Design of a Shared Hardware Library for Multi-Core Environments in FPGA Fabrics

Honors Thesis Submitted by
David Max Meisner
In partial fulfillment of the
Sc.B. in Electrical Engineering
Brown University
4/24/2007

Prepared under the Direction of
Professor Sherief Reda, Advisor
Professor Iris Bahar, Reader
Professor Jennifer Dworak, Reader

Abstract

Due to increasing constraints from power density and signal delay, microprocessor architectures are migrating towards multi-core design. Previously, when a program required increased performance in a certain function, designers typically implemented a single accelerator for their single processor. The move to multi-core designs introduces the possibility that instead of simply copying the design multiple times, designers can share hardware accelerators between cores.

We propose a Hardware Library, a pool of accelerated functions that is accessible by multiple cores. We find that sharing provides significant reductions in the area and logic usage required for hardware acceleration. Contention for these units may exist in certain cases; however, the savings in terms of chip economy are more appealing to many applications. We study the performance implications for our system using various arrangements. Our implementation of this system is realized on an FPGA fabric. These devices are particularly appropriate because of their need to reduce power and the area savings enable designers to easy add functionality without significant chip revision.

TABLE OF CONTENTS

Chapter 1: Introduction

1.1 Transition to Multi-core Processors

Until recently, most microprocessor design methodologies have revolved around a single pipelined processor. In creating a system for a particular application, algorithms were developed to run on this single scalar processing stream. Whenever a particular algorithm required performance enhancement, a custom hardware accelerator could be inserted in the pipeline to reduce computation time for critical applications such as in Figure 1.1. Instead of execution a function in software, the pipeline would use a custom block of hardware that is significantly faster. Only one logical configuration exists for such applications, so the design flow remained "simple".

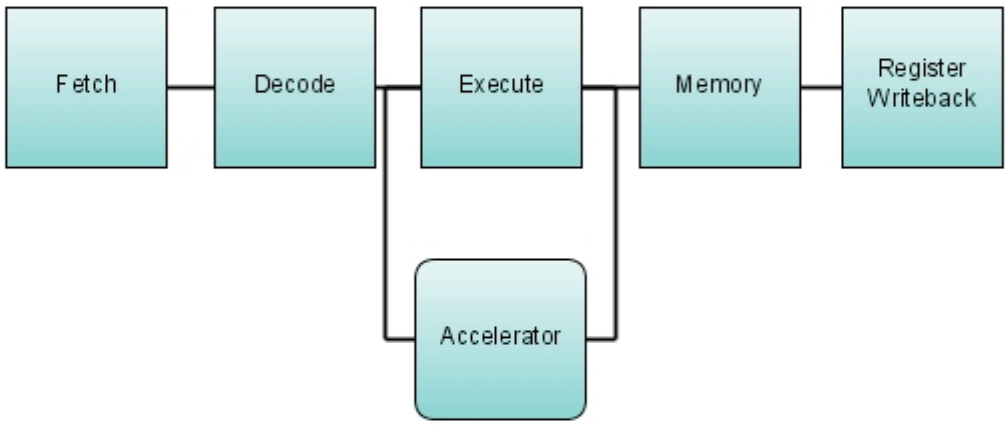

Figure 1.1: Pipelined Single Processor with Hardware Accelerator

Now, microprocessor designers are migrating to designing processors with multiple cores. Traditionally, designers have been able to take advantage of shrinking feature size to increase cache size, logic density and clock speed. However, power density and delay considerations have made it impossible to continually increase clock

speed. Because of this, designers are choosing to utilize the chip area gained from shrinking feature size to add more cores.

With the advent of multi-core, old maxims must be revaluated. Both new opportunities and difficulties arise in a multiprocessor system design. Under nominal conditions, multiple cores will not require the same hardware acceleration at the same time. Numerous cores will run the code implementing the same algorithm, but their execution will not necessarily be concurrent. Clearly, this condition lends itself to a sharing schema.

