SHARED INSTRUCTION-SET EXTENSIONS FOR SOFT MULTIPROCESSOR SYSTEMS IMPLEMENTED ON FIELD-PROGRAMMABLE GATE ARRAYS

by

Erin Johnston

BASc, The University of British Columbia, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE COLLEGE OF GRADUATE STUDIES

(Electrical Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

December 2012

Abstract

Soft-core embedded systems implemented on FPGAs offer a high level of flexibility. Application specific customizations can be added in the form of extensions to the processor's regular instruction-set. These custom instructions benefit run-time performance, but come at the cost of increased resource usage. Reducing the overall FPGA area required to implement a system will decrease static power consumption and allow a smaller, cheaper device to be used. There is a constant effort to reduce area and power consumption while maintaining performance benefits attained through customizations.

This thesis presents a new architecture to share custom instruction units among multiple processors in a system. This implementation allows run-time performance benefits to be maintained while decreasing the overall resource usage. The shared architecture is implemented using an arbitrator to determine processor access to each custom instruction in a set. Custom instruction inputs and outputs are controlled using additional multiplexors and selection hardware. Results for a sample system using fine-grained custom instructions show that sharing can reduce the implementation area by up to 24% with minimal impact to the critical path delay. This reduction remains high at 19% for a coarse-grained case study of an encryption algorithm called SHA.

The custom instruction configuration depends on the application being performed. A benchmark generator and simulator are also developed to evaluate candidates for custom instruction implementation and efficiently explore the design space. The overall run-time performance of the candidate systems can also be evaluated using these tools. The simulator can also be used with an input trace to determine cycle accurate run-time performance for a real application, without requiring the entire system to be designed and implemented in hardware. The simulator shows up to 53% run-time improvement for a shared fine-grained system over a system with no custom instructions. Hardware run-time results for the coarse-grained case study improve run-time up to 13.5% over a system with no custom instructions.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CAD	Computer Aided Design
DMA	Direct Memory Access
FIPS	Federal Information Processing Standard
FPGA	Field-Programmable Gate Array
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IP	Intellectual Property
ISE	Instruction-Set Extension
MtM	More than Moore
NIST	National Institute of Security and Technology
NRE	Non-Recurring Engineering
RISC	Reduced Instruction-Set Architecture
SHA	Secure Hash Algorithm
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
TIE	Tensilica Instruction Extension

List of Operation Symbols

X, *	Multiply
+	Add
<<	Left-shift
^	Exclusive-or
|	Logical or
&	Logical and

Acknowledgements

First, I would like to thank my supervisor Dr. Peter Hallschmid. It is due to his encouragement that I endeavored to pursue graduate studies. He spurred my interest in research and asking endless questions.

I would like to thank my parents for encouraging me to keep learning. They have provided unlimited support and have always pushed me to do my best. Also, I would like to thank Ben, my roommates, and the rest of my family for their support and understanding while I completed my work.

This work was supported by NSERC and through hardware and software donations provided by Altera.

1 Chapter: Introduction

1.1 Background Information

Embedded systems are compact designs that implement specific applications or controls under a specific set of power, cost, speed, and/or reliability constraints. They usually comprise of one or many processing cores, on-chip memory, and input and output peripherals such as radio frequency transmitters, biosensors, and actuators. We encounter embedded systems every day in our cell phones, vehicles, navigation systems, MP3 players, and gaming consoles, to name a few. They can also be found in more complex devices such as aircraft guidance systems and medical devices. The cost and performance requirements for such devices are constantly evolving and pushing design to new levels.

1.1.1 Market Trends

The need to produce faster, more complex systems is constantly competing with the desire to fit designs into a smaller area and to consume less power. According to Moore's law [**1**] which was stated in 1965, the number of transistors on an integrated circuit will double roughly every two years. This trend has held true since that time, with semiconductor companies striving to meet or beat this goal.

Moore's law has pushed more than just the limits of integrated circuit design and transistor technology. In order to keep up with the scaling designs, engineers require more sophisticated computer aided design (CAD) tools. Despite advancing tools, a productivity gap trend is still occurring. This means there is an increasing gap between design complexity and engineer productivity.

Along with increasing complexity and reduced productivity, increasing time-to-market demands also means less time for revisions and testing, forcing more engineers per design task. Extra manpower and technological demands can start to make development costs unreasonable.

One option for reducing costs is to develop using a system-on-chip (SoC). According to Chang et al [**2**], an SoC incorporates a programmable processor, on-chip memory, interfaces to peripheral devices, and predesigned and pre-verified accelerating modules. These modules are referred to as intellectual property (IP) components. SoCs are favorable because they can help reduce development time and, therefore, overall cost.

SoCs can be verified on field-programmable gate arrays (FPGAs) to ensure correct functionality before fabrication. However, SoCs still have high non-recurring engineering (NRE) costs associated with masks and fabrication, which according to Margarshack and Paulin [**3**] is the main explanation for the growth in FPGAs. The convenience of an SoC depends on the availability of the IP components and the ease of use of the design platform. This type of embedded system offers programmability which allows for a wider variety of applications. Starting with a base system that includes a processor and memory, IP components and programmed customizations can be added to a system to tailor it to an application or application domain. Using a programmable device gives the designer better control over the system customization.

A second approach is to use advanced design tools such that system customizations can be created in a high-level language. Using a higher level of abstraction keeps the designer removed from transistor-level intricacies. For example, programmable logic allows the designer to create customizations using a hardware description language (HDL).

Synthesis tools are then used to compile the HDL and translate it to board-level connections. The engineer must have knowledge of the hardware and logic, but is not required to design the actual layout. Using IP components raises the level of abstraction even higher by allowing the engineer to provide parameters and then automatically generate the component.

Despite the urgent need to mitigate the productivity gap, some industry trends have led to increases in the productivity gap such as the increased use of multiprocessor systems. Although multiprocessor systems have led to another level of complexity to designs that further increases the productivity gap, work by Borkar [**4**] describes the reasoning and benefits behind the growing trend toward multiple cores. First, implementing multiple smaller processors can actually decrease the individual complexity and improve power performance over a single, large core. Also, it is easier to distribute the load and optimize for frequency on each processor.

1.1.2 Field-Programmable Gate Arrays

An FPGA is a programmable logic device that can be reconfigured after manufacturing. FPGAs are the most common form of programmable logic and can be configured to perform complex logical operations and even store results in memory blocks and registers. A typical island-style FPGA architecture is shown in Figure 1.

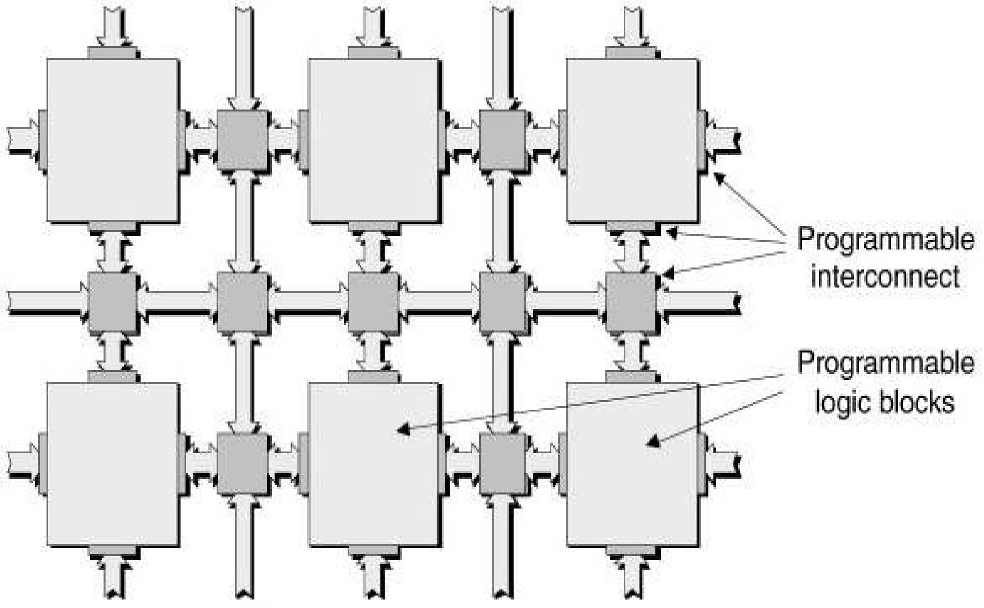


Figure 1. FPGA island style architecture [5]**.**

Programmable logic blocks are surrounded by programmable interconnect or routing channels. Each of the logic blocks make up a small portion of the overall circuit. By connecting these blocks using the interconnect, a larger, more complex circuit can be formed.

FPGAs can be used instead of application-specific integrated circuits (ASICs) to implement highly customized embedded system designs. While ASICs are designed in much the same way as systems on FPGAs, ASICs have much higher NRE costs associated with fabrication. Zuchowsky et al [**6**] discuss how even though performance and density for ASICs still lead over FPGAs, the flexibility and time-to-market make FPGAs highly appealing. Two companies occupy the majority of the FPGA market: Altera [**7**] and Xilinx [**8**]. According to Altera [**9**], FPGAs offer a variety of advantages over ASICs such as a shorter time to market and lower non-recurring engineering costs. Much of the design effort and cost created by designing an ASIC are eliminated using an FPGA. They have a long product life and can be reprogrammed by the designer for testing and changing needs.

FPGAs are available that implement either hard or soft processing cores. A hard-core processor is a core implemented from a predefined block. This is a fixed resource and cannot be reprogrammed or customized after fabrication. A soft-core processor is a core implemented by programming the logic blocks of the FPGA to act as a microprocessor. It is a predesigned intellectual property core that is easily synthesized and highly flexible. Soft-core processors are found in many everyday systems and are undetected by the end user. They can be found in voice over IP systems, routers, electronic signs, security systems, and vehicles.

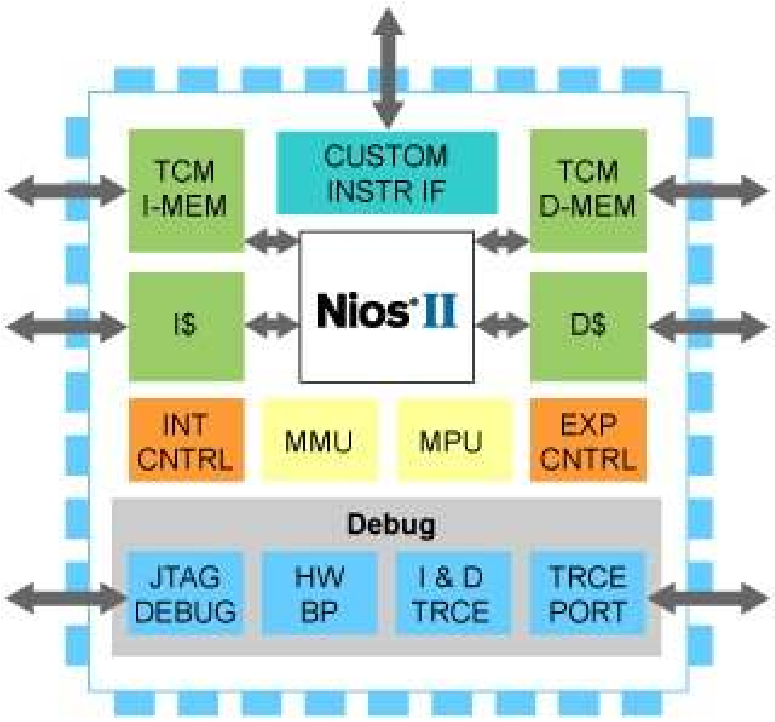


Figure 2. Nios II soft-core processor [10]**.**

According to Tong et al [**11**], the three leading soft-core processor providers are Altera with the Nios II, Xilinx with the MicroBlaze and PicoBlaze, and Tensilica with the Xtensa. Figure 2 shows a Nios II core surrounded by design options including instruction and data cache, a memory management unit, and debug hardware. All three companies use reduced instruction-set computer (RISC) cores and have options for pre-made fixed hardware block customizations or user-defined hardware modules. Fixed block implementations include multipliers, barrel shifters, and dividers. Custom implementations change from system to system and are specific to the task being performed.

1.1.3 Instruction-Set Extensions

1.1.3.1 Overview

Embedded system customization can come in many different forms. On a customizability spectrum, soft-core systems implemented on FPGAs lie between general-purpose processors (GPPs) and fixed logic processors. GPPs are robust microprocessors that are flexible in their application, serving a wide array of tasks from computation to communication to display. ARM is a well-known producer of GPPs that are often found in cell phones, MP3 players, and game consoles [**12**]. GPPs are purchased and ready to use "off-the-shelf", but do not provide the performance of a customized processor. At the other end of the customizability spectrum lie fixed logic processors. These are complete microprocessors designed for one specific purpose. They offer high performance, but unlike a GPP are not portable to other tasks. Fixed-logic processors have extremely high NRE costs and require significant time to design and manufacture.

Soft-core systems implemented on FPGAs provide a tradeoff between the flexibility of a GPP and the performance of a fixed-logic processor. They can be designed or reconfigured for varying applications, but offer customizability through instruction-set extensions (ISEs). ISEs are a form of soft-processor customization in which the instruction-set can be extended with custom instructions tailored to an application or application domain. ISEs are not an entire custom processor, but rather additions to a base core. The customizability options described are certainly not the only options available, as works like that by Clark et al [**13**] blur the lines by combining GPPs with custom hardware.

Several companies offer processors configurable with ISEs: Stretch and the previously mentioned Altera, Xilinx, and Tensilica. Both Altera [**7**] and Xilinx [**8**] products require the user to define custom instructions using an HDL. Stretch [**14**] tools allow users to automatically create ISEs using only C/C++. Tensilica [**15**] allows developers to produce custom instructions in one of two ways. First, custom instructions can be defined by the user in the Tensilica Instruction Extension (TIE) language. The other option is to use the Xpress tool that will automatically generate custom instructions in the TIE language based on the application source code.

1.1.3.2 CAD for ISEs

ISEs can range in size and complexity from a few simple operations to an entire co-processing unit. Custom instructions comprising of few operations are referred to as fine-grained custom instructions. Fine-grained instructions use hard-coded values and inputs passed to the custom instruction as operands. They are defined by selecting critical clusters of operations from the application to be executed on the processor.

Larger ISEs are called coarse-grained custom instructions, and they often take the place of an entire function or application in software. Coarse-grained instructions often require a greater number of operands that can be hard-coded, passed to the custom instruction, or retrieved directly from memory. For both types, custom instructions are implemented in hardware as a single, complex instruction. For example, Figure 3 shows a directed acyclic graph representation of a small section of application code. Each node or vertex represents an operation, and each arrow or edge represents the operands associated with it. The encircled section of the graph shows a candidate for fine-grained custom instruction implementation.

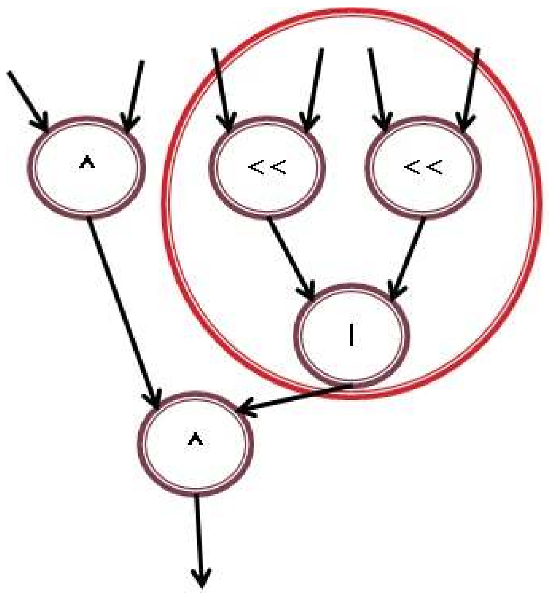
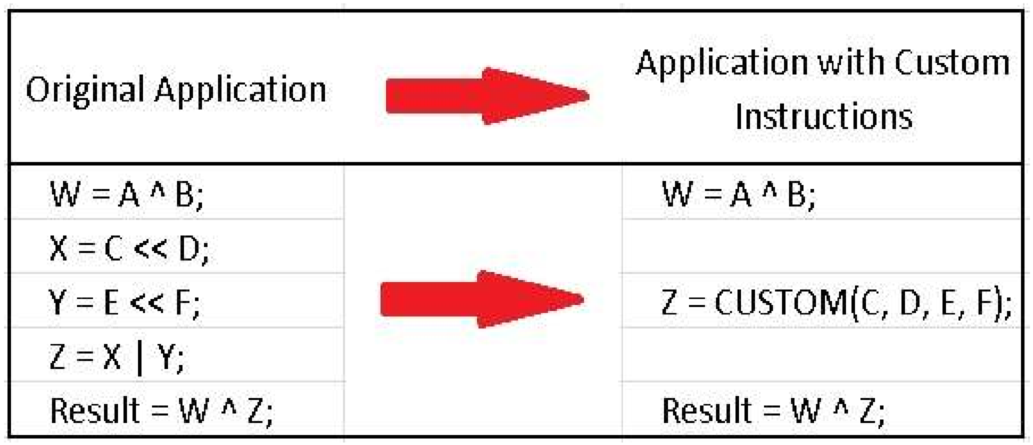


Figure 3. Directed acyclic graph showing custom instruction selection.

The three selected operations in Figure 3 will be implemented in hardware, and all calls in the application following this pattern will be replaced with a single call to the custom instruction. Figure 4 is the application code represented by the directed acyclic graph in Figure 3.

After implementing the selected instructions in hardware, the three operations, shown on the left, in the original application are replaced with a single custom instruction call, shown on the right. The syntax for using the custom instruction will vary depending on the processor being used, but they are often a simple macro or function call. There are many works, such as that by Peymandoust et al [**16**], that not only select custom instructions from application source code, but also replace lines of code in the original application with calls to the custom instructions.



Original Application	➡	Application with Custom Instructions
W = A ^ B;		W = A ^ B;
X = C << D;		
Y = E << F;	➡	Z = CUSTOM(C, D, E, F);
Z = X | Y;		
Result = W ^ Z;		Result = W ^ Z;

Figure 4. Application code replaced with single custom instruction call.

When evaluating candidate custom-instructions, the primary figure-of-merit is the speed increase gained by running the operations in hardware and the frequency of execution in the application. Unfortunately, not all candidate sets of instructions can be selected for implementation due to constraints imposed by the hardware. In particular, the processor's register file limits the number of inputs and outputs that can be used for any instruction, including custom instructions. Further, a sub-graph convexity constraint must also be adhered to as discussed in Atasu et al [**17**].

Figure 5 shows the same directed acyclic graph from Figure 3, but with a different set of encircled operations. This sub-graph is not a valid selection for custom instruction implementation for two reasons: it is not a convex sub-graph, and the register file constraints are violated. In the figure, the output from the ***left-shift*** becomes an input to the ***or***, an operation that is not contained within the sub-graph. The output from the ***or*** then becomes an input to the ***exclusive-or***, an operation that is again within the sub-graph. To satisfy the convexity constraint, the flow of operands within a custom instruction cannot leave the sub-graph and then re-enter. This selection also violates the most common register file constraints which allow two or four inputs and one output. In Figure 5, bold arrows represent inputs to the selection, and dashed arrows represent outputs. This custom instruction selection shows a total of five inputs and two outputs.

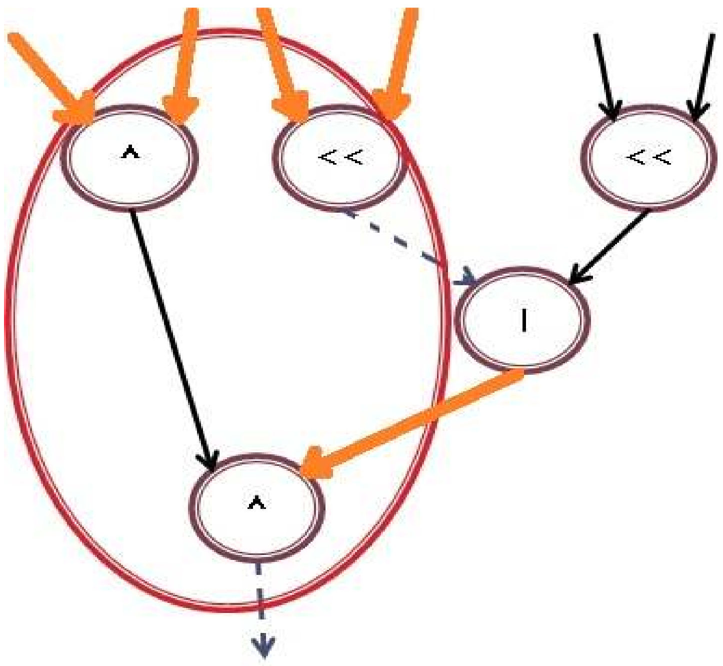


Figure 5. Custom instruction convexity and register file constraint.

