Dec 1995.

[82] S. Hurley, "The Application of Tomasulo's Method", University of Dublin, Trinity College, May 2005.

[83] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis, "Register Renaming and Dynamic Speculation: An Alternative Approach", in *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, Los Alamitos, CA, USA, 1993, pp. 202–213, IEEE Computer Society Press.

[84] Roberto Perez-Andrade, Rene Cumplido, Fernando Martin Del Campo, and Claudia Feregrino-Uribe, "A Versatile Linear Insertion Sorter Based on a FIFO Scheme", *VLSI, IEEE Computer Society Annual Symposium on*, vol. 0, pp. 357–362, 2008.

[85] Chen-Yi Lee and Jer-Min Tsai, "A Shift Register Architecture for High-Speed Data Sorting", *J. VLSI Signal Process. Syst.*, vol. 11, no. 3, pp. 273–280, 1995.

[86] A.A. Colavita, A. Cicuttin, F. Fratnik, and G. Capello, "SORTCHIP: a VLSI Implementation of a Hardware Algorithm for Continuous Data Sorting", *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 6, pp. 1076–1079, June 2003.

[87] Lluis Ribas, David Castells, and Jordi Carrabina, "A Linear Sorter Core based on a Programmable Register File", in *XIX Conference on Design of Circuits and Integrated Systems, DCIS*, 2004, pp. 635–640.

[88] Jorge Ortiz and David Andrews, "A Configurable High-Throughput Linear Sorter System (submitted)".

[89] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team, "Scaling to the End of Silicon with EDGE Architectures", *Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[90] Alan R. Weiss, "Dhrystone benchmark: History, Analysis, Scores and Recommendations", EEMBC White paper, Noverber 2002.

[91] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit, "A Dynamic Reconfiguration Run-Time System", in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, Apr 1997, pp. 66–75.

[92] Jürgen Becker, Thilo Piontek, and Manfred Glesner, "DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communications Applications", in *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, London, UK, 2000, pp. 312–321, Springer-Verlag.

[93] Zhiyuan Li and Scott Hauck, "Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation", in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, New York, NY, USA, 2002, pp. 187–195, ACM.

[94] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck, "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing", *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 10, no. 3, pp. 209–220, 2002.

[95] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs", in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 399–400.

[96] C. Bobda, M. Huebner, A. Niyonkuru, B. Blodget, M. Majer, and A. Ahmadinia, "Designing Partial and Dynamically Reconfigurable Applications on Xilinx Virtex-II FPGAs using Handel-C", Technical Report 03-221, 12, 221, University of Erlangen-Nuremberg, Department of CS, December 2004.

[97] J.A. Williams and N. Bergmann, "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip", in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[98] M. Hübner and J. Becker, "Exploiting Dynamic and Partial Reconfiguration for FPGAs: Toolflow, Architecture and System Integration", in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, New York, NY, USA, 2006, pp. 1–4, ACM.

[99] Ken Eguro and Scott Hauck, "Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development", in *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2003, p. 111, IEEE Computer Society.

[100] R. Hartenstein, "The Microprocessor Is No Longer General Purpose: Why Future Configurable Platforms Will Win", in *Innovative Systems in Silicon, 1997. Proceedings., Second Annual IEEE International Conference on*, 1997, pp. 2–12.

[101] Rich Faris and Anil Khanna, "Case Studies: Using Physical Synthesis to Improve FPGA Performance", Technical publication, Mentor Graphics Corporation, March 2004.

[102] S. Singh, "Death of the RLOC", in *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[103] Xilinx, Inc, *Constraints Guide*, April 2009.

[104] J. Thorvinger, "Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support", Master's thesis, Lund Institute of Technology, 2004.

[105] G. Mermoud, "A Module-Based Dynamic Partial Reconfiguration Tutorial", *Logic Systems Laboratory, Ecole Polytechnique Fédérale de Lausanne, November*, 2004.

[106] Xilinx, Inc, *Xilinx XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based, Application Note*, September 2007.

[107] Tanay Karnik and Sung-Mo Kang, "An empirical model for accurate estimation of routing delay in fpgas", *Computer-Aided Design, International Conference on*, vol. 0, pp. 0328, 1995.

[108] J. Henkel and S. Parameswaran, *Designing Embedded Processors: A Low Power Perspective*, Springer, 2007.

[109] Jorge Ortiz, "A Reconfigurable Superscalar Processor Architecture for FPGA-Based Designs", in *CDES: Proceedings of the 2009 International Conference on Computer Design*, July 2009, pp. 211–217.

[110] Masayuki Hiromoto, Hiroyuki Ochi, and Yukihiro Nakamura, "An Asynchronous IEEE-754-standard Single-precision Floating-point Divider for FPGA", *IPSJ Transactions on System LSI Design Methodology*, vol. 2, pp. 103–113, 2009.

[111] Florent de Dinechin, Cristian Klein, and Bogdan Pasca, "Generating high-performance custom floating-point pipelines", Tech. Rep. 2009-16, cole Normale Suprieure de Lyon, 2009.

[112] I. Kuon and J. Rose, "Area and Delay Trade-offs in the Circuit and Architecture Design of FPGAs", in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays.* ACM New York, NY, USA, 2008, pp. 149–158.

[113] E.M. Panainte, K. Bertels, and S. Vassiliadis, "FPGA-area Allocation for Partial Run-Time Reconfiguration", in *Proceedings of ProRISC*, 2005, pp. 415–420.

[114] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood, "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1355–1371, 2001.

[115] Jorge Ortiz, "Area Evaluation for Parallel Execution in Reconfigurable Processor Architectures", in *ERSA: Proceedings of the 2009 International Conference on Engineering of Reconfigurable Systems Algorithms*, July 2009, pp. 328–331.