1.2 SOPCs on FPGAs

Field Programmable Gate Arrays (FPGAs) have risen to prominence in System on a Programmable Chip (SOPC) design for their unique properties and have displaced Application Specific Circuits (ASICs) in certain markets. Classic ASIC design requires a large investment in engineering time and fabrication cost. Not only do designers need to take into account the algorithmic, functional, and architectural needs of a design, they must also consider the underlying circuit and its fabrication and physical packaging constraints. FPGAs cut out the need to fabricate a new chip for every design. By using Combinational Logic Blocks (CLBs) composed of lookup tables, multiplexers and registers, any user defined design can be synthesized and mapped into the FPGA fabric. As shown in Figure 1.2, CLBs are organized into a two dimensional grid and are connected by a switching matrix. This system can configure and route any logic given enough blocks and interconnects. Instead of manufacturing a chip, designs simply need to be programmed on the FPGA. The removal of the large non-recurring engineering cost in product development associated with ASICs makes FPGAs an attractive option in small

to medium volume products and prototyping applications. Our proposed multi-core system is designed and implemented in such devices. Nevertheless, our design methodology concepts still apply to ASIC designs.

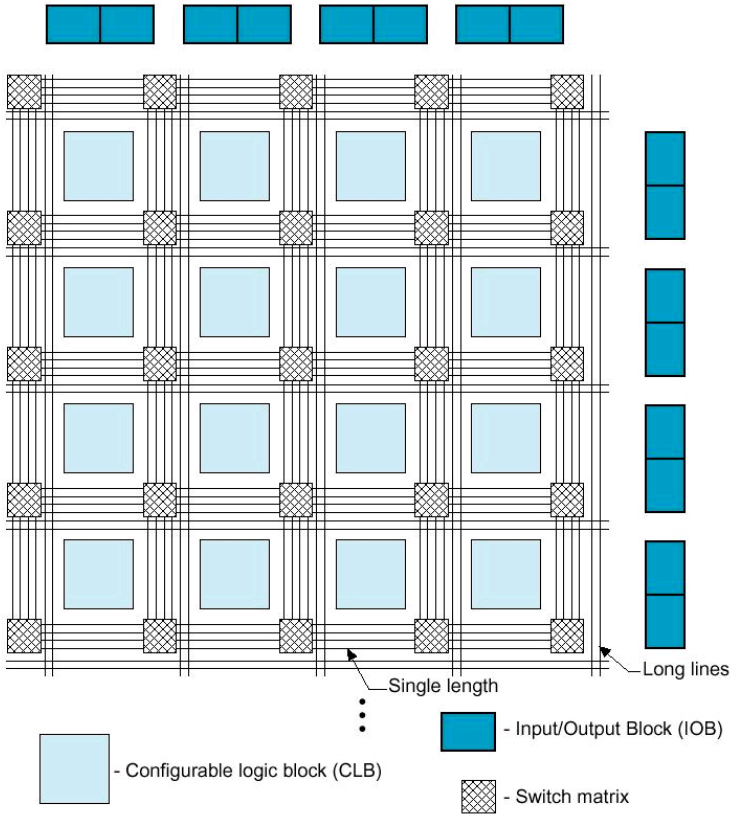

Figure 1.2: FPGA Fabric[1]

Modern technologies have produced FPGAs with hundreds of thousands of CLBs. We investigate Reduced Instruction Set Computer (RISC) cores. RISC architectures are characterized by their small instruction set, single cycle operations and register to register data operations which lend it to pipelining. These cores take as few as a couple thousand CLBs to synthesize; hence, a single FPGA can hypothetically hold hundreds of cores. Furthermore, custom logic can be added to control these cores to work in tandem with the standard design and accelerators can be synthesized and added easily to augment their

[1] Image Courtesy: http://www.clifford.at/papers/2004/bfcpu/fpga.jpg

functionality. While adding extra logic to an FPGA is a trivial task, this process is near impossible with an ASIC. This makes FPGAs an increasingly attractive option for multi-core designs because of their flexibility, especially when a design is to be extended to multiple applications. On the other hand, FPGAs cannot offer the same low-power or high clock speeds some finely crafted ASICs provide. At the very least, FPGAs can claim an invaluable niche in prototyping. Designs can be rapidly synthesized and programmed to test architecture functionality and performance. For these reason, we have chosen to investigate our design on a FPGA platform. However, the following methodologies apply to both FPGA and ASIC applications.