Although obeying register file constraints is the easiest method to implementing custom instructions, it limits the performance gains achievable with larger custom instructions. One workaround is to access memory directly from within the custom instruction using direct-memory-access (DMA). This is useful for cases in which a much greater number of inputs or outputs are required than the register file will allow. Temporary inputs and outputs are not stored by the register file, but rather are kept within the custom instruction until the final outputs are produced.

A simpler method, that can be used when just a few extra inputs or outputs are needed, is to pipeline the custom instruction with additional custom instructions designed to read/write additional operands to/from the register file. For example, an instruction requiring four inputs and one output, but with a register file constraint of two inputs and one output, could use the first custom instruction call to load the first two inputs. These inputs are then saved within the custom instruction while a second custom instruction call is made that loads the next two inputs. This second call to the custom instruction then performs the operations, and the result can be returned at the end of its execution.

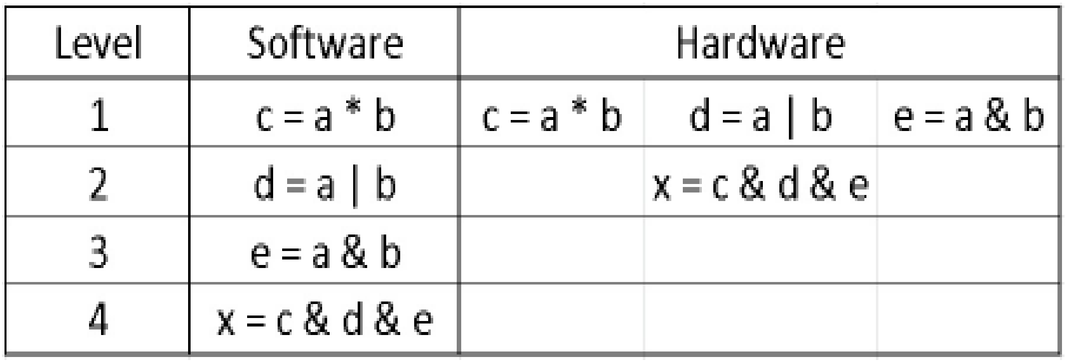
Although ISEs can result in significant performance gains, they come at the cost of increased area and static power. The impact of additional logic is especially important to consider for FPGA implementations in which even a small improvement in area could lead to a much cheaper part. Thus, with the addition of each new instruction, one must carefully consider the implications on resource usage and the predicted impact on run-time performance. In a multiprocessor system, both the performance benefits and area drawbacks are multiplied with the addition of each custom instruction.

1.1.3.3 Benefits of ISEs

The impact of custom instruction implementation is reduced run-time and reduced dynamic power dissipation. By replacing a cluster of operations in the application with one custom instruction call, execution cycles are saved in several ways. First, the Von Neumann bottleneck [**18**], caused by the latency in transfers between the processor and memory, is reduced. Even the fastest processors are limited by this transfer rate and spend time stalled, waiting for data and instructions. ISEs require fewer fetches from instruction memory or cache thus reducing the bottleneck. Only one instruction call is required to perform an entire set of operations.

ISEs also save execution cycles when parallelism is exploited. Parallelism is the ability of a system to execute multiple tasks at the same time. Instruction-level parallelism is the execution of multiple instructions at once. Only instructions that do not depend on each other's results can be executed in parallel. According to Callahan and Wawrzynek [**19**], reconfigurable systems such as those implemented on an FPGA can easily capitalize on the benefits of thread-level parallelism. Sequential programs are reduced in length when basic blocks are converted to a hardware implementation and multiple instructions execute at the same time.

Custom instructions employ a form of instruction-level parallelism to reduce run-time over the original application. For example, Figure 6 shows a section of code that is implemented sequentially in software. When implemented as a custom instruction in hardware, the first three levels can be executed at the same time, since one does not depend on the result from another. This improves run-time which in turn reduces overall power consumption.



Level	Software	Hardware		
1	c = a * b	c = a * b	d = a | b	e = a & b
2	d = a | b		x = c & d & e	
3	e = a & b			
4	x = c & d & e			

Figure 6. Software vs custom instruction parallelism.

The addition of custom instructions can also benefit the system in other ways such as requiring a reduced instruction memory due to a smaller application code. Also, the use of constants in an application can be a challenge for the processor if the constant length cannot be represented properly in an instruction op-code. The value must either be pulled from a literal pool, or the operation must be broken down into multiple instructions. Constants in custom instructions, however, are implemented in the hardware.

Finally, ISEs affect the number of registers and the number of accesses to the register file required. Energy is saved when operations are combined into a single custom instruction and fewer register file accesses are needed. Assembly code is also generated much easier during program compilation when custom instructions are used. First, there are fewer instructions and lines of code to compile. Second, “register pressure” is reduced during register allocation because fewer registers are needed to complete the same computation. This is because many of the temporary values normally needed now reside within the custom instruction.

1.1.4 Multi-core Systems

As mentioned in Section 1.1.1, there is a growing trend towards multi-core systems. According to Wolf [**20**], strict performance requirements and power and cost constraints have pushed engineers to develop multiprocessor systems. In multiprocessor systems, performance benefits are seen as each processor is customized for its application. Rather than using one large processor to perform a variety of tasks, multiple cores can be used to execute specific tasks.

Compared to separate implementations, multiprocessor systems on a single chip can reduce cost and size by providing the opportunity for resource sharing among cores. Resources are better utilized by sharing, and costly space can be eliminated or freed up for other custom accelerators. Although multiprocessor systems are often used with GPPs, they are easily implemented using soft-cores on FPGAs. Also, multi-core systems on FPGAs can exploit the benefits of customizability and resource sharing.

When applied to multi-core architectures, ISEs introduce heterogeneity. The instruction-set of each core is tailored to its assigned task, thus maximizing performance and reducing dynamic power. Heterogeneous multi-core architectures of this type can be created with low non-recurring costs and high flexibility when implemented on a programmable fabric. Biswas et al [**21**] [**21**] show that, with just one large ISE, the average application speed increase is 1.4 times while the energy savings can reach up to 40%. These savings are even more significant when applied to a multiprocessor system, where each core has the opportunity to use custom instructions.

1.2 Research Objectives

The goal of this research is to evaluate the impact of resource sharing, in the form of ISEs, among customizable processors implemented on an FPGA. The effects on FPGA area and system run-time will be monitored. This thesis will focus on two types of ISEs: fine-grained custom instructions and coarse-grained custom instructions. Both types will be considered in the evaluation of customizable processors.

A method will be determined for increasing multiprocessor system performance through custom instructions while meeting strict area constraints. The design space will be thoroughly explored to increase performance and reduce area for a wide range of applications. The focus will be on resource sharing in soft-core multiprocessor systems that promote flexibility and easily allow for modifications.

To accomplish these objectives, a new architecture must be developed that will maintain run-time benefits incurred by custom instructions, but will also decrease the FPGA resource usage. The design should be developed using concepts that apply to any soft-core multiprocessor system.

More specifically, this research has two main objectives:

1. An architecture will be proposed that maintains the performance benefits of ISEs while reducing implementation area and power consumption to lower device costs.
2. The system will be fully evaluated to assess both the benefits and drawbacks concerned with run-time, area, and power. The solution will be compared to previous works to ensure it is original and makes a contribution to custom computing.

1.3 Previous Work

Previous works have defined algorithms to select custom instructions for an application or application domain [**22**] [**23**] [**24**] [**25**] [**26**]. These algorithms often require parameters regarding register file constraints to enumerate possible custom instruction candidates. Then, based on performance benefits and frequency of use, candidates are selected for custom instruction implementation. Some algorithms aim for smaller instructions with greater frequency of use, while others try to find the largest grouping possible assuming this will produce the greatest performance benefit. Chen et al [**24**] and Yu and Mitra [**25**] focus mainly on the speed of the algorithm in an attempt to reduce design time.

Algorithms to select shared ISEs for multiprocessor systems are not within the scope of this research; however, future work is needed in this area to define an algorithm specific to multi-core systems. It would also be beneficial for algorithms to be developed capable of selecting custom instruction candidates across multiple similar applications. To date, few of the previous works have delved into multi-core systems and none have proposed architectures to share custom instructions.

Previous works by Lin and Fei [**27**], Dinh, Chen, and Wong [**28**], and Zuluaga and Topham [**29**] developed methods for sharing logic between more than one custom instruction for a single processor. In the directed acyclic graph representation of a sample program in Figure 7, the operations depicted on the left and the operations depicted on the right share common logic, the multiplier. The instruction in the center has an added multiplexor to combine the two sets of operations, reducing the overall implementation area.

These works differ from the proposed solution in that this is designed for sharing operations within a custom instruction unit for a single processor system. Instead, it is proposed that entire instructions be shared between multiple processors.

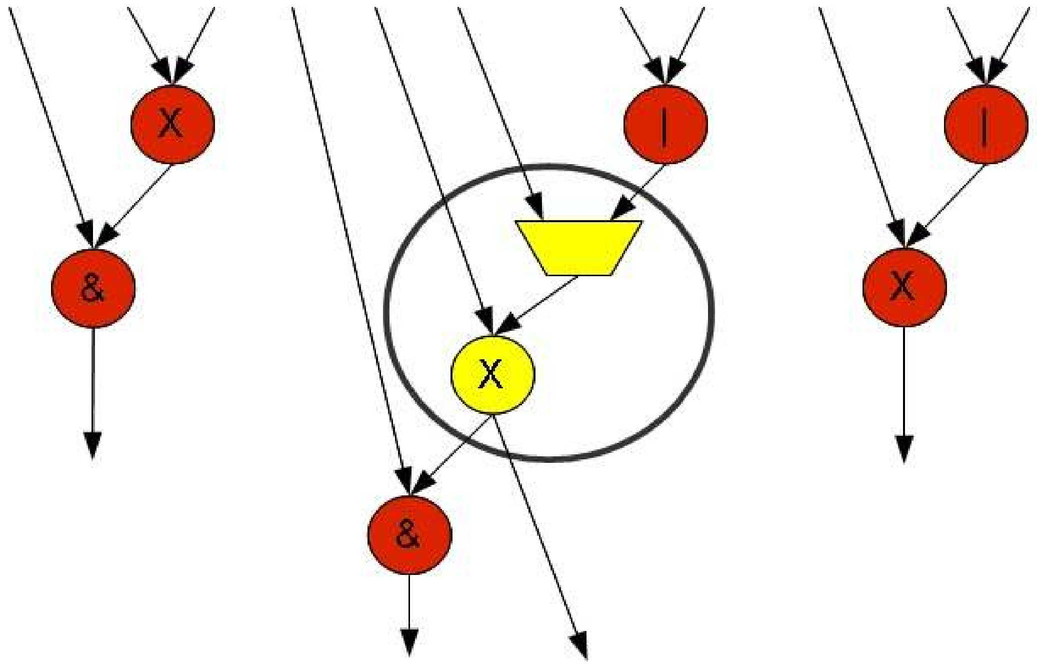


Figure 7. Shared logic between multiple custom instructions.

Lam and Srikanthan [**30**] developed an efficient method for determining the area of custom instruction candidates before implementation. Results show that the area estimates are within 8% of the actual implementation size. The usefulness of this algorithm is its ability to rule out costly custom instruction candidates thus leaving space for other, more area-efficient, instructions. Future work could have this solution to the area constraint problem combined with the proposed approach to provide even greater area savings.

Two other works [**31**] [**32**] propose architectures for sharing fabric and sharing fixed blocks, respectively, among multiprocessor systems. These papers and how they differ from the proposed solution to the area constraint and performance problems are described in more detail in Sections 1.3.1 and 1.3.2.

1.3.1 Shared Fabric

Chen and Mitra [**31**] propose an architecture for multi-core systems in which more than one core can share a programmable fabric for implementing custom instructions. In doing so, cores that can benefit from a large number of custom instructions can have a greater share of the fabric. This work focuses on improving performance through better system organization. It does not reduce the overall area, but simply makes better use of the available space.

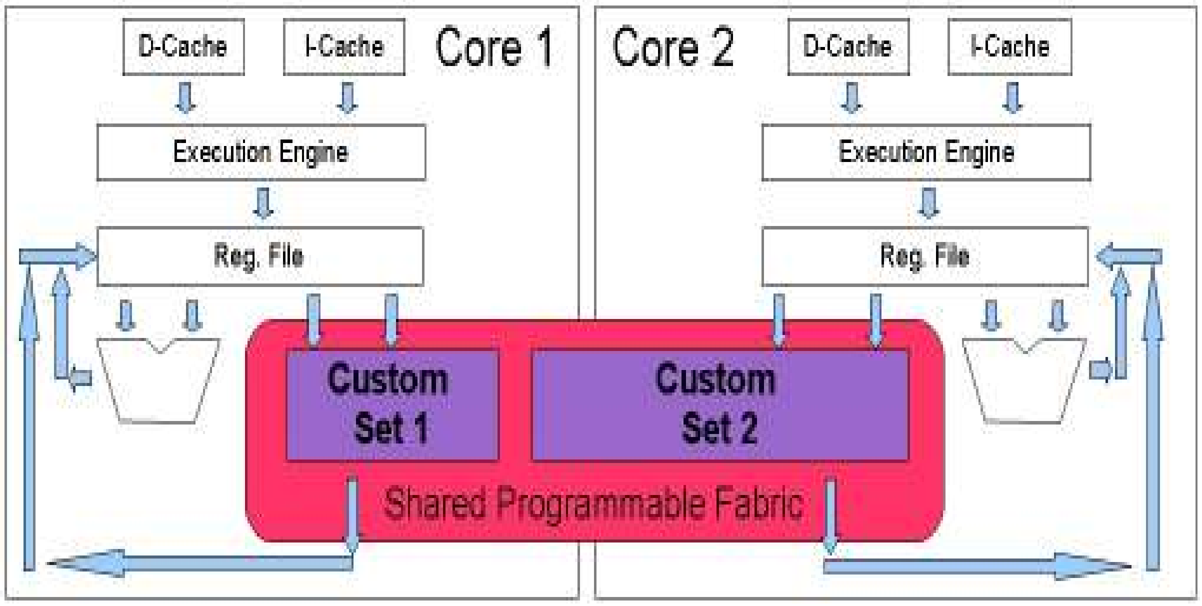


Figure 8. Shared programmable fabric architecture.

Consider an example non-shared, two processor system where each core may place up to four custom instructions on its own programmable fabric. For some applications, it may be beneficial for one processor to implement three custom instructions and the other to implement five custom instructions. This problem is solved by sharing the fabric as shown in Figure 8. Although each processor can still only access its own custom instructions, space for eight custom instructions can now be divided unequally among the processors.

This approach differs from the proposed method in that it has two systems sharing space for their custom instructions rather than having actual custom instructions shared.This approach does not reduce the size of the system, but rather it allows one core to make use of space unused by other cores.

1.3.2 Shared Fixed Blocks

Another method for reducing resource-usage in multiprocessor systems was explored by Sheldon et al [**32**]. In this paper, fixed hardware blocks are shared, or "conjoined" among multiple processors to minimize area in FPGAs. Results showed that, for two processor systems, area can be reduced by an average of 16% while causing less than 1% cycle-count overhead. The focus of this work was strictly on fixed hardware blocks such as multipliers and barrel shifters.

To select the combination of shared and non-shared fixed hardware blocks for implementation, the knapsack problem as described by Kellerer et al [**33**] is used. The knapsack problem describes an area constraint and a set of implementations greater than the allowed area. An algorithm is used to select which combination will produce the most benefit by being implemented in the constrained area.

To be more specific, this is a disjunctively constrained knapsack problem, meaning that if one choice is selected for implementation, others may be eliminated. For example, choosing to implement an individual multiplier for core 1 and an individual multiplier for core 2 can eliminate the option of having a shared multiplier between cores 1 and 2.

Figure 9 is a block diagram of the shared fixed hardware block architecture. It is similar in concept to the ideas presented in this thesis, however, fixed functional units are shared rather than custom instructions. Further, the proposed architecture is never synthesized or physically implemented on an FPGA. This means that the area results provided are an estimate based on the removal of individual blocks. Also, the additional sharing hardware such as multiplexors and an arbitration unit are not described in detail or estimated in size.

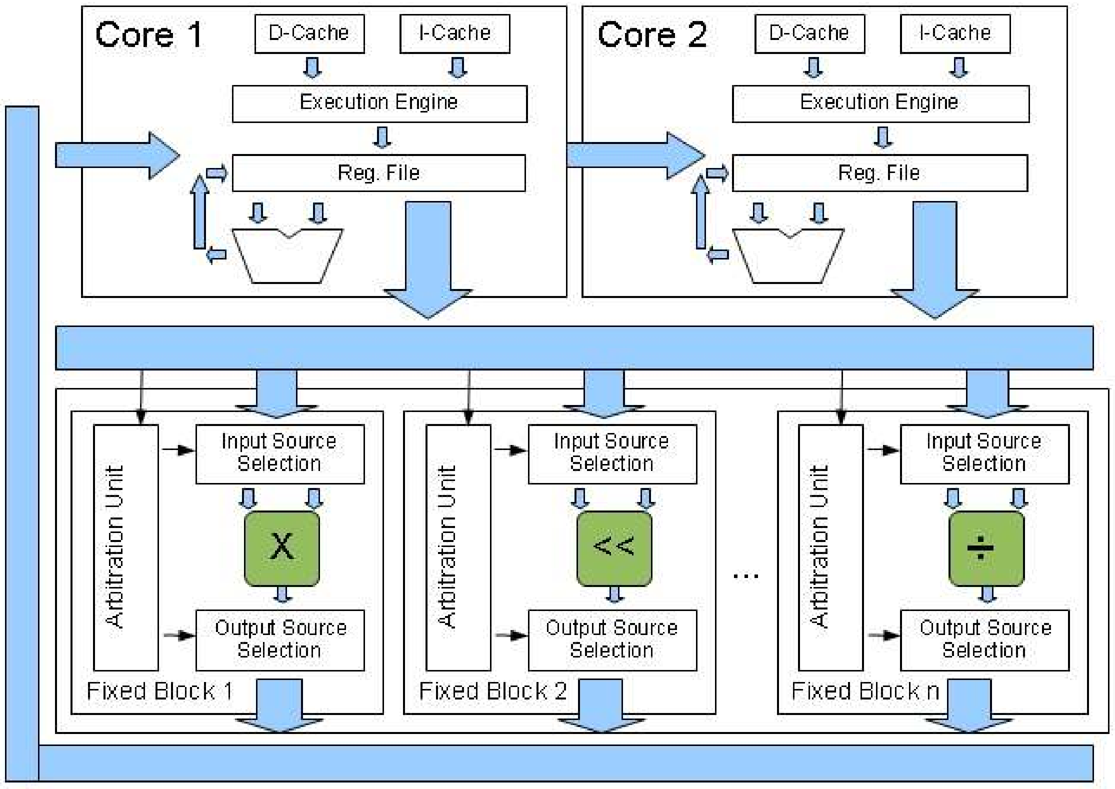


Figure 9. Shared fixed block architecture.

The work presented in this thesis differs from previous work in that it focuses on sharing ISEs. Although extra hardware is required for arbitration and signal selection in a system with sharing, the area overhead is overcome through the reduction in custom instruction implementations. Aside from reducing the required FPGA size, the advantage to resource sharing for ISEs is that the area made available can be used to implement additional custom instructions thus leading to further performance gains. This thesis clearly outlines the hardware overhead required for custom instruction sharing and provides run-time results that take any critical path penalties into account.

1.4 Thesis Organization

This thesis is organized into five chapters. Chapter 1 has outlined the motivation for this work, research objectives, and previous work. Chapter 2 describes the shared instruction architecture. It discusses two types of custom instructions and provides resource usage and critical path delay results. A simulator used to determine run-time operation and results is presented in Chapter 3 along with a benchmark generator used to quickly explore the design space. This simulator is verified with a case study in Chapter 4. Critical path delay and resource usage results are provided for the case study. Chapter 5 is a summary of all results and conclusions. Future work stemming from this research is also discussed.

2 Chapter: Shared Instruction Architecture

In this chapter, the shared instruction architecture is presented. A description of the proposed solution is provided in Section 2.1. Section 2.2 gives a general overview of the architecture. It describes the flow, from a processor's request to use the custom instruction, until the result is produced. In Section 2.3, finer details of the implementation are presented. The two types of custom instructions, fine-grained and coarse-grained, are introduced, and a design example is analyzed for resource usage and critical path delay in Section 2.4. A summary of all findings and key points from this chapter is discussed in Section 2.5.

2.1 Proposed Solution

To satisfy area and power constraints while maintaining increased performance, an architecture is proposed in which ISEs are shared between two or more cores. As shown in the two-core example in Figure 10, each processor core retains its own arithmetic logic unit (ALU), but shares a set of ISEs. In the case that a custom instruction is simultaneously requested by more than one core, the requests are queued and then executed consecutively via an arbitrator. A small amount of additional hardware is added to each processor to map the inputs and outputs to and from the correct core.

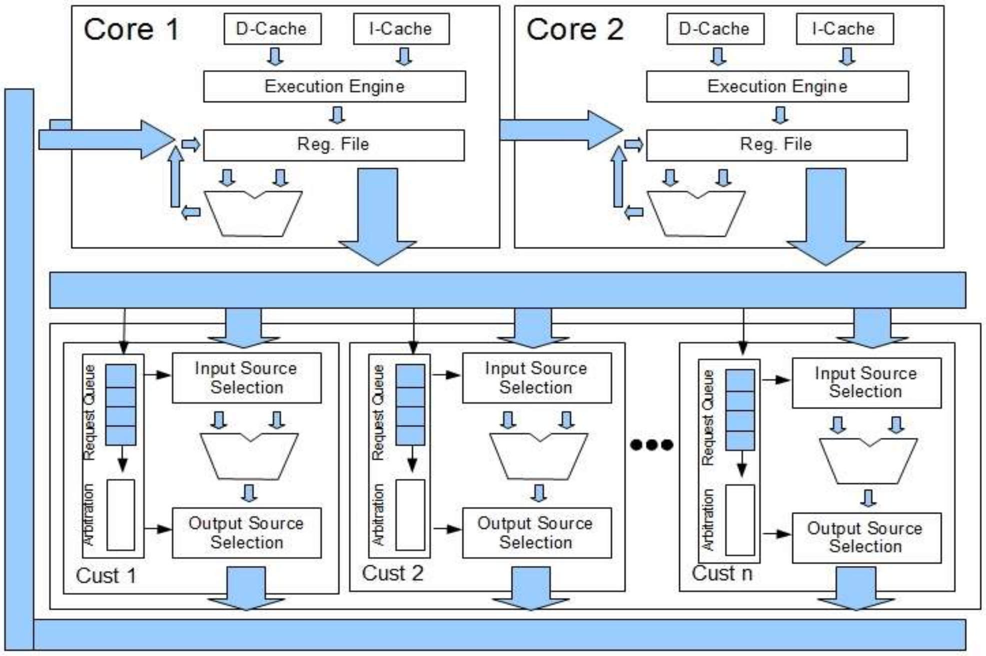


Figure 10. Proposed architecture with instruction-set extensions shared between two cores.

ISE sharing can benefit the system in one of two ways depending on the relative importance of cost versus run-time performance. First, the resources made available by allowing custom instruction sharing can be used to add additional custom instructions, thus leading to further run-time performance gains. Alternatively, a smaller FPGA can be used to implement the same design, if custom instructions are shared, thus reducing per part cost and static power dissipation.

In order to take advantage of custom instruction sharing, several challenges must be addressed. First, an architecture must be defined that provides run-time instruction sharing without significantly increasing critical path delay. There must be a way to evaluate each system and a way to test its limits under a variety of run-time conditions. Second, there must be compiler support such that ISEs are selected to maximize performance and to reduce implementation area through sharing opportunities.

The research presented in this thesis focuses on the first of these challenges. More specifically, an architectural framework is proposed in which custom instructions are shared between processors. This framework is implemented using Nios-II soft processors in a shared instruction-memory arrangement. The system was specified in a hardware description language (HDL) in a scalable way such that hardware can be easily generated for any number of processors, any degree of processor sharing, and any number of custom instructions. To the best of the author's knowledge, this is the only work that addresses the sharing of custom instructions between processors.

2.2 Implementation Overview

The proposed architecture allows processor cores to access a shared set of custom instructions, as was shown in Figure 10. In the normal operation of a single core, operands from the register file can be directed to either the ALU or the custom instruction unit. In the proposed architecture, inputs to the custom instruction unit are multiplexed to allow for operands to be provided by multiple cores. Multiplexing is controlled by an arbitration unit that also controls to which core the output is forwarded.

The arbitration unit is comprised of priority selection circuitry and a mechanism for queuing cores. When multiple cores request the use of an instruction simultaneously, the arbitrator determines which core will have access to the custom instruction based on a fixed priority scheme. All other processors in the queue are stalled until the instruction is released, after which the core with the next highest priority is allowed to proceed.

This architecture is designed for systems where multiple processors are running the same or similar applications. This allows for custom instructions to be selected that can be used by and can benefit all the processors in a system. The main goal is to reduce the overall FPGA implementation area; a reduced implementation area requires a smaller, less costly FPGA. Alternatively, the area saved by sharing could be used to implement additional custom instructions that will further benefit the performance of the system. While trying to improve the density of the design, it is important that run-time performance is still improved over a system that does not use custom instructions to execute the application. In Chapters 3 and 4, run-time results are provided for the proposed architecture for fine-grained and coarse-grained ISEs, respectively. A comparison is made to the run-time of architectures that do not share custom instructions.

2.3 Implementation Details

The proposed architecture was implemented for a Stratix III FPGA using Nios-II/s soft processors. According to the Nios II Processor Reference Handbook [**34**], each Nios II/s core uses less than 700 adaptive logic modules (ALMs) of the 56,800 ALMs available on the Statrix III device. An ALM is the basic building block of programmable hardware in which the digital logic on some FPGAs is built. ALMs can have up to eight inputs and eight outputs used in several implementation combinations. Resource usage results will be determined in units of ALMs so that comparisons can be made fairly between all systems.

The Nios II/s core uses static branch prediction and can have up to a 64KB instruction cache. No data cache is available for this core. The core uses a five stage instruction pipeline comprised of *fetch*, *decode*, *execute*, *memory*, and *writeback*. During the *fetch* stage, instructions are retrieved from memory and the next program counter value is predicted. The *decode* stage then takes place to read the register file and create the datapath control signals. Next, the *execute* stage performs the decoded instruction. When the *execute* stage has finished, data is written to memory during the *memory* stage. Finally, the register file is updated in the *writeback* stage. Figure 11 depicts the new five-stage pipeline for a shared custom instruction-set architecture.

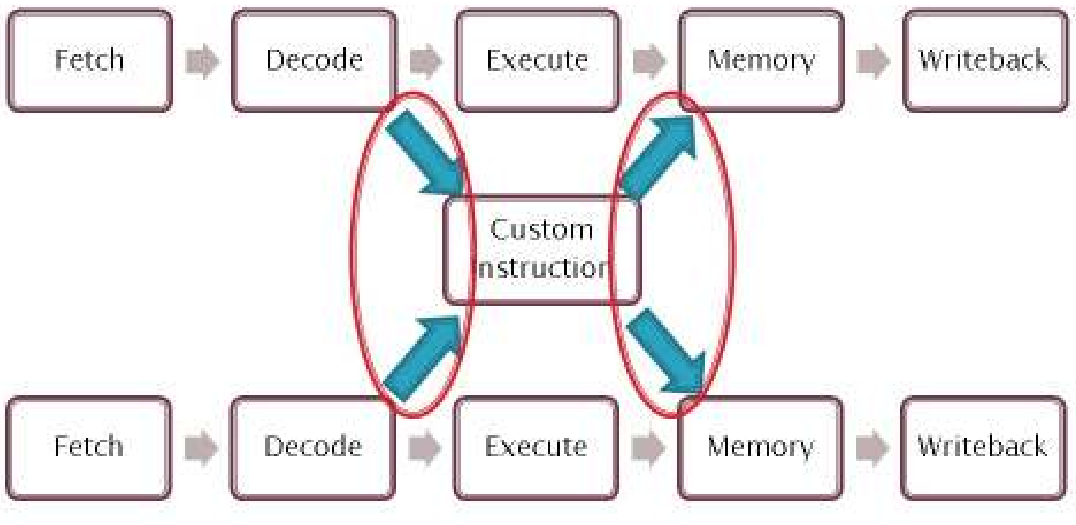


Figure 11. Shared custom instruction-set extension pipeline.

As for a single processor system, the custom instruction hardware replaces the regular execute stage performed by the arithmetic logic unit (ALU). For the proposed architecture, each processor retains its own ALU, but shares custom instruction units. The encircled areas on the figure represent the additional hardware required to implement sharing, which is the same regardless of the custom instruction unit. The custom instruction unit is like a "black box" that can be replaced with different instructions for each application.

In the proposed sharing architecture, all standard Nios II multi-cycle custom instruction signaling is maintained. These include the instruction *request ID*, two input operands, the *result* signal, the *start* signal, and the *done* signal. The *request ID* indicates which instruction the processor would like to execute. The *start* signal indicates a request has been made by the processor to execute a custom instruction. The *done* signal indicates that the custom instruction has finished executing and a valid result is available. The *start* and *done* serve as handshaking controls thus allowing instructions to take multiple clock cycles to execute.

A multi-cycle custom instruction timing diagram is shown in Figure 12. The *start* signal is set high to indicate the processor's request for the custom instruction. At this time, the inputs must be valid and remain constant until the end of the custom instruction. When execution is complete, the *done* signal is set high by the custom instruction to indicate a valid *result* is waiting for the processor. The instruction *request ID* must also remain constant for the duration of the instruction. The processor uses this signal to select the correct *result* for the extended case where there are multiple instruction options.

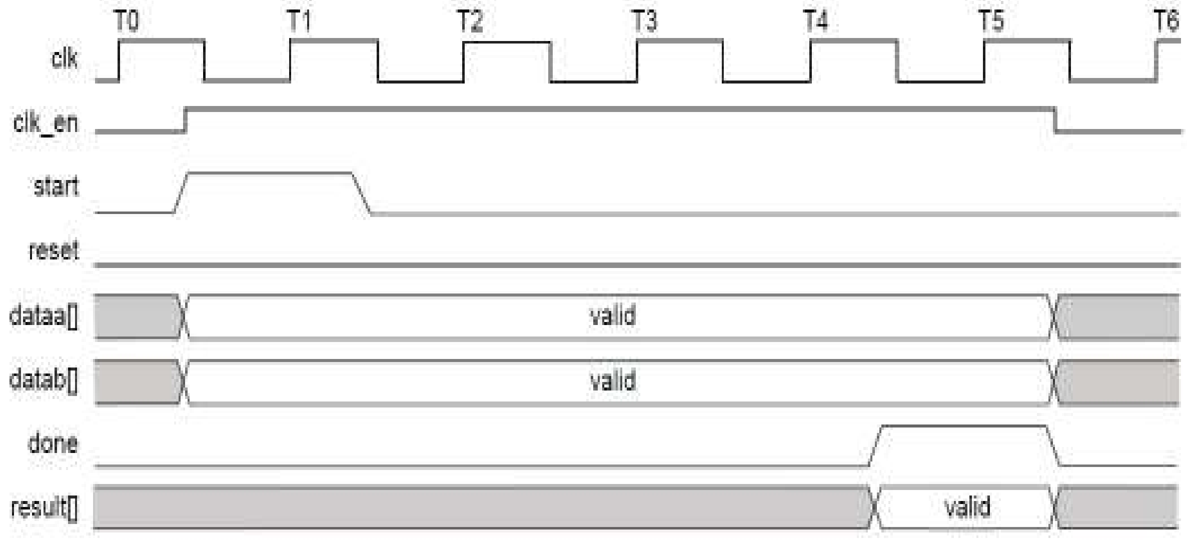


Figure 12. Multi-cycle custom instruction timing diagram [35].

2.3.1 Front-End Sharing Hardware

The front-end of each custom instruction contains data multiplexers, a priority-encoder, and hardware to latch start signals from all processors. Because processors normally only hold the *start* signal high for one clock cycle as shown in Figure 12, the request must be queued thus preventing the request from being lost if the instruction is currently in-use. The *start* signal is only saved for a particular custom instruction if the processor's instruction *request ID* signal matches the instruction's id. This is performed by a comparator as shown in the block diagram depicting the front-end hardware in Figure 13. The saved *start* signal is released when the corresponding *done* signal matching the processor and instruction is received.

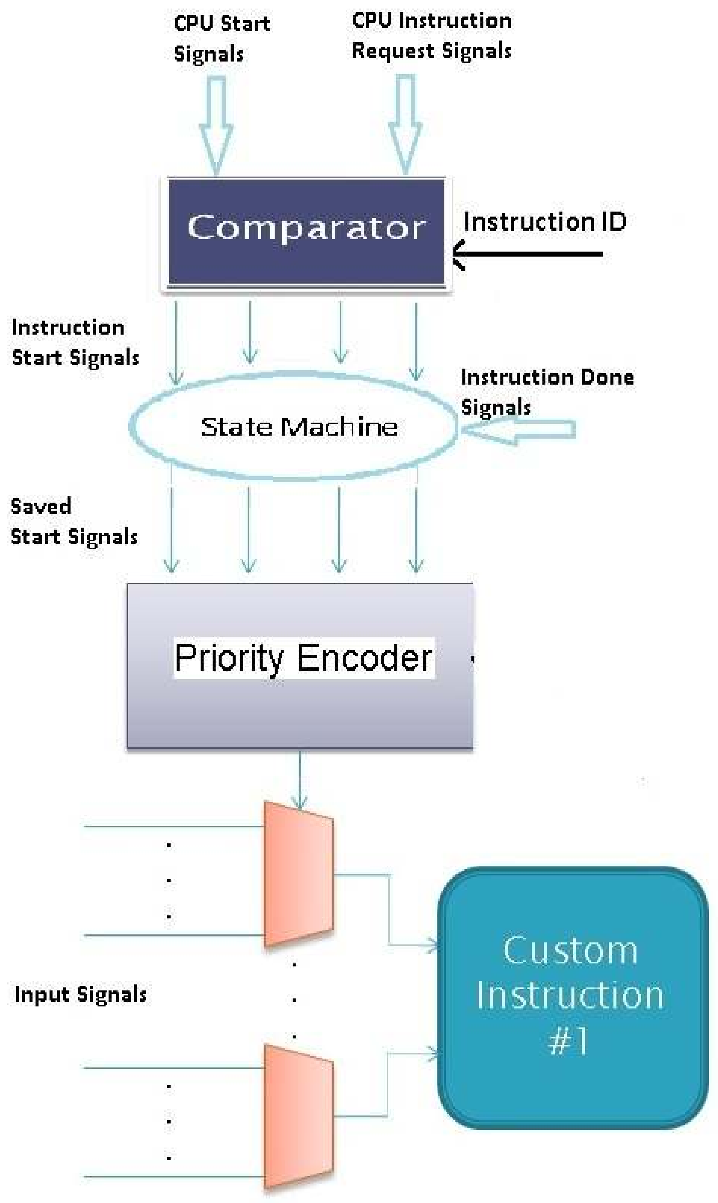
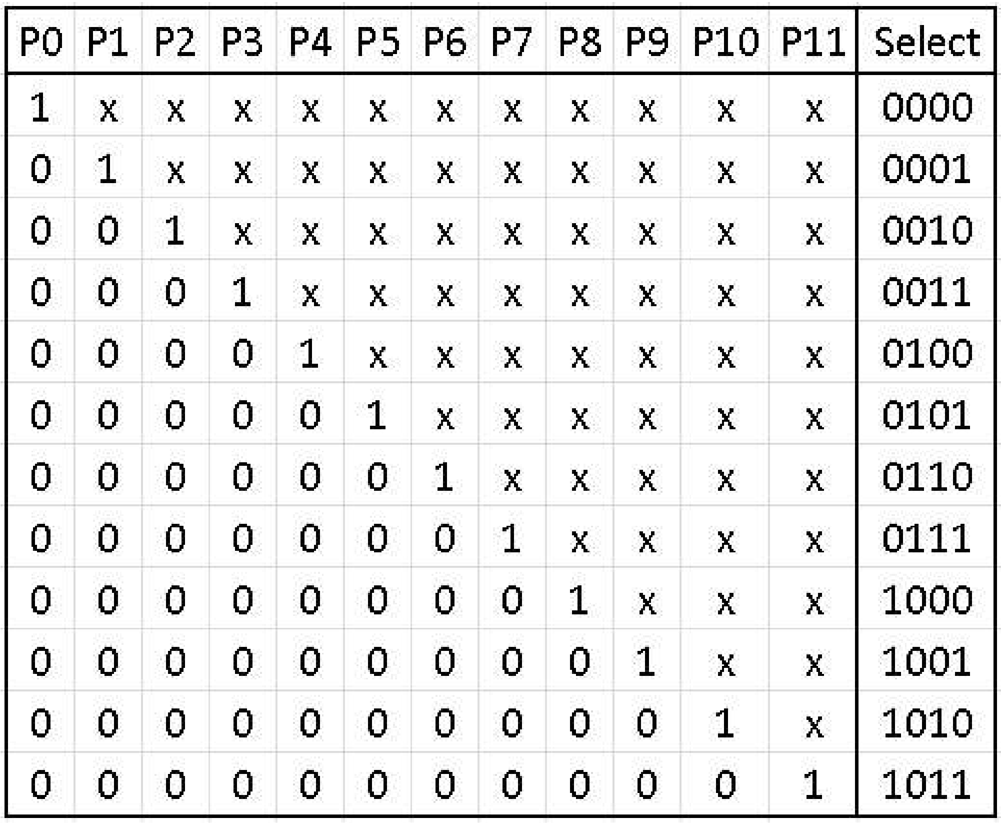


Figure 13. Front-end sharing hardware block diagram.

The saved *start* signals from all processors are connected to a priority encoder. A one-hot to binary encoder is used to give precedence to one processor when there are multiple requests. The truth table for such an encoder is given in Table 1 for a system with 12 processors. The output, or *select* signal, is determined based on the saved *start* signals from each processor, P0 through P11. If multiple processors are requesting the instruction, the processor with the lowest ID is given priority, and all other signals are ignored. The output from the encoder selects the correct input data which is fed into the custom instruction. This entire front-end set of components is required for each custom instruction to provide a fast and area-efficient mechanism for arbitration.

Table 1. One-hot to binary priority encoder truth table for a 12 processor system.



P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	Select
1	x	x	x	x	x	x	x	x	x	x	x	0000
0	1	x	x	x	x	x	x	x	x	x	x	0001
0	0	1	x	x	x	x	x	x	x	x	x	0010
0	0	0	1	x	x	x	x	x	x	x	x	0011
0	0	0	0	1	x	x	x	x	x	x	x	0100
0	0	0	0	0	1	x	x	x	x	x	x	0101
0	0	0	0	0	0	1	x	x	x	x	x	0110
0	0	0	0	0	0	0	1	x	x	x	x	0111
0	0	0	0	0	0	0	0	1	x	x	x	1000
0	0	0	0	0	0	0	0	0	1	x	x	1001
0	0	0	0	0	0	0	0	0	0	1	x	1010
0	0	0	0	0	0	0	0	0	0	0	1	1011

2.3.2 Back-End Sharing Hardware

When a *result* is produced by a custom instruction, it must be forwarded to the correct processor. The processor's instruction *request ID* signal is used to select the priority encoder output, *done* signal, and *result* signal from the corresponding instruction. As shown in Figure 14, the priority encoder output and the processor identification number are then compared. If these values match, the *done* signal is accepted by using the *and* gate shown. Upon receiving a high *done* signal, the processor will then latch the *result* value present. This ensures that a processor does not erroneously select a result belonging to another processor with higher priority. This entire back-end set of components is required for each processor core.

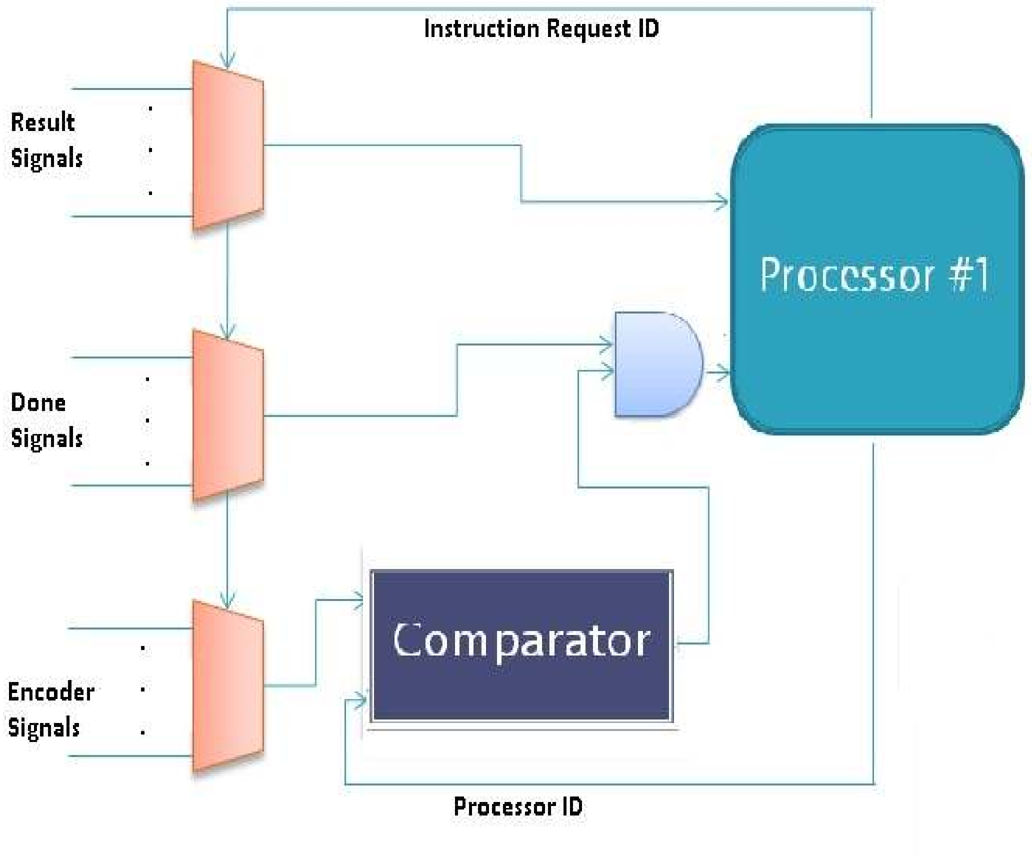


Figure 14. Back-end sharing hardware block diagram.