1.3 Software Libraries

Within the domain of software engineering, software libraries are created to provide multiple programs the same functionality without the need for multiple copies of identical code. This task is trivial in software because the instructions can be read by numerous programs with negligible overhead. Software libraries have the properties of *depth*, the ability for libraries to depend on other libraries in a hierarchy, as well as *reusability*, multiple applications being able to use the same library. Figure 1.3 demonstrates both these properties. For instance, the particle simulator makes function calls to the Vector Library which contains useful features such as dot products. In turn, the dot product function uses the Math Library for simpler functions like evaluating cosine. This example demonstrates the depth property. Meanwhile, a separate application running an image processing algorithm accesses the FFT library. This library also uses the Math Library for Cosine functions even though there is only one copy, hence demonstrating reusability. The later property is the inspiration for a Hardware Library.

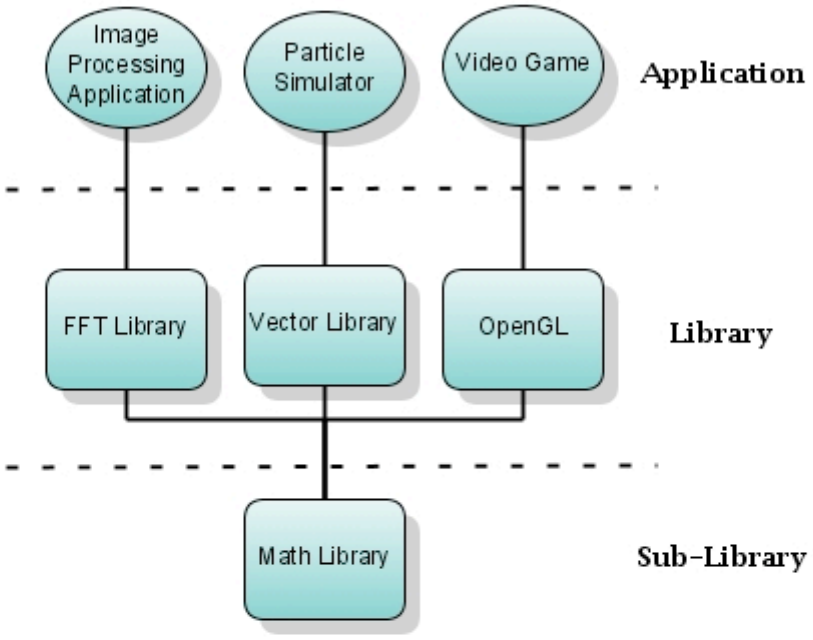

Figure 1.3: Software Library Sharing

1.4 Hardware Libraries

Reusability of hardware enables designers to save both power and chip area. We propose the concept of a *Hardware Library*. This entity exists as a pool of hardware resources that any core can use to accelerate its program's execution. Similar to software libraries, redundancy of resources is avoided by a sharing scheme. Unlike software however, hardware units cannot be used simultaneously by multiple applications. Furthermore, typically hardware libraries do not exhibit depth; computational units do not usually directly rely upon each other to execute their function before returning to the CPU. Therefore, depending on the popularity of a unit, multiple copies may be necessary. In Chapter three, we investigate the best scheme for sharing.

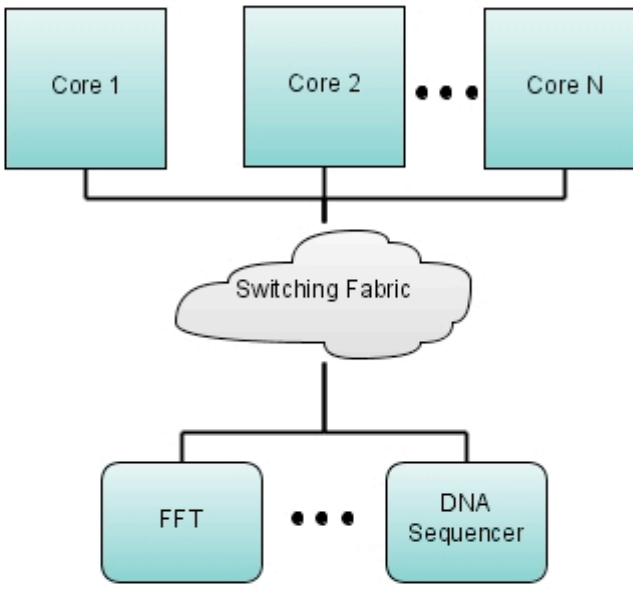

Figure 1.4: Proposed Hardware Library

As shown in Figure 1.4, the goal of a Hardware Library is to distribute and share multiple hardware accelerators among many cores. By developing a set of guidelines and a general design flow for such a library, engineers can more effectively utilize the available resources in a design. In Chapter three we propose an appropriate solution for the entity labeled "switching fabric" reasonable for delegating tasks and routing data.