2.4 Custom Instructions

As shown in Figure 13, the custom instruction unit is a “black box” within the shared architecture. Custom instructions can be tailored per application either manually or automatically while the infrastructure used to support sharing remains the same. Two types of custom instructions can be implemented: fine-grained and coarse-grained. Fine-grained instructions are made from smaller sections of source code and use few inputs. They usually execute in just a few clock cycles and are closely coupled with the pipeline. Coarse-grained instructions replace large sections of source code and can use a whole array of inputs. Typically, a system with fine-grained instructions will spend less time in each custom instruction, but use them more frequently than coarse-grained instructions. Examples of fine-grained instructions along with an implementation example are given in Section 2.4.1. Coarse-grained instructions are described in Section 2.4.2 with an explanation of the required architecture adjustments.

2.4.1 Fine-Grained Instructions

Typically, fine-grained instructions are selected from application source code based on register file constraints, speed increase, frequency, and hardware implementation cost. As mentioned previously, several algorithms to define and select custom instructions for single-processor systems have been developed. Automatic instruction selection and enumeration is beyond the scope of this thesis and, as a consequence, custom instructions are chosen manually.

Custom instructions were generated with attributes found to be typical based on an analysis of well-known benchmarks. The instructions were not extracted from source code, but rather generated using compiler instruction-enumeration and -selection statistics from several previous works. The assumptions used are tabulated in Table 2 along with the source used to determine their value. Each of the entries in the table represents an architectural attribute of the custom instruction such as the "depth" of logic in a custom instruction in terms of gates.

Table 2. Attributes of custom instructions assumed when generating the custom instruction-set.

Statistic	**Value**	**Source**
Average Custom Instructions per Application	9	Ienne and Leupers [**36**]
Average Frequency of Custom Instruction	32.3%	Clark [**37**]

Relative Frequency of Operations	Multiply = 13.3%
	Add = 31.0%
	Logic = 55.7%
		Clark et al. [**23**]

Custom Instruction Depths, Width, and Shape	Depth 2 = 47.5% Depth 3 = 24.8% Depth 4 = 10.3% Depth 5 = 6.2%	Clark et al. [**23**]

One assumption used to generate the custom instruction-set is that custom instructions tend to have a “triangular” shape as depicted in Figure 15. Statistically, custom instructions can be organized as a set of logic levels with a “triangular” shape as described by Clark et al [**23**]. In this diagram, each node represents an operation with the outputs of each level of operations being connected to the inputs of the next level via a cross-bar network. The level closest to the inputs tends to contain the most operations. The number of operations in each consecutive level tends to be progressively fewer the closer it is to the output. The custom instruction in Figure 15 has four logic levels with the first level performing three operations that use the two input operands. The results from this first level are then used as operands to the second level which performs two operations. The third level then also performs two operations, and the fourth level performs one operation that produces the output. This overall shape is assumed for the set of custom instructions defined and depicted in Figure 16.

Assuming the “triangular” shape assumption, 9 custom instructions were randomly generated to define the custom instruction-set shown in Figure 16. The decision to use 9 custom instructions was based on previous work by Ienne and Leupers [**36**] that found that the average number of custom instructions per benchmark application is 8.72. Clark et al [**23**] found that the probabilities that custom instructions have depths of 2, 3, 4, and 5 levels are 47.5%, 24.8%, 10.3%, and 6.2%, respectively, for systems in which 5-levels is the maximum allowed depth. Based on this, we assumed that five of the nine proposed instructions have 2 levels, two have 3 levels, one has 4 levels, and one has 5 levels.

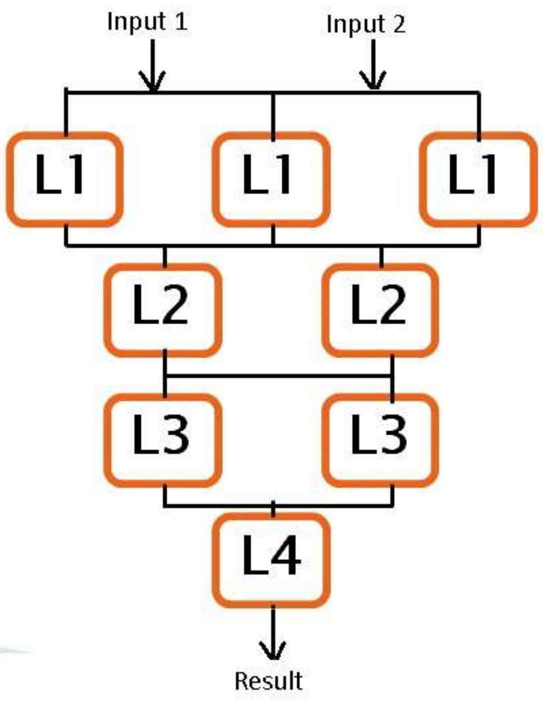


Figure 15. Average custom instruction shape and layout.

After further processing of the statistics extracted from Clark et al [**23**], it was determined that depth-2 instructions have an average of 3.7 operations, depth-3 instructions have an average of 5.3 operations, depth-4 instructions have an average of 7.0 operations, and depth-5 instructions have an average of 8.3 operations. For this reason, it was assumed that the generated custom instructions have 4, 5, 7, and 8 operations for depth-2, depth-3, depth-4, and depth-5 instructions, respectively.

With respect to operation types, the probabilities that an operation is a multiply, add or subtract, or a logic operation are 13.3%, 31.0%, and 55.7%, respectively [**23**]. Thus, operations were randomly placed in each of the nine proposed instructions using these proportions. The custom instructions can have one or two inputs. The Verilog implementation of the nine average custom instructions can be seen in Appendix A.

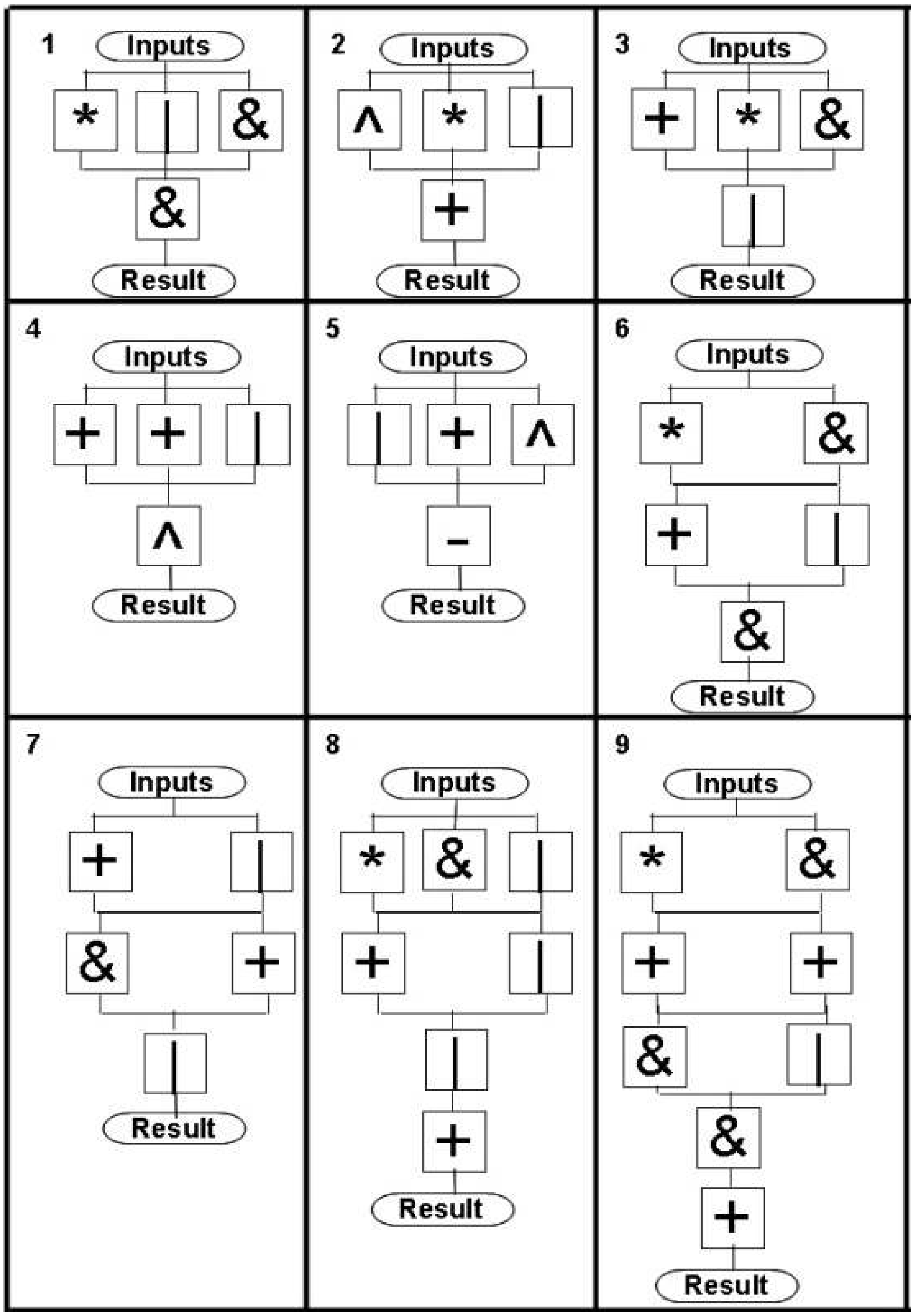


Figure 16. Set of nine custom instructions used for the proposed architecture.

2.4.1.1 Experimental Framework

The proposed architecture and fine-grained custom instructions were implemented for a Stratix III FPGA using Nios-II/s soft processors. As defined in the Nios II Custom Instruction User Guide [**35**], the proposed custom instructions are extended multicycle instructions, requiring *clock*, *clock enable*, *start*, *reset*, *instruction select*, and *done* signals. These fine-grained instructions are confined to two input operands and one output by the Nios II register file.

To determine the impact of sharing as the number of processors varies, Quartus II was used to generate systems containing varied numbers of processors using both the original, non-shared architecture and the new, shared architecture. Results for these systems are discussed in Section 2.4.1.2.

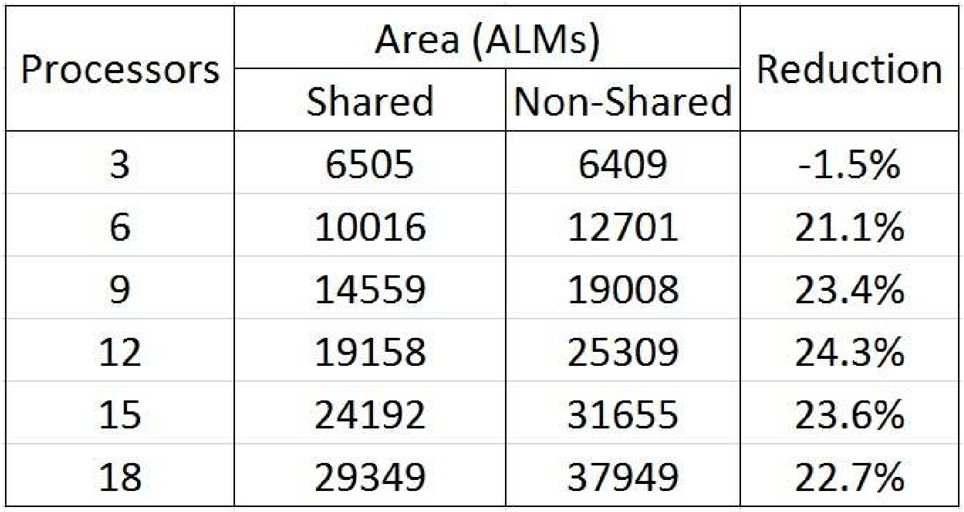
Using results from the varied processor experiment, systems with a fixed number of processors are generated that allow varying degrees of custom instruction unit sharing. This means processors can share more than one set of custom instructions in order to alleviate bottlenecks caused by competition for the same resource. These results are discussed in Section 2.4.1.3.

For the varied processor and varied sharing systems described above, all cores have their own 8 Kb instruction cache and 6 Kb on-chip data memory. The cores share a common 256 Kb on-chip instruction memory. Synthesis was performed using Quartus II which was used to provide both resource usage and critical path delay results. All area and critical path delay results have been averaged over five fitter seed values.

2.4.1.2 Varied Processor Resource Usage and Critical Path Delay

Quartus II was used to generate systems containing 3, 6, 9, 12, 15, and 18 processors that all share one set of custom instructions. Table 3 shows the percentage area reduction of the shared system compared to the non-shared architecture for each case. The shared systems containing 6, 9, 12, 15, and 18 processors all showed an area reduction of more than 21% over the non-shared architecture containing the same number of processors. The shared three processor system did not produce any area benefits over the non-shared architecture. These results demonstrate two important points to consider when designing a shared system: a minimum amount of sharing is required before area reduction is seen, and a maximum amount of sharing can be reached where the area benefits begin to decrease again.

Table 3. Resource usage for fine-grained systems sharing one custom instruction unit.



Processors	Area (ALMs)		Reduction
	Shared	Non-Shared	
3	6505	6409	-1.5%
6	10016	12701	21.1%
9	14559	19008	23.4%
12	19158	25309	24.3%
15	24192	31655	23.6%
18	29349	37949	22.7%

The three processor system demonstrates a case where the minimum amount of sharing was not reached to produce an area reduction over the non-shared architecture. Although two custom instruction units are removed for this case, the additional infrastructure needed to provide custom instruction sharing (i.e. multiplexors, comparators, priority encoders, and state machines) is greater than the area saved.

The most prominent reduction in area is achieved for the 12 processor system representing the maximum amount of sharing for this instruction-set before the area benefits begin to decrease again. This peak is likely caused by two opposing trends. One trend is that an increase in the amount of sharing, achieved by increasing the number of processors, results in an increase in resource usage because it leads to larger multiplexors and a larger arbitrator. The opposite is true for the resource usage of the custom instructions themselves; the more sharing that occurs, the less resource usage. The amount of sharing that will lead to an optimal reduction in area will vary depending on the custom instruction unit size and complexity.

Although reducing the implementation area of a design is the main advantage of sharing, it can have a negative impact on critical path delay and thus overall runtime. A more compact design generally means a shorter critical path delay. However, in the shared architecture, the placement of the instruction unit in relation to each processor will be much different than in the non-shared architecture. Also, the additional sharing hardware lengthens the time required to complete the *execute* stage of the pipeline. For both the shared and non-shared architectures, it is desirable to create ISEs that do not affect the critical path delay of the system.

An extended path over the custom instruction will lengthen the time required to complete the execute stage of the processor's pipeline. Figure 17 shows the five-stage pipeline of the Nios II/s processor. For the fine-grained custom instructions, the execute phase has been split into two clock cycles to ensure the critical path of the entire design is not affected by larger instructions.

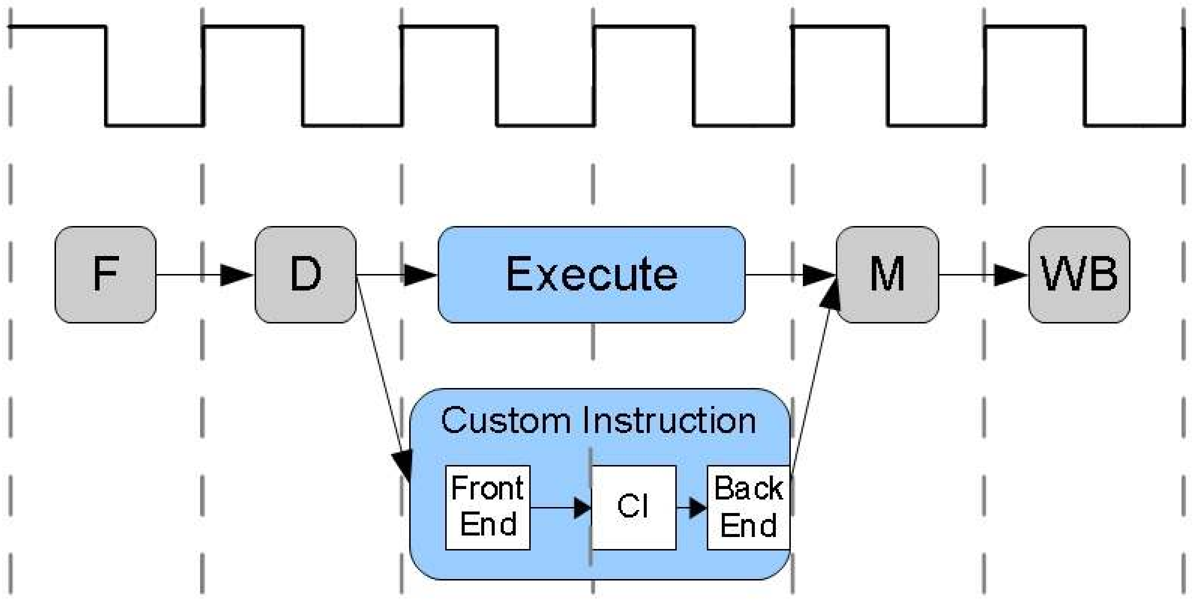


Figure 17. Execute phase timing for custom instruction.

The first cycle in the execute stage performs the front-end operations that select the input values. These values are then latched, and the actual custom instruction is performed in the second clock cycle of the execute stage. The back-end operations, that select the *result* and *done* signals for the correct processor, are also performed during the second clock cycle. For more complex instructions, the custom instruction could be split into more than two clock cycles to prevent the instruction from affecting the critical path if necessary.

The critical path of the shared architecture is not affected when using the two-cycle execute as demonstrated in Table 4. Negligible changes in the critical path delay over the non-shared architecture are seen in the 6, 9, 12, 15, and 18 processor cases. These small differences are likely caused by variations in the layout due to the non-deterministic nature of the layout algorithms used by Quartus II. The 3 processor case shows a larger reduction in the critical path, but was not a beneficial implementation in terms of area reduction. Since the 12-core system showed the greatest area reduction and a slight reduction in critical path delay over the non-shared architecture, it was selected for further analysis. The 12-core system also presents several opportunities to explore custom instruction unit sharing between varied numbers of processors.

Table 4. Critical path delay for systems with varying cores sharing one custom instruction unit. Each custom instruction unit has 9 fine-grained instructions.

Processors	Critical Path Delay (ns)		Reduction
	Shared	Non-Shared	
3	7.335	7.757	5.4%
6	8.229	8.361	1.6%
9	9.220	9.242	0.2%
12	9.021	9.094	0.8%
15	9.958	9.778	-1.8%
18	10.082	10.094	0.1%

2.4.1.3 Fixed Processor Resource Usage and Critical Path Delay

Systems containing a fixed number of processors can be arranged into groups, such that each group shares a set of custom instructions. In Figure 18 and Figure 19, FPGA resource usage and run-time are presented for a 12-core system arranged into different sharing configurations. The total number of processor cores in the system remains constant, but the number of custom instruction units varies depending on the degree of sharing.

For example, one arrangement has two cores per custom instruction unit, where each unit contains the 9 custom instructions shown in Figure 16. For this system, the twelve cores are evenly divided into six groups. In this thesis, this type of arrangement is referred to as *2-way*.

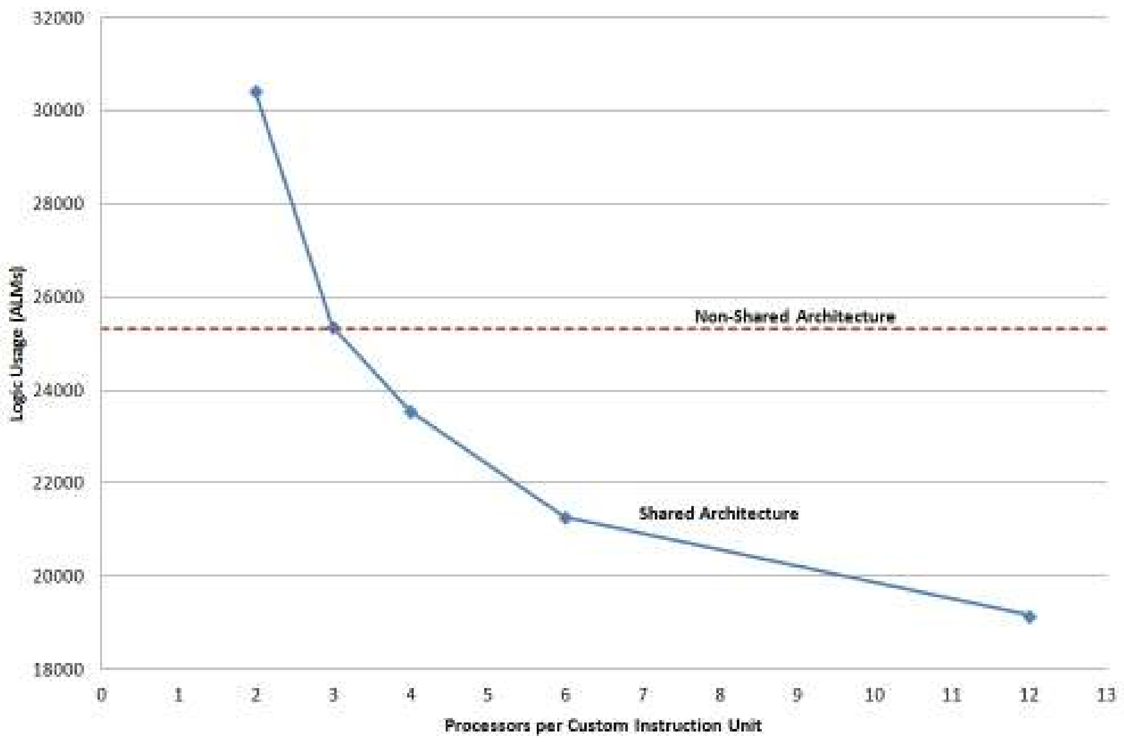


Figure 18. Resource usage for a system with 12 cores and with varying degrees of sharing. Each custom-instruction unit has 9 fine-grained instructions.

For the area and critical path delay results, the x-axis represents the number of processors sharing each custom instruction unit. The graphs show a magnified view of the results to better demonstrate the changes in the y-axis values for each case. For the 4-way, 6-way, and 12-way sharing configurations, significant reductions in resource usage are seen. In fact, the resource usage of a 12-core system is reduced from 25,309 ALMs in the non-shared architecture to 19,158 ALMs in the 12-way system. This translates to a 24% reduction in FPGA logic resources. The 2-way and 3-way sharing configurations do not show area reductions compared to the non-shared architecture due to the additional input multiplexors, comparators, and priority encoders required. For example, a 2-way system requires 6 custom instruction units and the input multiplexors, comparators, and priority encoders for each instruction in each unit. A 12-way system only requires a single custom instruction unit and additional front-end hardware for each custom instruction in the set.

The results presented in Figure 19 show that any changes in the critical-path delay of the 12-core system are negligible and likely attributed to layout variations. Quartus II timing results showed that the additional sharing hardware did not appear on the critical path. Therefore, the slight range in values from 9.02 ns to 9.51 ns is likely caused by repositioning in the design layout and not by the additional sharing hardware. It could be expected, however, that a peak number of processors would be reached where the additional multiplexing and encoding hardware would become large enough to cause delays in the critical path.

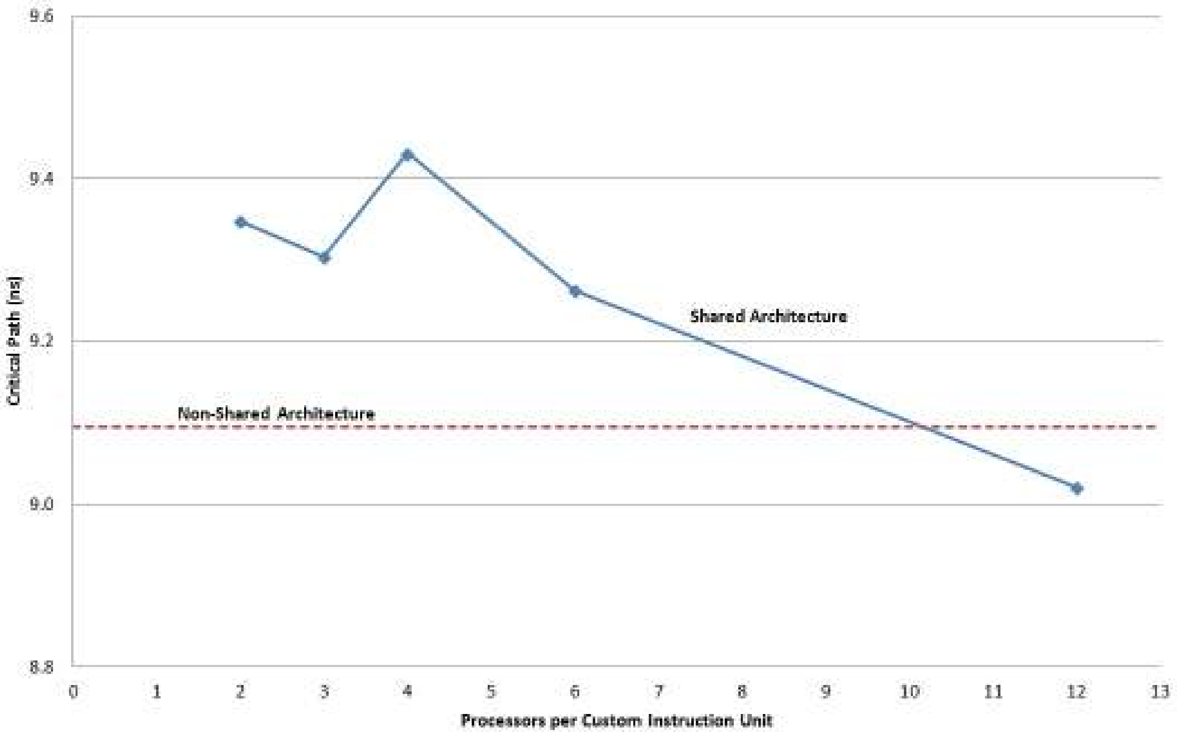


Figure 19. Critical path delay for a system with 12 cores and with varying degrees of sharing. Each custom-instruction unit has 9 fine-grained instructions.

2.4.2 Coarse-Grained Instructions

Larger instructions that require more inputs and more clock cycles to complete can also be selected from application source code for custom instruction implementation. These coarse-grained instructions are usually called less frequently by the application, but have a much larger per-use speed increase.

As mentioned previously, Nios II processors are restricted by their register files, allowing for just two inputs and one output. Figure 20 shows a shared architecture with two cores capable of using multiple inputs and multiple outputs in the custom instruction. It is similar to the architecture in Figure 10 but has a few adjustments for input selection and result writeback.

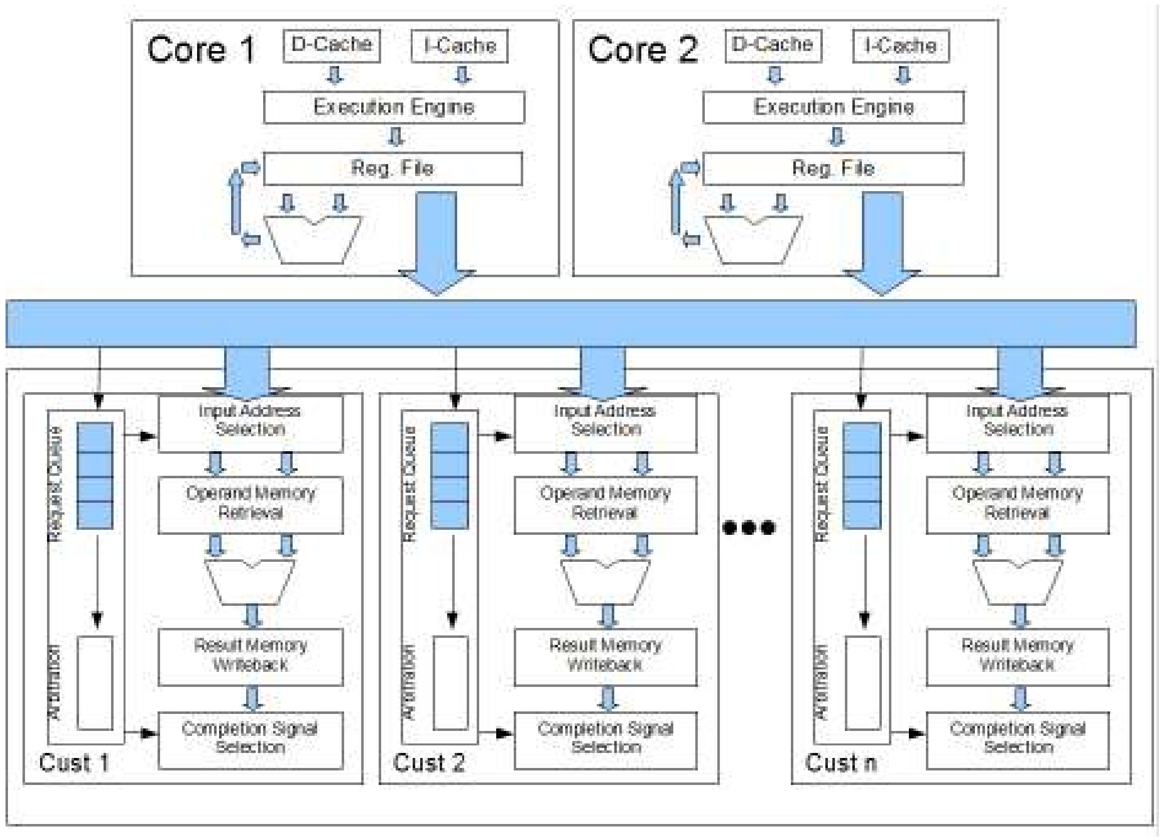


Figure 20. Proposed architecture for coarse-grained instruction-set extensions shared between two cores.

Rather than providing two input operands, the register file operands can consist of a pointer to the front of an array. The values stored in the array can be transferred in bursts from memory to internal custom instruction registers and used as operands in the custom instruction. The reverse can be done for output operands, with multiple results being transferred directly to memory rather than being returned via the register file. Effectively, the custom instruction behaves like a co-processor by taking control of the data bus to perform memory read and write operations.

From this point, the custom instruction proceeds as normal. When the custom instruction is completed, the outputs can be written back to memory in any location specified. No result is passed back to the processor, as it has already been written to memory. The arbitrator is used to select the correct *done* signal that will notify the processor that the custom instruction has completed. A coarse-grained custom instruction case study is given in Chapter 4.

2.5 Summary

The architecture for sharing sets of custom instructions was introduced in this chapter. The sharing architecture has hardware overhead caused by multiplexors, an encoder, and an arbitrator, that grows with increased sharing. The area reduction from sharing instruction units must be enough to overcome this additional hardware. The new architecture maintains all signals required for custom instruction operation as defined in the Nios II Custom Instruction User Guide [**35**] and meets all signal timing requirements.

An example system containing twelve processors was developed for both fine-grained and coarse-grained custom instruction sharing analysis. The fine-grained instructions used to gather results were based on statistics for average applications and custom instructions. The custom instruction-set comprised of nine custom instructions following a triangular shape. Resource usage results show that the area of a shared system can be reduced up to 24% over the original, non-shared architecture. The critical path is affected minimally and actually shows a slight decrease in the 12-way sharing case.

Finally, the architecture adjustments required to accommodate coarse-grained instruction sharing were described. The coarse-grained scenario will be analyzed with a case study in Chapter 4.

3 Chapter: Simulating Run-time Behaviour

To evaluate the run-time performance of the proposed scheme, a simulation framework was developed. This chapter details the framework which consists of a benchmark generator and a high-level run-time simulator. The simulator was designed to speed-up design space exploration and to quickly evaluate the run-time effectiveness of instruction sharing. This tool simulates the run-time interaction between processors competing for instruction resources and simulates the resulting effect on total run-time. To aid the simulator in the efficient analysis of applications and custom instructions best-suited for sharing, a benchmark generator was developed that is capable of randomly generating applications compatible with the simulator.

This benchmark generator and simulator are described in more detail in Section 3.1 and Section 3.2, respectively. Later, a case study in Chapter 4 is used to determine the accuracy of the simulator and demonstrate the effectiveness of exploring the design space using the simulator. Section 00 provides run-time results for the fine-grained system introduced previously in Section 2.4.1. Finally, processor and custom instruction utilization as the amount of sharing increases are discussed in Section 3.4.

3.1 Benchmark Generator

A benchmark generator was developed that randomly generates an execution trace based on user-specified parameters such as program length, the ratio of custom instructions to base instructions, and the number of custom instructions available. The benchmark generator allows us to efficiently determine if a certain type of application will benefit from custom instruction implementation and the shared custom instruction-set architecture. The tool can either generate unique benchmarks for each processor in the multi-core system or it can generate one benchmark to be used by all cores. The unique benchmarks represent a case where processors are running similar applications that will still make use of the same set of custom instructions.

The benchmark generator allows the user to define any custom instruction-set. First, the generator creates an application of the user-defined length. Then, the cycles required for the software implementation of a custom instruction are removed and replaced with the custom instruction cycles. For example, the first instruction from the fine-grained set shown again in Figure 21 implements four operations. These four operations require 11 cycles when implemented in software, but only 2 when implemented in hardware as a custom instruction. The 11 cycles are removed from the original application and replaced with 2 cycles that are marked as custom instruction one.

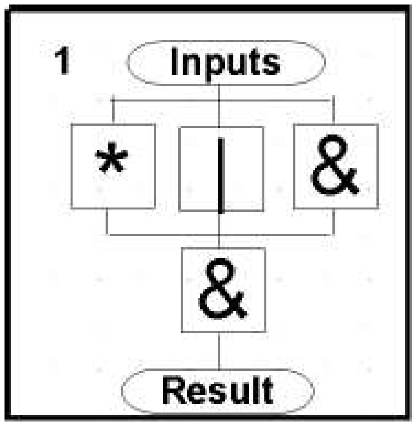


Figure 21. Four operation custom instruction requiring 11 cycles in software or 2 cycles in hardware.

3.2 Run-Time Simulator

A high-level run-time simulator was also developed that simulates the effects of processor cores competing for a set of shared instructions. The simulator is written in C++ and can work independently or in conjunction with the benchmark generator. The simulator can be operated with separate benchmark traces for each processor in the system. These traces can be generated using the benchmark generator, or they can be extracted from the run-time behavior of an application running on a real processor.

The simulator operates at a higher level than typical instruction-set simulators. The only processor state that is maintained is the program counter and the number of clock cycles remaining in the execution of the current instruction. For instruction units, the simulator maintains a record of which processors have access to its instruction resources and which processors are currently queued.

The simulator provides an output text file containing run-time statistics for all processors in addition to utilization statistics for processors and shared instructions. It details the status of each processor on every execution cycle. The cycle status shows an *r* when a regular instruction is being executed, an *s* when the processor is stalled, and a number when a custom instruction is being executed. The processor utilization statistics indicate how often a processor is stalled in relation to the overall cycle count. The custom instruction utilization statistics indicate how many cycles are spent using a custom instruction unit compared to the overall cycle count.

In Chapter 4, hardware is implemented as part of a case study, and run-time results are compared against those produced by the simulator. Results of the case study show that the behavior of the simulator closely matches that of a real system. The key benefit of the simulator is it that it accelerates design-space-exploration. Using hardware implementations to perform design-space-exploration would require each candidate to be synthesized and then programmed onto the FPGA. If one were to explore the various permutations of 9 instructions being assigned and shared amongst 12 processors, design-space-exploration using hardware could take months. Using a traditional hardware simulator such as ModelSim would be exponentially worse. The simulator developed in this thesis can estimate the impact of a proposed ISE configuration on run-time in roughly one second.

3.3 Fine-Grained Run-Time Results

The run-time results presented in Figure 22 were created by multiplying total run-time cycles generated by the simulator with the critical path delay results, shown in Figure 19, generated by the synthesis tool. These results were generated for three system types: the shared architecture with 12 cores and varying degrees of sharing, the non-shared architecture with custom instructions that are not shared between processors, and the base architecture which has no custom instructions.

Figure 22 shows run-time results for two different scenarios: all cores running the same application and all cores running different applications. When applications are generated automatically for the proposed simulation framework, it is assumed that the frequency of custom instructions executed relative to all other instructions is 32.3%. This value was originally reported by Clark [**37**]. The line with “diamond” markers represents 12-core varied sharing results with the same 15,000 cycle application run on each core. The line with “X” markers represents 12-core varied sharing results with different applications, each being 15,000 cycles, run on each core.

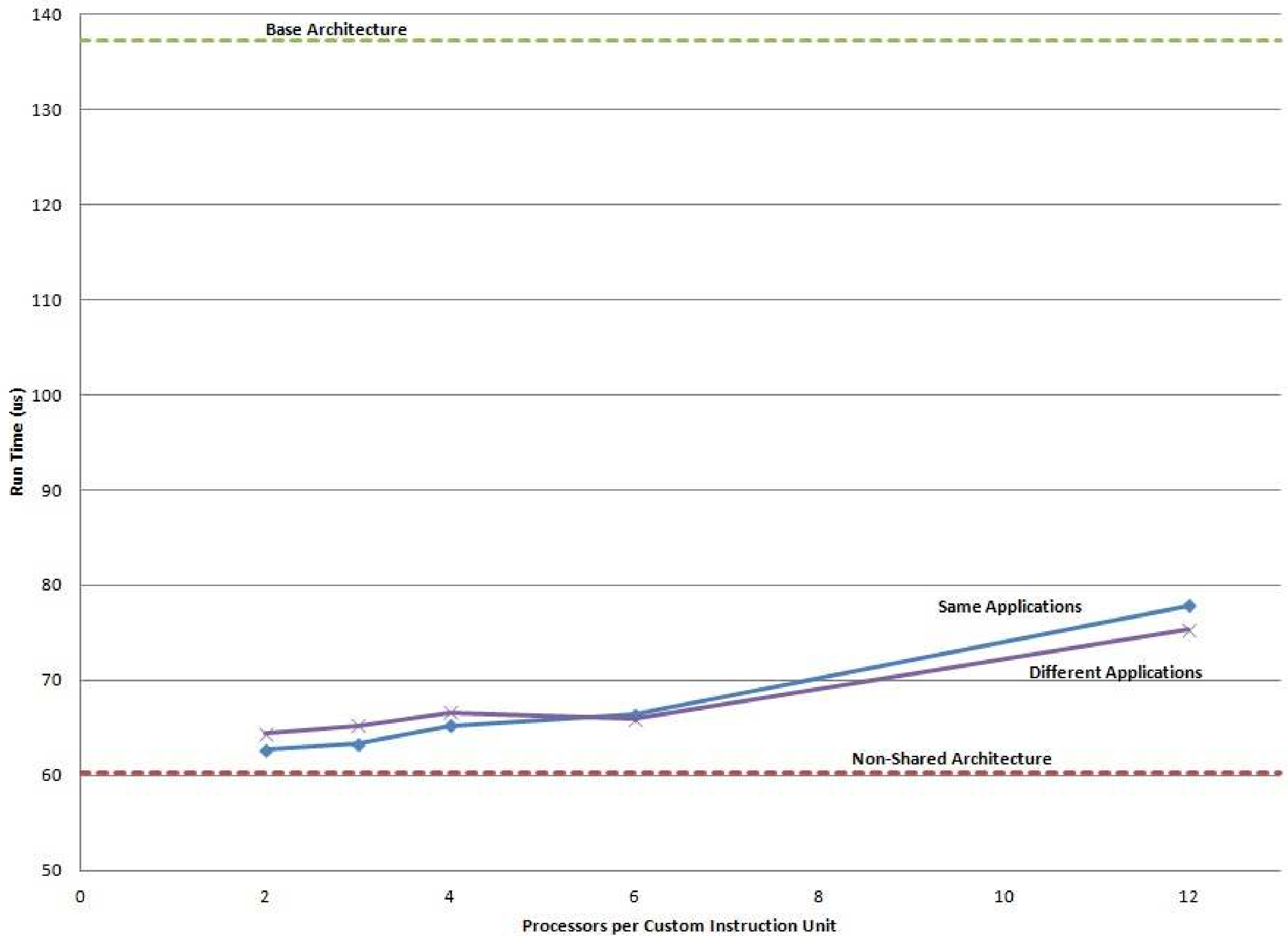


Figure 22. Run-time for a system with 12 cores and with varying degrees of sharing. Each custom instruction unit has 9 instructions.

Results are similar for both of the generated application cases: 12 processors running the same application and 12 processors each running different applications. For the 12-way arrangement, the run-time is reduced by 43% over the base, no custom instruction system for the same application case. This increases to 53% for the 4-way arrangement. For the 4-way, 6-way, and 12-way arrangements, large improvements in run-time are still seen over the base system, while causing a significant decrease in the resource usage over the non-shared architecture. The increase seen in runtime from 2-way to 12-way sharing is expected and is due to stalls caused by processors competing for the same instruction. For this case, the impact is minimal as the shared architecture run-time is still beneficial over the base architecture.