This hardware library would be highly desirable in many systems. In time critical applications, software is often too slow and hardware accelerators are necessary. Moreover, embedded systems in particular may have limited chip area yet still require speed in specialized functions. A DSP unit for example, could have separate threads running on multiple cores and then access a FFT unit when necessary. A FFT unit is costly in terms of logic and if multiple processes can share a unit, there will be a major savings in terms of area without sacrificing performance. Since area can be a scarce resource in embedded systems, such a setup becomes an attractive option.

This thesis has the following contributions:

- We propose the concept of a Hardware Library, a pool of accelerators that take the place of software routines and are shared by multiple cores.
- We propose a system architecture revolving around a *Dispatcher* capable of sharing the accelerators with requesting CPUs.
- We analyze the performance and chip economy tradeoffs for differing sharing topologies.
- We show how FPGAs unique properties may make them a better candidate than ASICs for certain applications using this design.

This thesis is divided into two main chapteras:

- Chapter 2: Previous Work
- Chapter 3: Design of a Shared Hardware Library

Chapter 2: Previous Work

2.1 Soft Core Processor Design in FPGAs

Two variations are available when embedding microprocessors in FPGA systems. Either one can have a microprocessor built into the silicon in what is called a *hard processor* or it can be synthesized out of CLBs. The later option, creating a *soft processor* out of configurable logic, drives the focus of this investigation. Soft processors possess numerous advantages over hard processors. In theory, one can have complete control over the design of the processor, from the pipeline through the instruction set to the coprocessors. In practice, it is not worth the investment of time to build up a processor from scratch. Instead, many vendors offer intellectual property (IP) which are a complete synthesizable processor description [7, 8]. Open source options are also available with the advantages of cost and customizability [9].

Recently, much attention has been paid to optimizing the implementation of multiple soft cores in FPGAs. Particularly, one important obstacle to overcome revolves around making FPGAs more efficient than ASICs despite their tendency to consume more power. Trimming unneeded hardware by Instruction Set Architecture (ISA) subsetting for a given application is one method of reducing power [6]. Another strategy is to reuse hardware that is shared between soft processors . Vahid *et al* suggest that reconfigurable logic can be mutated by *profilers* at runtime to accelerate and map the binaries executed by multiple CPUs [3]. Unfortunately such a scheme suffers from the overhead of a profiling system. Furthermore, this system needs an FPGA fabric that can reconfigure on the fly. Such devices are not commonplace, rendering such setups lacking

the ability to generalize over all applications. Nevertheless, the ability to share hardware remains the main focus of our pooling strategy.

2.2 NIOS II Processor

Altera Cyclone II

Our experiments use the Altera Cyclone II FPGA chip. This chip is manufactured in a 90nm process and can hold up to 68,416 logic elements organized as 60 columns by 25 rows [11]. The FPGA is mounted in a DE2 education/prototyping board which includes a USB programmer/UART, clock and other important features.

Architecture

Altera offers their second generation NIOS II processor to embed in their programmable logic devices. The NIOS is a 32-bit RISC processor capable of most standard instructions. The NIOS II comes in three different "flavors": economy, standard and fast. These three configurations essentially trade increased logic usage for better performance and features. Figure 2.1 summarizes the characteristics of the various NIOS II processors including functionality and logic usage.

	○ Nios II/e	◉ Nios II/s	○ Nios II/f
Nios II Selector Guide Family: Cyclone II f_{system}: 50 MHz	**RISC** **32-bit**	RISC 32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Performance at 50 MHz	Up to 5 DMIPS	Up to 25 DMIPS	Up to 51 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks	Two M4Ks + cache	Three M4Ks + cache

Figure 2.1: NIOS II "Flavors"

For our experiments, the standard processor was chosen as it has useful features used for most applications including a hardware multiplier/divider, branch prediction and instruction cache. All experiments were run at the standard 50 MHz clock speed. Furthermore, it uses a reasonable 1200 logic elements leaving a small footprint.

The overall architecture of the NIOS II can be seen in Figure 2.2. It contains features common to any RISC processor such as a large general purpose register file. It also has a JTAG UART used for programming and communication. Most importantly it has a standard Avalon data and instruction bus so it can be connected to any devices conforming to this specification.