3.4 Processor and Instruction Utilization

Run-time processor and custom instruction utilization results were generated by the simulator for the 12-core system with all cores running the same application and are presented in Figure 23 and Figure 24. By construction, processor utilization is 100% for the non-shared architecture. As processor sharing is added and increased from 2-way to 12-way sharing, processor utilization drops to 88.4%. Thus, configurations with high degrees of sharing will encounter higher frequencies of collisions and therefore processors spend a higher percentage of their time stalled.

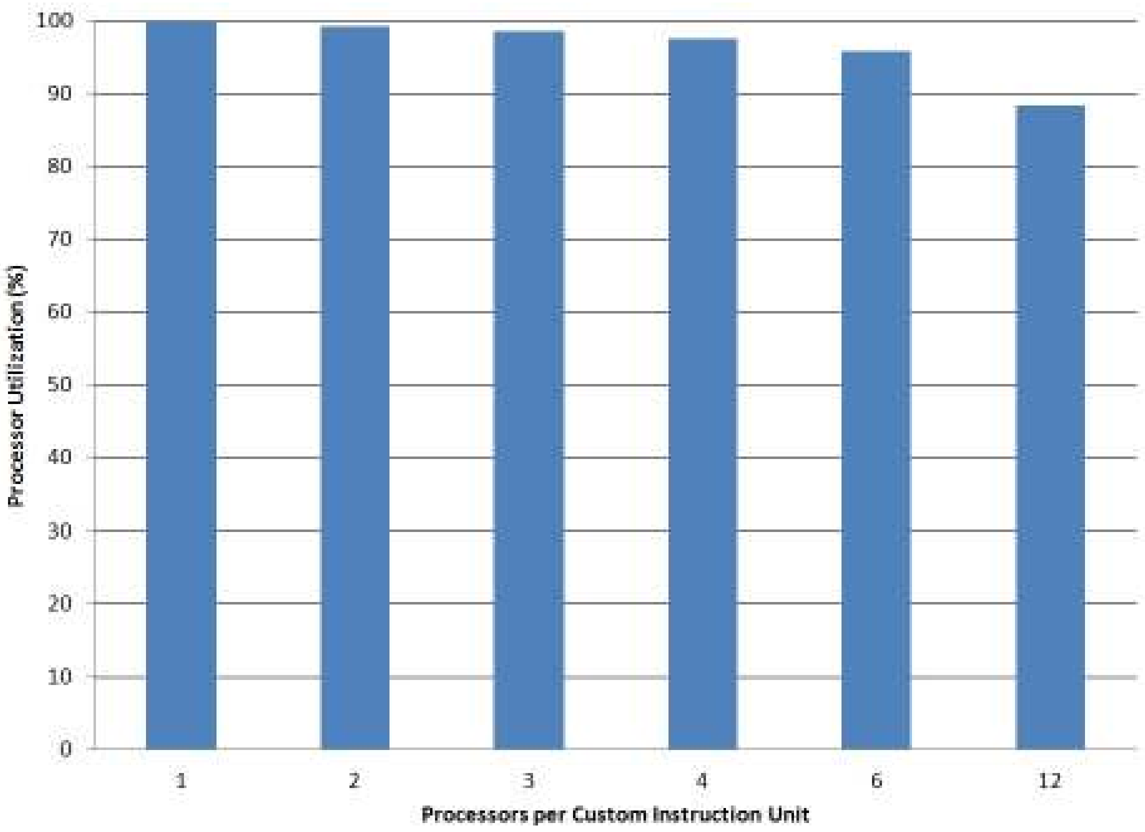


Figure 23. Processor utilization for a fine-grained system with 12 cores running the same application with varying degrees of sharing.

Custom instruction utilization follows the opposite trend to processor utilization. As the systems progress from no sharing to 12-way sharing, the average utilization for a single custom instruction in the set increases from 3.6% to 32.9%. These relatively low utilization numbers suggest that custom instructions represent a somewhat "resource expensive" approach to accelerating systems. Evidently, this expense can be minimized using the sharing approach presented.

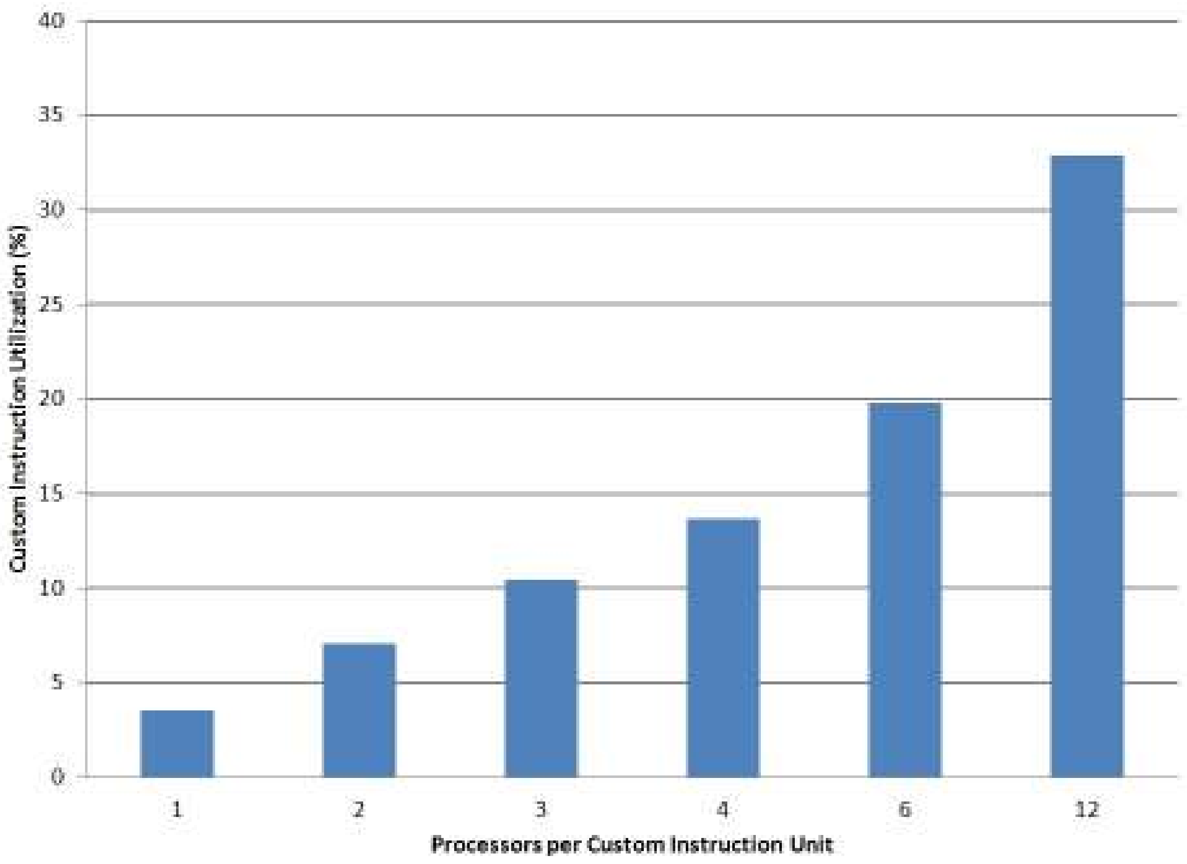


Figure 24. Custom instruction utilization for a fine-grained system with 12 cores running the same application with varying degrees of sharing.

3.5 Summary

The benchmark generator presented in this chapter can be used to create sample applications to check the benefits of custom instruction sharing. The generated benchmarks are used as inputs to the high-level simulator that provides run-time results. For the fine-grained instructions introduced in Section 2.4.1, the simulator shows that the run-time of a 12-way sharing system is reduced 43% over the base system with no custom instructions when all processors are running the same application. This same system in Section 2.4.1.3 showed a decrease in resource usage of 24% over the non-shared architecture.

Also, this chapter discussed how processor and custom instruction utilization are affected by sharing. As the degree of sharing increases, processor utilization decreases to 88.4% due to stalls, but custom instruction utilization increases to 32.9%. The increase in instruction utilization can make a “resource expensive” custom instruction implementation more appealing to designers.

4 Chapter: Coarse-Grained Case Study

As described in Chapter 2, coarse-grained instructions are larger custom instructions that require more inputs and outputs than the register file allows. This chapter presents a case study that implements, in hardware, a multi-processor system with a shared coarse-grained instruction-set. The primary focus of this chapter is to demonstrate the effectiveness of coarse-grained instruction sharing on a real platform. Because the application was implemented in hardware, results are also provided to validate the accuracy of the simulator first presented in Chapter 3. The application and custom instruction implementation are described in Section 4.1 and Section 4.2, respectively. Resource usage and critical path delay for the coarse-grained system are presented in Section 4.2.1.

The hardware run-time results from the case study are used to verify the simulator accuracy in Section 4.3. Coarse-grained custom instruction sharing and the usefulness of the simulator are discussed in Section 4.3.2. The processor and custom instruction utilization statistics follow in Section 4.3.3.

4.1 Secure Hash Algorithm (SHA)

The benchmark selected for the case study is the Secure Hash Algorithm (SHA) from the security section [**38**] of the MiBench benchmark suite developed by Guthaus et al [**39**]. The SHA was developed by the National Institute of Standards and Technology (NIST) [**40**]. It is a Federal Information Processing Standard (FIPS) [**41**] for the US government. The encryption algorithm will take any string less than 2^{64} bits and condense it into a 160 bit output. It is used to encrypt email or stored data and is also used whenever a smaller version of a message is required.

The SHA performs many logical and bit-wise operations that make it well-suited for implementation in hardware. Four sections of code were implemented in hardware as four different coarse-grained custom instructions as shown in Appendix B. The custom instructions were selected and implemented manually. The result from the application when using the custom instructions was verified against the result from the original application with no custom instructions to ensure correct implementation. More details on the coarse-grained custom instructions are provided in the next section.

4.2 Coarse-Grained Custom Instruction-Set

The SHA benchmark provides the opportunity to select coarse-grained custom instructions that replace large sections of the application source code. These custom instructions are called less frequently than fine-grained instructions but have a higher per-call impact on the speed increase. Architecture changes to accommodate coarse-grained instructions were described in Section 2.4.2.

Figure 25 is a high-level depiction of the SHA custom instruction operation. Each coarse-grained custom instruction takes the place of a full loop, performed 20 times in the SHA application. Initially, six operands are read from memory from inside the custom instruction: five from the location at input A and one from the location at input B. Encryption operations are then performed on the initial five values from input A. Next, a new value is read from memory using input B, and the encryption operations are performed again. This new value is obtained by updating the second address located at input B, and performing another read. When read and encryption operations have been performed 20 times, the instruction is complete and the initial five values are overwritten in memory.

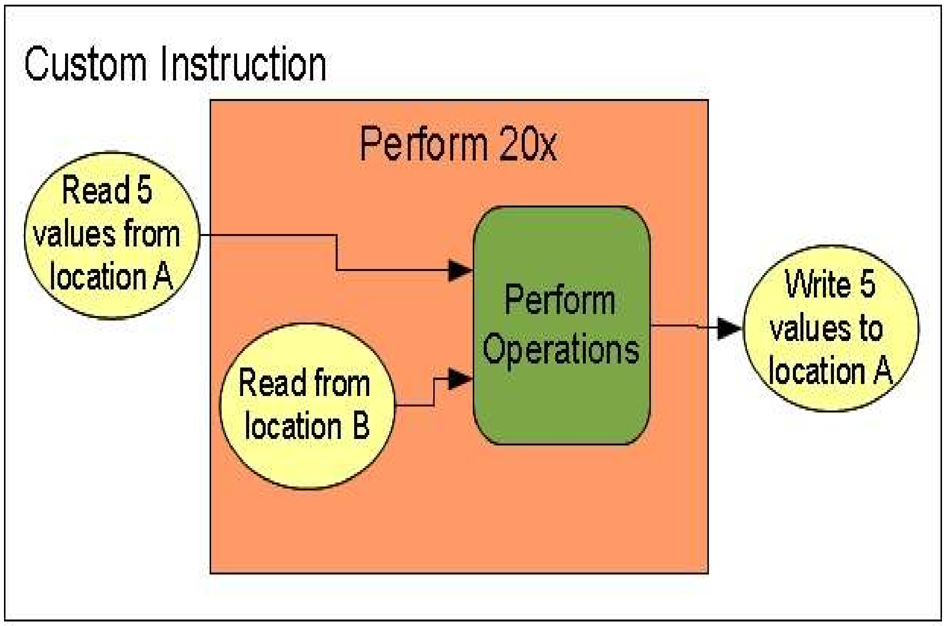


Figure 25. Coarse-grained SHA custom instruction operation.

4.2.1 Coarse-Grained Resource Usage and Critical Path Delay

The experimental setup in this section is the same as used for the fine-grained system in Chapter 2. To gather resource usage and critical path delay results for the coarse-grained system, 12 cores are implemented for a Stratix III FPGA using Nios-II/s soft processors, shared instruction memory, and separate data memories.

Figure 26 shows resource usage results similar to those presented for the fine-grained instructions in Figure 18. Benefits of sharing can be seen for 4-way, 6-way, and 12-way systems. With a 4-way sharing configuration, the implementation area is reduced by 4%. Increasing the system to 12-way sharing provides a 19% area reduction over the non-shared architecture.

Results for the 2-way and 3-way systems are larger than the non-shared architecture. In these cases, the area saved by removing custom instruction units is less than the area required for the additional multiplexors, comparators, and priority encoders. However, these results are dependent on the benchmark and the custom instruction size. It should be noted that these results were achieved through manual custom instruction selection. Other benchmarks may provide a better area reduction due to larger custom instructions. Sharing larger custom instructions will always provide a greater area reduction over the non-shared architecture. Instructions that contain larger operations such as *multipliers* and *adds* may also be beneficial with less sharing (ie. 3-way).

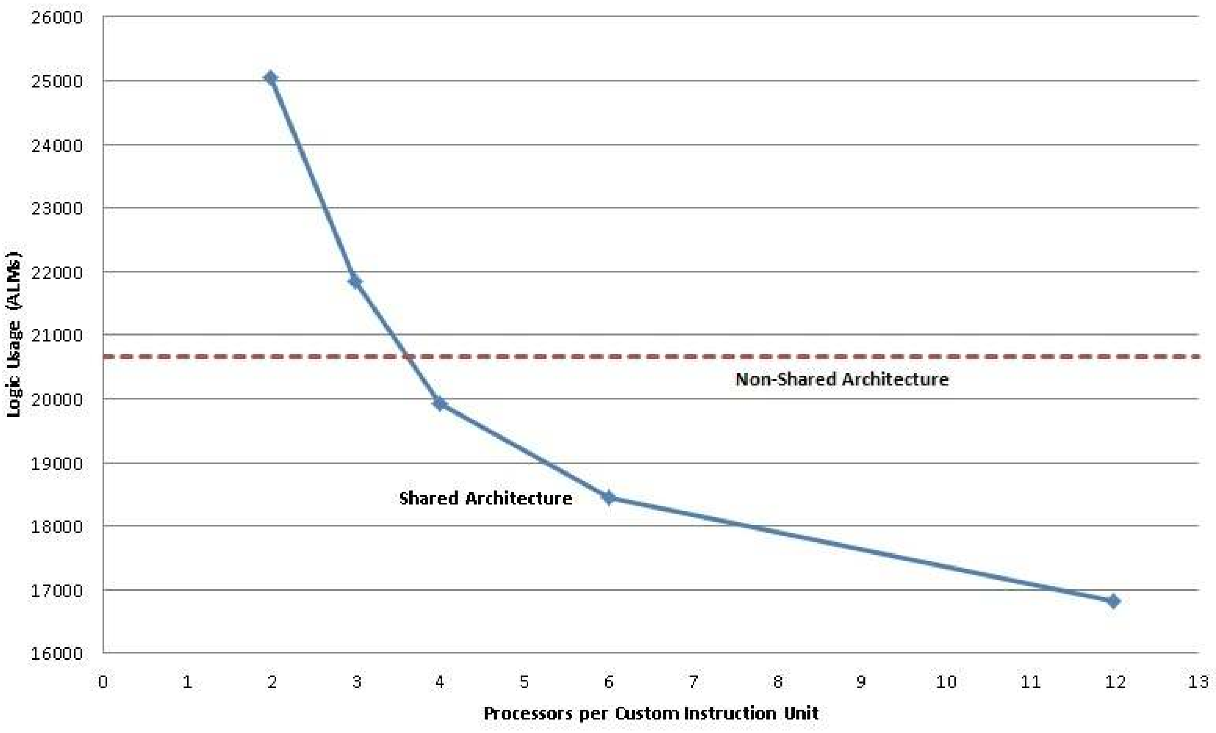


Figure 26. Resource usage for a system with 12 cores and with varying degrees of sharing. Each custom instruction unit has 4 coarse-grained instructions.

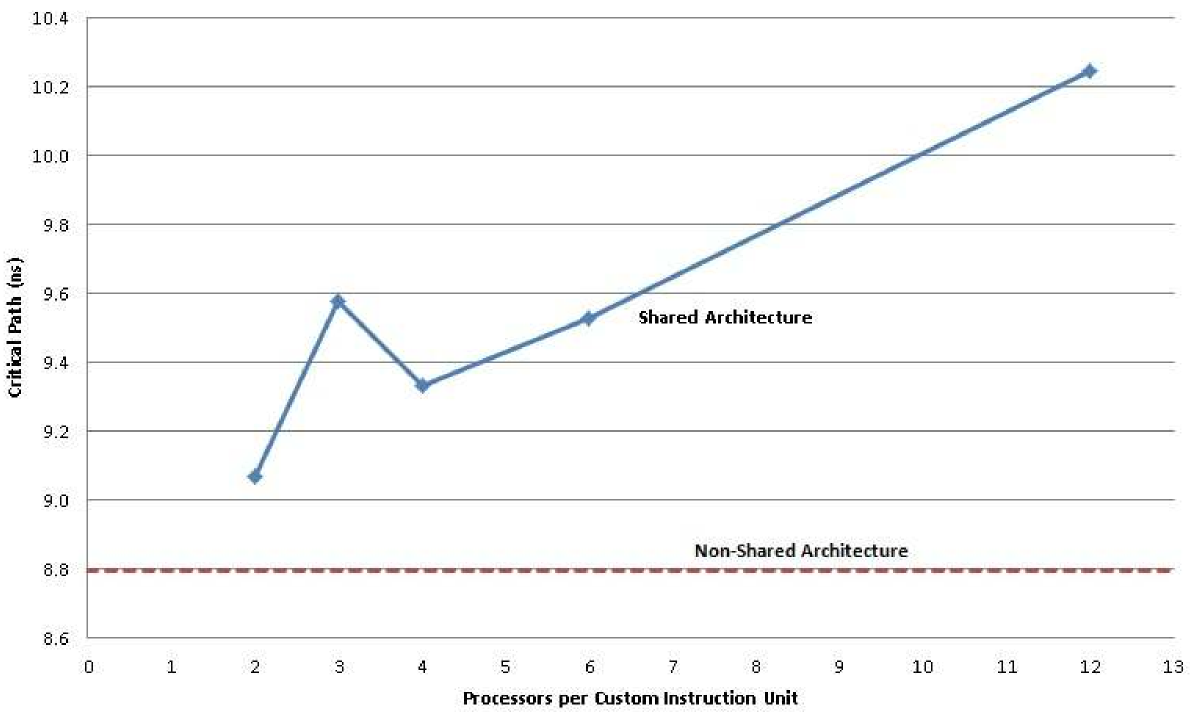


Figure 27. Critical path delay for a system with 12 cores and with varying degrees of sharing. Each custom instruction unit has 4 coarse-grained instructions.

The critical path delay results shown in Figure 27 demonstrate an example where the shared custom instruction architecture begins to affect the critical path. For 2-way through 6-way sharing systems the critical path is within 9% of the non-shared architecture. These increases are minimal and caused by layout on the FPGA, not the sharing architecture. For the 12-way configuration, the critical path delay jumps 16% over the non-shared architecture. This can be attributed to the growing size of the multiplexors and encoder required for sharing. The sharing architecture begins to sprawl more on the FPGA pushing the custom instruction on to the critical path.

4.3 Simulator Verification

Because hardware has been built for the case-study in this chapter, it can be used to help verify the accuracy of the simulator used in this chapter and Chapter 3. To perform this verification, the hardware run-time results from the SHA application case study are compared with the simulator results for the case study. The setup and method used to gather run-time information is included in Section 4.3.1 with the results produced from the experiment in Sections 4.3.2 and 4.3.3.

4.3.1 Experimental Setup

To perform run-time analysis for the coarse-grained case study, a few adjustments were made to the 12-core system. First, a performance counter core is added to the system to record the application execution cycle count. The Embedded Peripherals IP User Guide [**42**] defines the operation of the performance counter. The counter can be started when program execution begins and can be accessed by each processor as it finishes execution to retrieve the total elapsed cycle count. Second, a soft-block interface for the JTAG UART [**42**] is added to each processor. The UART allows the processors to send character streams to a terminal window via a serial connection by writing to a data register. With the counter and UART capabilities added, the processors will be able to determine execution cycle counts and print the values to the terminal window.

Finally, a hardware mutex is added to the system. A mutex is used to determine processor control of a shared resource or task. In this case, the mutex restricts each processor from executing the application until all processors have completed booting up. This ensures that all processors start at the exact same time making the cycle count accurate and making the comparison to the simulator the most reliable.

The system is once again generated and compiled using Quartus II. Next, the design is loaded onto a Stratix III FPGA on an Altera DE3 Development System [**43**] using the Quartus II programmer. Using the Nios II Integrated Development Environment (IDE), the SHA application is compiled and loaded onto each processor. The entire system is run for each sharing configuration, and the number of cycles required for each processor to complete the application is printed to the terminal window. For each configuration, the largest cycle count is taken to be the total run-time for the system.

After the hardware execution cycle-count has been determined, an instruction trace must be determined for the SHA application. The instruction trace will be used as an input to the simulator instead of employing the benchmark generator. The trace is extracted by running one of the processors in the system in ModelSim [**44**] and analyzing the wave outputs of the *performance counter*, custom instruction *start*, *done*, and *request ID* signals, and the *clock*.

By using the *start*, *done*, and *request ID* signals, one can determine whether a regular instruction or a custom instruction is being executed. If a custom instruction is being used, the *instruction ID*, *start*, and *done* signals indicate which instruction it is and when it starts and ends. Figure 28 is a small section of the ModelSim waveforms. It shows that at cycle 13,456, custom instruction 0 finishes executing as marked by *done* going high. At cycle 13,461, the *start* signal goes high indicating that custom instruction 1 is beginning execution. The performance counter will show how many clock cycles elapsed between the application start and finish. The signal values from the waveforms are placed into a spreadsheet and analyzed to produce a string of numbers representing the application execution. Each number represents one clock cycle, and the value will tell the simulator if it is a regular or custom instruction.

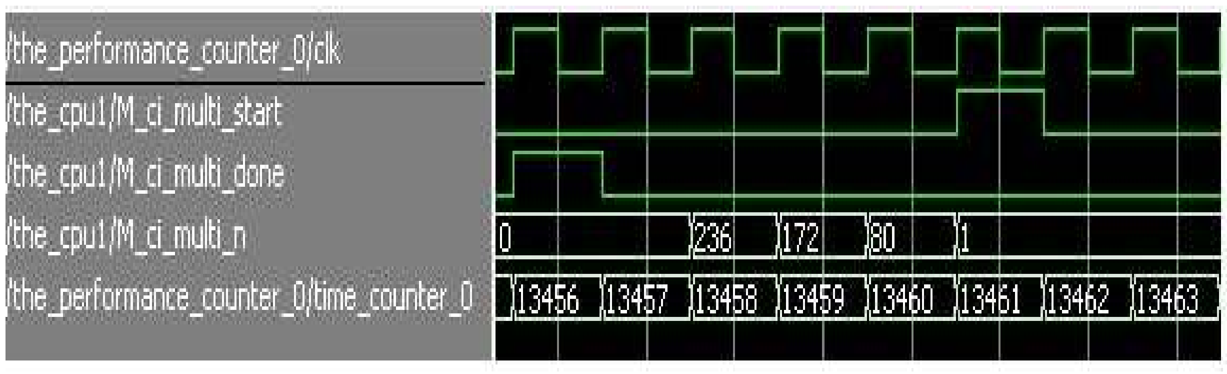

/the_performance_counter_0/clk
/the_cpu1/M_ci_multi_start
/the_cpu1/M_ci_multi_done
/the_cpu1/M_ci_multi_n
/the_performance_counter_0/time_counter_0
0
236
172
80
1
13456
13457
13458
13459
13460
13461
13462
13463

Figure 28. Modelsim waveform used to determine application trace.

Rather than using ModelSim for all sharing configurations, a trace can now be extracted for a single processor and quickly applied to all sharing configurations using the simulator. ModelSim requires the entire system to be designed, generated, and compiled before a simulation can be performed. The simulation alone can take hours, and this time will only grow as more processors and complex instructions are added to the system. The simulator only requires a trace from a single processor and can produce run-time results for all sharing configurations in just minutes. This can help us determine if a system will benefit from a certain configuration and specific instructions before the time and effort is put into implementing the system.

4.3.2 Coarse-Grained Run-Time Results

Hardware run-time results for the coarse-grained system are shown in Figure 29. The execution cycles, determined using the performance counter, are multiplied by the critical path delay determined previously. This value represents the overall run-time. Run-time results are also determined using the same method for a 12 processor system running the SHA application with no custom instructions, referred to as the base architecture. Compared to the base architecture that takes 228 μs to execute, the 12-way sharing case takes 173 μs, improving run-time by 24%. For the 4-way sharing case which still shows an area improvement over the non-shared architecture, the execution time compared to the base system is improved by 32%.

Compared to the non-shared architecture, the 4-way sharing configuration only requires 1.08 times longer to execute. The 12-way sharing configuration only requires 1.19 times longer than the non-shared architecture. Although the 2-way and 3-way sharing configurations show significant run-time improvements over the base architecture, they will not be analyzed as these configurations did not provide any improvement in resource usage over the non-shared architecture.

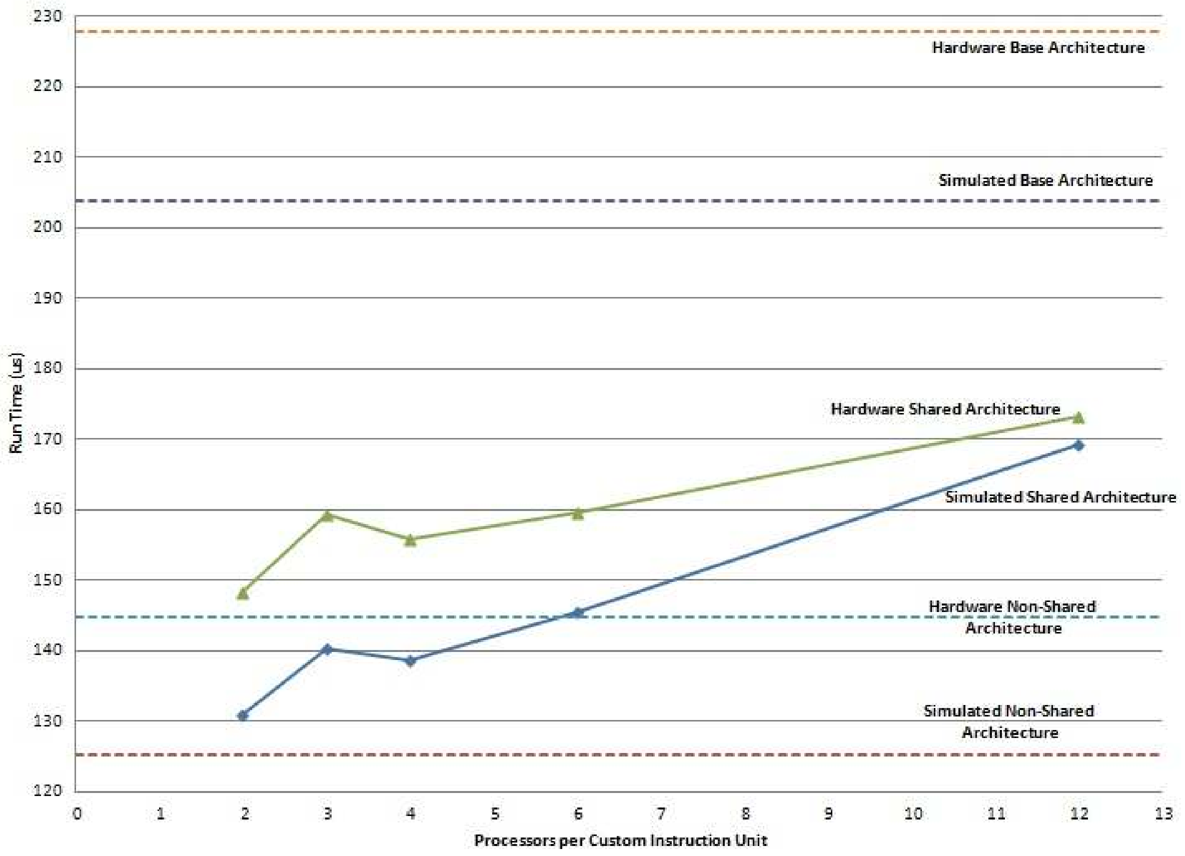


Figure 29. Hardware and simulated run-time for a system with 12 cores and with varying degrees of sharing. Each custom instruction unit has 4 instructions.

The simulated run-time results for the coarse-grained case study are also presented in Figure 29. Results show that for the 2-way, 3-way, and 4-way sharing configurations, where less sharing occurs, the hardware run-time differs from the simulated by 17 to 20 μs. At 6-way sharing this drops to 14 μs, and it drops again to just 4 μs at 12-way sharing. In all sharing cases, the simulated runtime is less than 12% different from the hardware runtime.

The difference between the two sets of data can be attributed to conflicting attempts to access instruction memory. For the configurations where less sharing occurs, more processors will continually be at the same execution point in the application. When a processor must fetch instructions from the instruction memory, rather than its individual instruction cache, there may be other processors attempting the same action. Much like when multiple processors request the same custom instruction, only one processor can access instruction cache at a time. The others must queue until the resource becomes available. Since the simulator operates at a high level, it is unable to take these stalls into account. The effect of instruction memory access stalls would be less noticeable in a system with more sharing, since processor execution is already offset due to stalls for the custom instructions.

The main reason the two data sets tend to converge as the sharing increases is due to pauses in the performance counter. Each time a processor reads the cycle count from the performance counter, the counter pauses. It must then be told to resume by the processor. This allows approximately 140 clock cycles to pass by uncounted. For configurations where more sharing occurs, each processor will finish at a different time, creating more individual reads from the performance counter.

Figure 30 shows the adjusted hardware run-time taking these uncounted cycles into consideration. The 12-way configuration still shows a hardware run-time reduction of 17%

over the base architecture. Now, the difference between simulated run-times and hardware run-times is consistent between 19 µs to 24 µs. This means the simulated run-time stays within 10.5% to 13.5% of the hardware run-time. The offset between the two data sets is roughly constant and can be attributed to boot-up cycles in the hardware run-time that are unaccounted for in the simulation.

These results demonstrate the accuracy of the simulator, proving that the fine-grained simulator results are valid. The simulator is an effective way to explore the design space and determine run-time results without the required effort and time to fully implement the design.

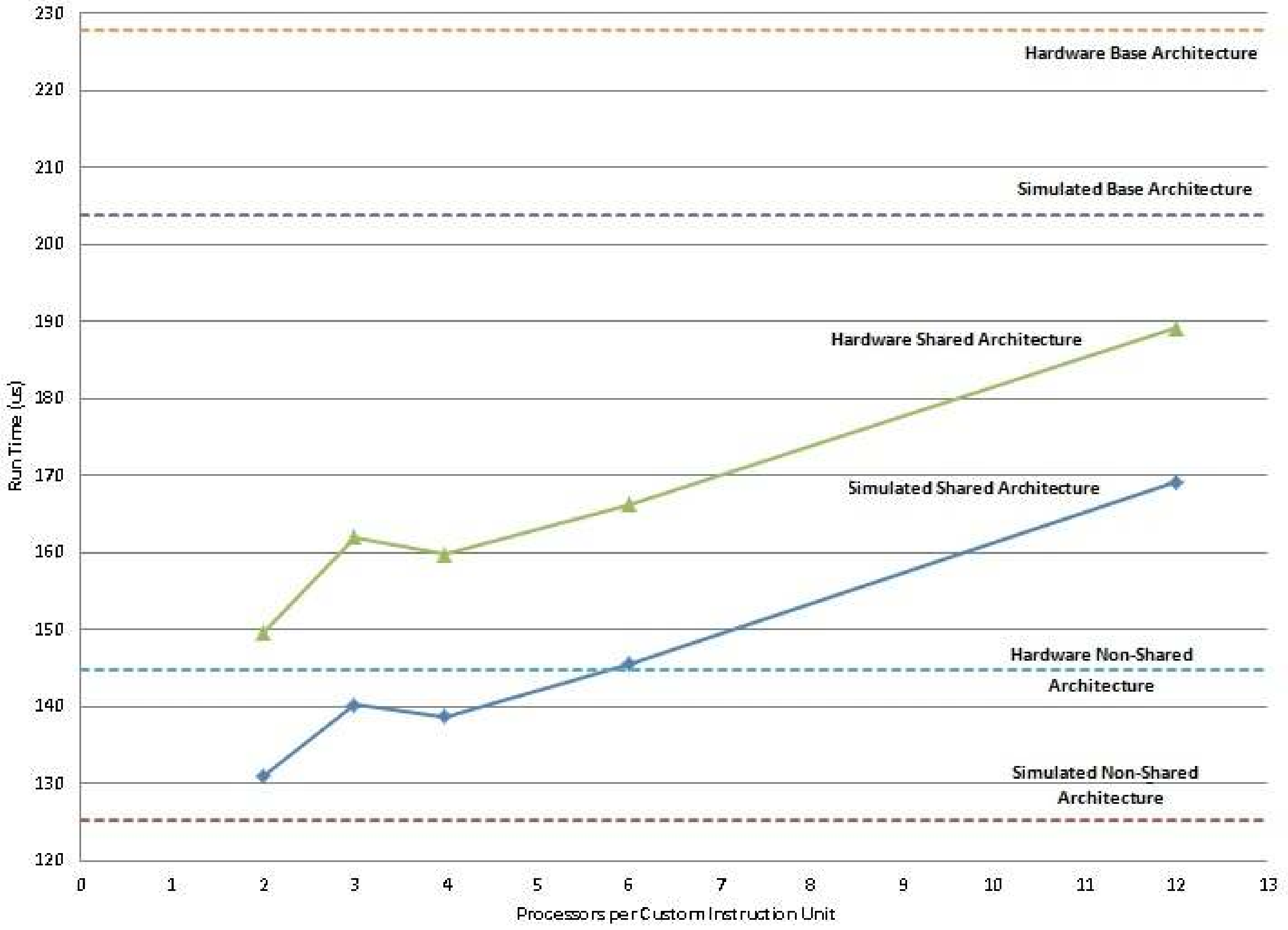


Figure 30. Adjusted hardware and simulated run-time results.

4.3.3 Processor and Instruction Utilization

Processor and custom instruction utilization during simulation are shown in Figure 31 and Figure 32 for the 12-core system with each core running the SHA application using the coarse-grained instructions. From the non-shared architecture to 12-way sharing, the processor utilization decreases from 100% to 92.8% due to stalls caused by multiple concurrent requests to the same custom instruction. The custom instruction utilization once again follows the opposite trend to the processor utilization. As the sharing increases from the non-shared architecture to 12-way, the utilization increases from 0.6% to 6.7% for a single custom instruction.

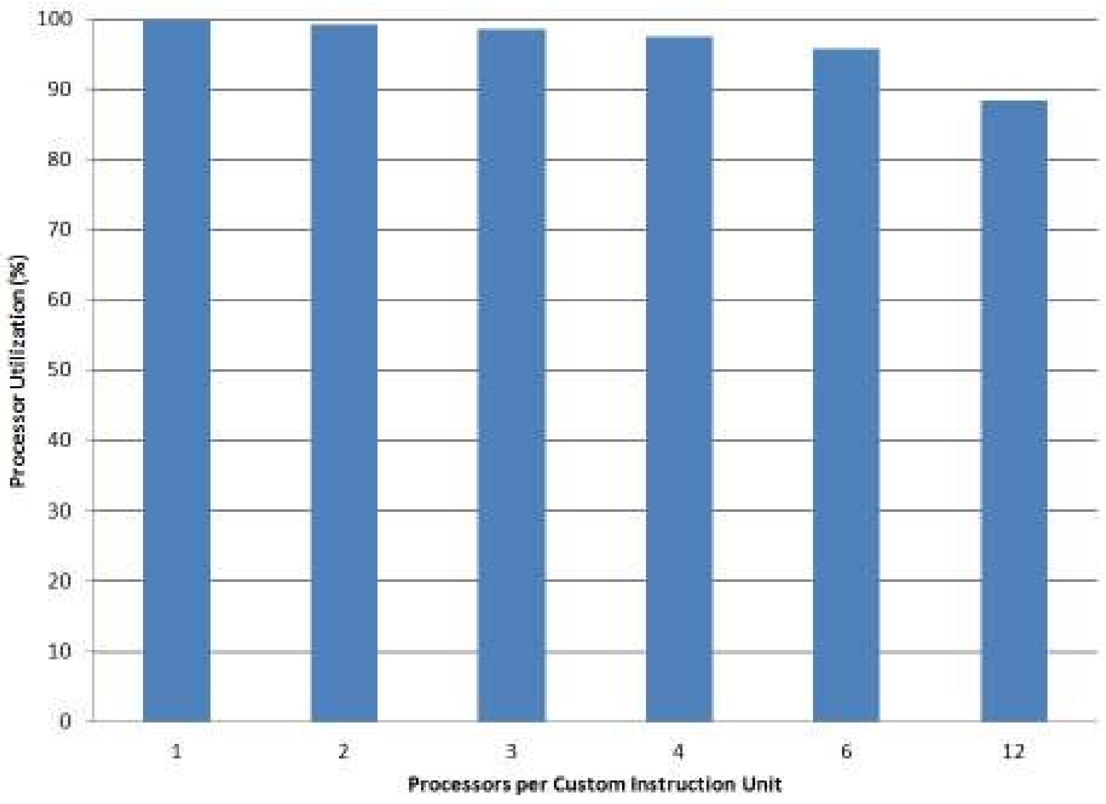


Figure 31. Processor utilization for a coarse-grained system with 12 cores running the SHA encryption algorithm with varying degrees of sharing.

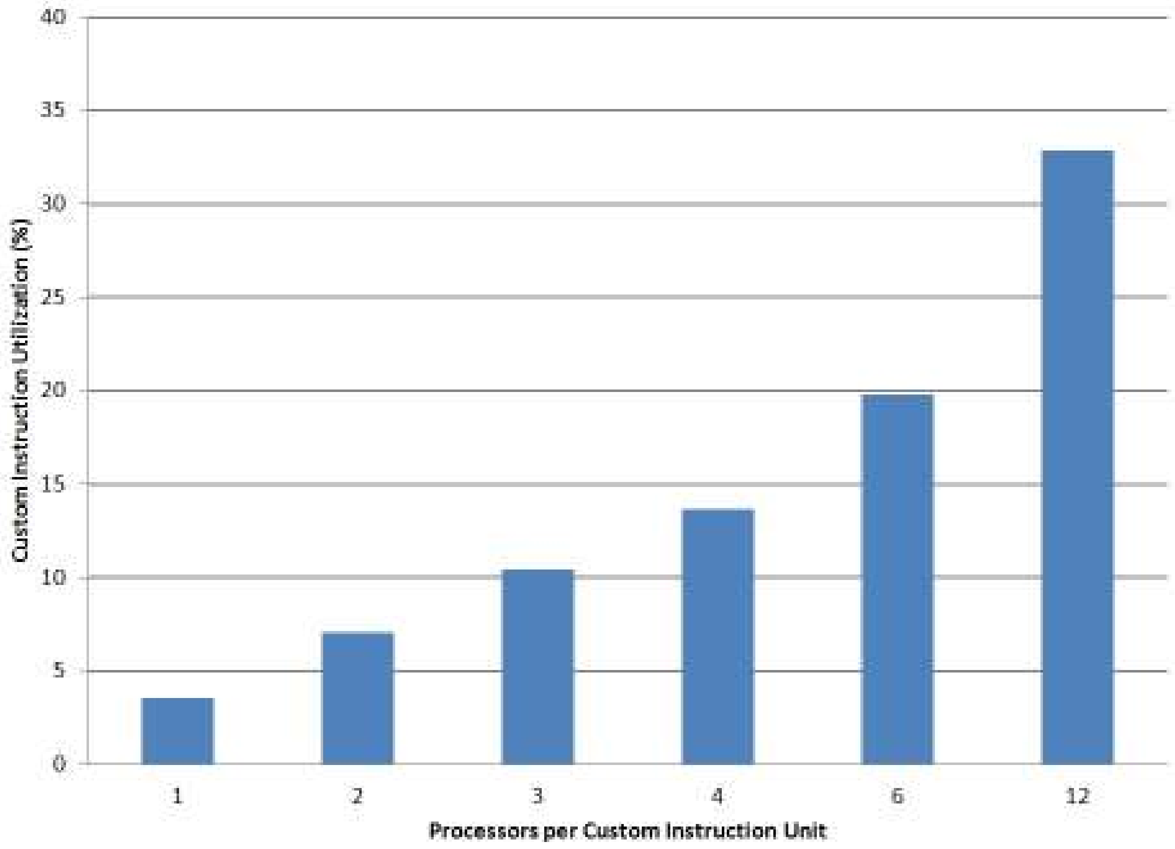


Figure 32. Custom instruction utilization for a coarse-grained system with 12 cores running the SHA encryption algorithm with varying degrees of sharing.

4.4 Summary

This chapter introduced the coarse-grained case study. The SHA application from the MiBench benchmark suite was determined to benefit performance-wise from four coarse-grained custom instructions. When compiled for varied sharing configurations, the 12-way configuration showed a 19% reduction in resource usage, while the critical path is 16% greater than the non-shared architecture.

Next, hardware and simulated run-time results were determined for the case study. Results show that the reported run-time is affected by pauses in the performance counter that occur each time the counter is read. After accounting for these pauses, results show that the simulated run-time is within 10.5% to 13.5% of the adjusted hardware run-time. The difference in run-time results between the two sets can be attributed to instruction memory access stalls and unaccounted boot-up cycles.

Finally, utilization statistics determined by the simulator report that processor utilization drops from 100% to 92.8% as sharing increases to 12-way. The custom instruction utilization changes from 0.6% to 6.7% as sharing increases.

5 Chapter: Conclusion

In this thesis, the sharing of custom instruction units in soft multi-core systems was investigated. In an effort to reduce the resource usage required to implement instruction-set extensions while maintaining performance benefits, a shared architecture was developed. Previous work focused on sharing fixed hardware blocks or implementation fabric. This work is the first to share full custom instruction units. The effectiveness of the new architecture was verified using fine-grained examples and a coarse-grained case study. A simulator was also developed to efficiently determine run-time results.

Chapter 2 introduced the new architecture that allows soft core systems implemented on an FPGA to share custom instruction resources. Using Nios II processors and Altera design tools, an architecture was developed to reduce implementation area by sharing custom instruction units. Processor cores that simultaneously request access to the shared instructions are queued until given access based on a fixed priority scheme.

Next, an experiment was designed to test fine-grained resource usage and critical path delay using systems with 9 fine-grained custom instructions. Resource usage and critical path delay results were determined for fine-grained systems containing 3, 6, 9, 12, 15, and 18 processors. Based on these results, it was determined that the 12 processor system should be used for further analysis. Using the 12 processors in varied sharing configurations, results showed that as much as 24% resource usage can be saved with minimal impact on critical-path delay. In the 4-way, 6-way, and 12-way configurations, overall area reductions were seen since the reduced number of custom instruction units implemented saved enough area to overcome the additional sharing infrastructure.

An application generator and simulator to evaluate the run-time effectiveness of instruction sharing were developed and detailed in Chapter 3. The benchmark generator created an application of specified length and then reduced the number of execution cycles by inserting custom instructions. The generated application was run using the simulator to determine the impact of sharing on run-time. It was found that the run-time performance for the 4-way sharing case, which still showed area improvements, could be improved by as much as 53% over a system without custom instructions. Sharing also increased the individual custom instruction utilization from 3.6% for the non-shared architecture to 32.9% for the 12-way sharing case.

Finally, the new architecture was used to implement a 12 processor system with each processor executing the SHA application from the MiBench benchmark suite. Four coarse-grained custom instructions were manually selected from the application source code and implemented in hardware. Results showed that implementing the new architecture could reduce resource usage by as much as 19% over a system with no sharing. Hardware run-time results for the coarse-grained case study were determined by executing the design and application on a Stratix III device. The 12-way sharing case improved run-time by 24% over a system with no custom instructions.

The hardware run-time results for the coarse-grained case study were compared with the simulated results to verify the accuracy of the simulator. The performance counter used to determine hardware run-time cycles pauses with each access. Once these uncounted cycles are considered, the hardware run-time results remain within 10.5% to 13.5% of the simulated run-time results. The difference between the two data sets can be attributed to unaccounted boot-up cycles and instruction memory access stalls.

These results verify the accuracy of the simulator and its results for the fine-grained system. The simulator can be used as a quick way to determine the run-time impact of sharing custom instruction units, before having to design and implement the entire system.

Overall, this work provides a solution to the resource usage problem caused by instruction-set extensions. It has been shown that sharing is possible and beneficial in terms of area and application run-time. The architecture developed is highly flexible and can be easily adjusted to fit any number of processors using any number of custom instructions.

5.1 Limitation of the Work

The main limitation of the work described in this thesis is caused by the extra hardware required to implement the shared architecture. The additional multiplexors and arbitration unit can overtake the benefits of sharing. As the number of processors in the system grows, the additional hardware also becomes larger, taking up more of the area savings.

It is also seen from the results that the amount of sharing must reach a certain threshold before resource usage benefits are noticed. However, this must be determined on a case-by-case basis as every custom instruction unit requires a different amount of resources. A system that uses larger custom instructions will see a reduction in area sooner than systems with smaller custom instructions.

Also, the current architecture only supports equally sized sharing groups. For example, a 12-core system can have two groups of six processors sharing a single custom instruction unit. But, an 11-core system cannot have one group of five processors and one group of six processors. This limitation is one that could be solved, however, with just a few adjustments to the design.

5.2 Future Work

As mentioned in Chapter 1, this work is a first step at examining the use of instruction sharing in multi-core systems. More work is needed to develop compiler support such that instruction candidates can be selected to maximize sharing opportunities among multiple processors. Algorithms currently exist to define custom instruction candidates for single processor systems, but more work must be done to expand these to analyze multiprocessor systems. I feel that the sharing of instruction resources will eventually become a critical part of all future multi-core systems.

After compiler support is developed, a thorough power analysis would be beneficial to determine the effects on both static and dynamic power due to sharing. Compiler support would allow systems to quickly be customized for specific benchmarks. It is expected that a drop in static power due to reduced area would be seen, but that an increase in dynamic power due to increased run-time would be seen.

This work has also spurred opportunities to investigate varying degrees of sharing within the custom instruction unit. For example, it may be beneficial to share some custom instructions among all processors and have individual implementations for more commonly used instructions. This presents a chance to optimize the sharing configuration for each application.

5.3 List of Contributions

The contributions of my work and the completion of my objectives are summarized as follows:

1. I developed an architecture to allow multiple processors to share ISEs. The architecture is able to maintain performance benefits achieved by implementing custom instructions, while reducing the implementation area required.
2. Area, critical path delay, and run-time results of fine-grained and coarse-grained systems implementing the shared architecture were analyzed.
3. A benchmark generator and simulator were developed to assist in the analysis of the shared architecture and to allow the design space to be thoroughly and quickly explored.

References

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, pp. 114-117, Apr. 1965.

[2] H. Chang, et al., *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Norwell, Massachusetts, USA: Kluwer Academic Publishers, 1999.

[3] P. Magarshack and P. G. Paulin, "System-on-chip beyond the nanometer wall," in *Proceedings of the 40th Design Automation Conference*, Anaheim, CA, 2003, pp. 419-424.

[4] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, San Diego, CA, 2007, pp. 746-749.

[5] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*. Newnes, 2004.

[6] P. S. Zuchowski, et al., "A hybrid ASIC and FPGA architecture," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, 2002, pp. 187-194.

[7] Altera Corporation. (2012, Jul.) Altera. [Online]. www.altera.com

[8] Xilinx Inc. (2012, Jul.) Xilinx. [Online]. www.xilinx.com

[9] Altera. (2012, Jul.) FPGAs. [Online]. http://www.altera.com/products/fpga.html

[10] Altera. (2012, Jul.) Nios II Processor: The World's Most Versatile Embedded Processor. [Online]. www.altera.com/devices/processor/nios2/ni2-index.html

[11] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid, "Soft-Core Processors for Embedded Systems," in *International Conference on Microelectronics*, Dhahran, Saudi Arabia, 2006, pp. 170-173.

[12] ARM. (2012, Jul.) The Architecture for the Digital World. [Online]. www.arm.com

[13] N. Clark, et al., "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, Madison, Wisconsin, 2005, pp. 272-283.

[14] Stretch. (2012, Jul.) Software-Configurable Processors: Solving the Embedded System Design Dilemma. [Online]. www.stretchinc.com

[15] Tensilica. (2012, Jul.) Xtensa Processor Developer's Toolkit. [Online]. http://www.tensilica.com/products/hw-sw-dev-tools/for-processor-designers.htm

[16] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic instruction set extension and utilization for embedded processors," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors.*, The Hague, Netherlands, 2003, pp. 108-118.

[17] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, "Fast custom instruction identification by convex subgraph enumeration," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, Leuven, Belgium, 2008, pp. 1-6.

[18] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613-641, Aug. 1978.

[19] T. Callahan and J. Wawrzynek, "Instruction-Level Parallelism for Reconfigurable Computing," *Lecture Notes in Computer Science*, vol. 1482, pp. 248-257, 1998.

[20] W. Wolf, "The future of multiprocessor systems-on-chips," in *Proceedings of the 41st annual Design Automation Conference*, San Diego, CA, 2004, pp. 681-685.

[21] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, and L. Pozzi, "Performance and energy benefits of instruction set extensions in an FPGA soft core," in *Proceedings of the 19th International Conference on VLSI Design*, Hyderabad, India, 2006.

[22] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209-1229, Jul. 2006.

[23] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004.*, Portland, 2004, pp. 30-40.

[24] X. Chen, D. L. Maskell, and Y. Sun, "Fast Identification of Custom Instructions for Extensible Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuts and Systems*, vol. 26, no. 2, pp. 359-368, Feb. 2007.

[25] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *CASES*, Washington, 2004, pp. 69-78.

[26] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, "Automatic Selection of Application-Specific Instruction-Set Extensions," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, Seoul, Korea, 2006, pp. 160-165.

[27] H. Lin and Y. Fei, "Resource Sharing of Pipelined Custom Hardware Extension for Energy-efficient Application-specific Instruction Set Processor Design," in *ICCD*, Lake Tahoe, 2009, pp. 158-165.

[28] Q. Dinh, D. Chen, and M. D. F. Wong, "Efficient ASIP Design for Configurable Processors with Fine-Grained Resource Sharing," in *FPGA*, California, 2008, pp. 99-106.

[29] M. Zuluaga and N. Topham, "Resource Sharing in Custom Instruction Set Extensions," in *SASP*, Anaheim, 2008, pp. 7-13.

[30] S.-K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *Journal of Systems Architecture*, vol. 55, no. 1, pp. 1-14, Jan. 2009.

[31] L. Chen and T. Mitra, "Shared Reconfigurable Fabric for Multi-core Customization," in *DAC*, San Diego, 2011.

[32] D. Sheldon, R. Kumar, F. Vahid, D. Tullsen, and R. Lysecky, "Conjoining Soft-Core FPGA Processors," in *ICCAD*, San Jose, 2006, pp. 694-701.

[33] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Germany: Springer, 2004.

[34] Altera. (2011, May) Nios II Processor Reference Handbook. [Online]. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[35] Altera. (2011, Jan.) Nios II Custom Instruction User Guide. [Online]. http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

[36] P. Ienne and R. Leupers, Eds., *Customizable Embedded Processors: Design Technologies and Applications*. San Francisco, USA: Morgan Kaufmann, 2007.

[37] N. Clark, "Customizing the computation capabilities of microprocessors," PhD Thesis, University of Michigan, Ann Arbor, 2007.

[38] M. R. Guthaus, et al. (2012, Mar.) MiBench Version 1.0. [Online]. http://www.eecs.umich.edu/mibench/source.html

[39] M. R. Guthaus, et al., "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, 2001.

[40] NIST. (2012, Jul.) National Institute of Standards and Technology. [Online]. www.nist.gov

[41] Federal Information Processing Standards. (1995, Apr.) Secure Hash Standard. [Online]. www.itl.nist.gov/fipspubs/fip180-1.htm

[42] Altera. (2011, Jun.) Embedded Peripherals IP User Guide. [Online]. www.altera.com/literature/ug/ug_embedded_ip.pdf

[43] Terasic. (2012, Jul.) Altera DE3 Development System. [Online]. http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=260

[44] Mentor Graphics. (2012, Nov.) ModelSim. [Online]. www.model.com

Appendices

Appendix A - Fine-Grained Custom Instruction Implementation

The following is a section of code from shared_inst.sv that shows the implementation for 9 “average” fine-grained custom instructions.

```
generate
case (n)
    0:      begin: a
            always @(posedge clk)
            begin
                    adata <= dataa;
                    bdata <= datab;
                     x1 = adata * bdata;
                     x2 = adata | bdata;
                     x3 = adata & bdata;
                     result = x1 & x2 & x3;
            end
            end
    1:      begin: b
            always @(posedge clk)
            begin
                    adata <= dataa;
                    bdata <= datab;
                     x1 = adata ^ bdata;
                     x2 = adata * adata;
                     x3 = adata | bdata;
                     result = x1 + x2 + x3;
            end
            end
    2:      begin: c
            always @(posedge clk)
            begin
                    adata <= dataa;
                    bdata <= datab;
                     x1 = adata + bdata;
                     x2 = bdata * bdata;
                     x3 = adata & bdata;
                     result = x1 | x2 | x3;
            end
            end
    3:      begin: d
            always @(posedge clk)
```

```
        begin
                adata <= dataa;
                bdata <= datab;
                 x1 = adata + adata;
                 x2 = bdata + bdata;
                 x3 = adata | bdata;
                 result = x1 ^ x2 ^ x3;
        end
        end

4:      begin: e
        always @(posedge clk)
        begin
                adata <= dataa;
                bdata <= datab;
                 x1 = adata | bdata;
                 x2 = adata + bdata;
                 x3 = adata ^ bdata;
                 result = x1 - x2 - x3;
        end
        end
5:      begin: f
        always @(posedge clk)
        begin
                adata <= dataa;
                bdata <= datab;
                 x1 = adata * bdata;
                 x2 = adata & bdata;
                 y1 = x1 + x2;
                 y2 = x1 | x2;
                 result = y1 & y2;
        end
        end
6:      begin: g
        always @(posedge clk)
        begin
                adata <= dataa;
                bdata <= datab;
                 x1 = adata + bdata;
                 x2 = adata | bdata;
                 y1 = x1 & x2;
                 y2 = x1 + x1;
                 result = y1 | y2;
        end
        end
7:      begin: i
```

```
            always @(posedge clk)
            begin
                    adata <= dataa;
                    bdata <= datab;
                     x1 = adata * bdata;
                     x2 = adata & bdata;
                     x3 = adata | bdata;
                     y1 = x1 + x2;
                     y2 = x1 | x3;
                     z1 = y1 | y2;
                     result = z1 + y2;
            end
            end
    8:      begin: j
            always @(posedge clk)
            begin
                    adata <= dataa;
                    bdata <= datab;
                     x1 = adata * bdata;
                     x2 = adata & bdata;
                     y1 = x1 + x2;
                     y2 = x2 + x2;
                     z1 = y1 & y2;
                     z2 = y1 | y2;
                     zz = z1 & z2;
                     result = z1 + zz;
            end
            end
endcase
endgenerate
```

Appendix B - Case Study Implementation

The following is a section of code from the SHA application that was selected for coarse-grained custom instruction implementation. The Verilog code for each custom instruction is also provided.

B.1 SHA Application

The highlighted section below, including the for loops, represents the source code selected for custom instruction implementation. The highlighted section uses the macros defined at the top of the code.

```
/* SHA f()-functions */
#define f1(x,y,z)   ((x & y) | (~x & z))
#define f2(x,y,z)   (x ^ y ^ z)
#define f3(x,y,z)   ((x & y) | (x & z) | (y & z))
#define f4(x,y,z)   (x ^ y ^ z)

/* SHA constants */
#define CONST1          0x5a827999L
#define CONST2          0x6ed9eba1L
#define CONST3          0x8f1bbcdcL
#define CONST4          0xca62c1d6L

/* 32-bit rotate */
#define ROT32(x,n)      ((x << n) | (x >> (32 - n)))

#define FUNC(n,i)                                               \
   temp = ROT32(A,5) + f##n(B,C,D) + E + W[i] + CONST##n;       \
   E = D; D = C; C = ROT32(B,30); B = A; A = temp
```

```
/* do SHA transformation */
static void sha_transform(SHA_INFO *sha_info)
{    int i;
    LONG temp, A, B, C, D, E, W[80];

    for (i = 0; i < 16; ++i) {
      W[i] = sha_info->data[i];     }
    for (i = 16; i < 80; ++i) {
       W[i] = W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16];
       W[i] = ROT32(W[i], 1);     }
    A = sha_info->digest[0];
    B = sha_info->digest[1];
    C = sha_info->digest[2];
    D = sha_info->digest[3];
    E = sha_info->digest[4];

    for (i = 0; i < 20; ++i) {
    FUNC(1,i); }
    for (i = 20; i < 40; ++i) {
    FUNC(2,i); }
    for (i = 40; i < 60; ++i) {
    FUNC(3,i)  }
    for (i = 60; i < 80; ++i) {
    FUNC(4,i)  }

    sha_info->digest[0] += A;
    sha_info->digest[1] += B;
    sha_info->digest[2] += C;
    sha_info->digest[3] += D;
    sha_info->digest[4] += E;
}
```

B.2 Coarse-Grained Custom Instruction Implementation

The following is a section of code from CI.sv that shows the implementation for the 4 coarse-grained custom instructions. It should be noted that this strictly shows the instruction implementation and does not show the state machines and modules required to read inputs from and write outputs to memory.

```
generate
        case (n)
                0:      begin
                        always @ (posedge clk)
                        begin
                // FUNC(1,i)
                        // temp = ROT32(A,5) + f1(B,C,D) + E + W[i] + CONST1;
                        // E = D; D = C; C = ROT32(B,30); B = A; A = temp
                                // ROT32(x,n) ((x << n) | (x >> (32 - n)))
                                // f1(x,y,z)     ((x & y) | (~x & z))

                        x1 = A << 5;
                        x2 = A >> 27;
                        y1 = x1 | x2;

                        x3 = B & C;
                        x4 = ~B & D;
                        y2 = x3 | x4;

                        Out1 = y1 + y2 + E + R5 + 32'h5a827999;
                        Out2 = D;
                        Out3 = C;

                        x5 = B << 30;
                        x6 = B >> 2;
                        Out4 = x5 | x6;

                        Out5 = A;
                        end
                end
```

```
1:      begin
        always @ (posedge clk)
        begin
// FUNC(2,i)
        // temp = ROT32(A,5) + f2(B,C,D) + E + W[i] + CONST2;
        // E = D; D = C; C = ROT32(B,30); B = A; A = temp
                // ROT32(x,n) ((x << n) | (x >> (32 - n)))
                // f2(x,y,z)       (x ^ y ^ z)

        x1 = A << 5;
        x2 = A >> 27;
        y1 = x1 | x2;

        x3 = B ^ C;
        y2 = x3 ^ D;

        Out1 = y1 + y2 + E + R5 + 32'h6ed9eba1;
        Out2 = D;
        Out3 = C;

        x5 = B << 30;
        x6 = B >> 2;
        Out4 = x5 | x6;

        Out5 = A;
        end
end
```

```
2:      begin
        always @ (posedge clk)
        begin
// FUNC(3,i)
        // temp = ROT32(A,5) + f3(B,C,D) + E + W[i] + CONST3;
        // E = D; D = C; C = ROT32(B,30); B = A; A = temp
                // ROT32(x,n) ((x << n) | (x >> (32 - n)))
                // f3(x,y,z)      ((x & y) | (x & z) | (y & z))

        x1 = A << 5;
        x2 = A >> 27;
        y1 = x1 | x2;

        x3 = B & C;
        x4 = B & D;
        x5 = C & D;
        y2 = x3 | x4 | x5;

        Out1 = y1 + y2 + E + R5 + 32'h8f1bbcdc;
        Out2 = D;
        Out3 = C;

        x6 = B << 30;
        x7 = B >> 2;
        Out4 = x6 | x7;

        Out5 = A;
        end
end
```

```
3:      begin
                always @ (posedge clk)
                begin
            // FUNC(3,i)
                // temp = ROT32(A,5) + f4(B,C,D) + E + W[i] + CONST4;
                // E = D; D = C; C = ROT32(B,30); B = A; A = temp
                    // ROT32(x,n) ((x << n) | (x >> (32 - n)))
                    // f4(x,y,z)      (x ^ y ^ z)

                x1 = A << 5;
                x2 = A >> 27;
                y1 = x1 | x2;

                x3 = B ^ C;
                y2 = x3 ^ D;

                Out1 = y1 + y2 + E + R5 + 32'hca62c1d6;
                Out2 = D;
                Out3 = C;

                x6 = B << 30;
                x7 = B >> 2;
                Out4 = x6 | x7;

                Out5 = A;
                end
            end
        endcase
endgenerate
```