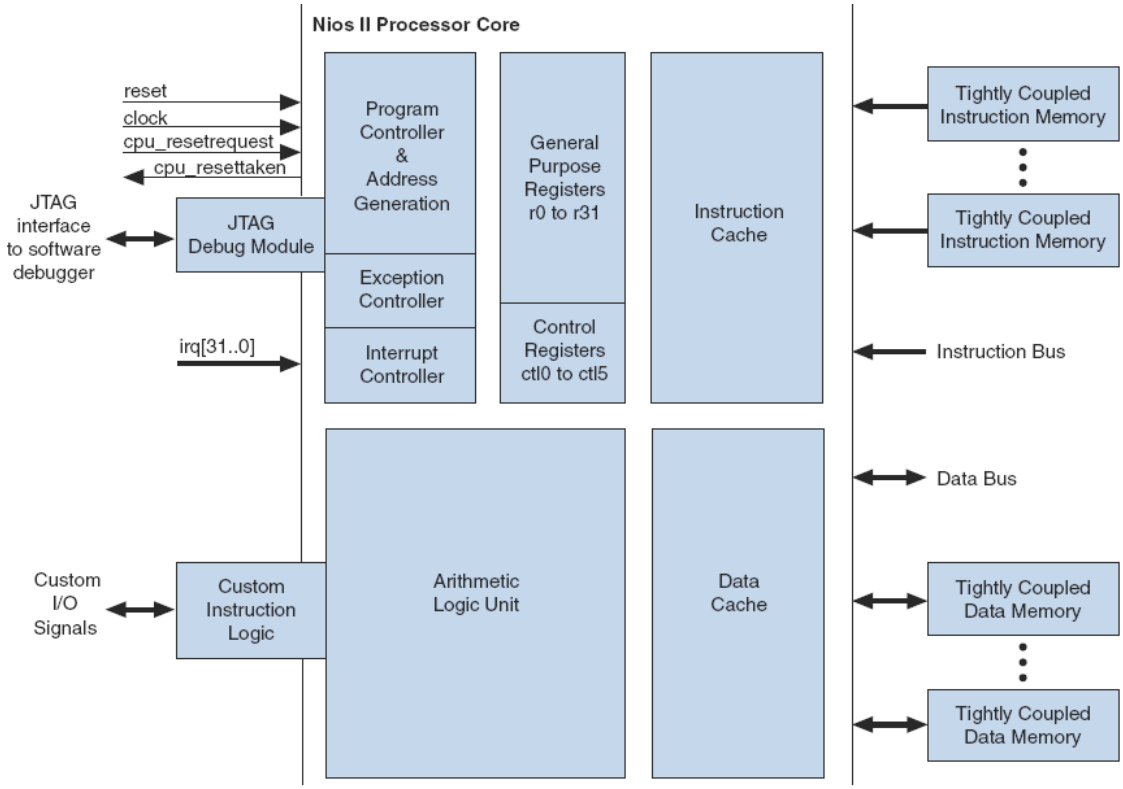

Figure 2.2: NIOS II Block Diagram[2]

Custom Instructions

[2] Altera, *NIOS II Processor Reference Handbook*

The custom instruction interface of the NIOS II can be seen in Figure 2.2. This logic resides directly next to the Arithmetic Logic Unit (ALU) because it will replace its logic when used. The custom instruction logic and I/O signals incorporated in the design enable designers to extend the functionality of the CPU without completely redesigning the processor.

In order to write one's own instruction, HDL is created that conforms to a specific I/O interface. This hardware will be accessed when the instruction is called in software. Moreover, C code macros are automatically created by Altera's integrated development environment (IDE) tools to abstract the implementation to simple function calls. Instructions can be of variable length and will return based upon hardware signaling if desired. The NIOS II's custom instruction functionality will be used extensively in the multi-core architecture to enable access to the Hardware library without completely redesigning the processor.

Avalon Switch Fabric

Along with the NIOS II, Altera provides many pre-designed IP components that integrate with the processor. These various components instantiated in the FPGA by the SOPC tools provided are connected through Altera's proprietary Avalon Switch Fabric. This fabric consists of Avalon master and slaves device. Logically, slave devices (such as memory controllers) can only be connected to and are controlled by master devices (such as a CPU). The switching fabric is composed of a network of multiplexers as seen in Figure 2.3.

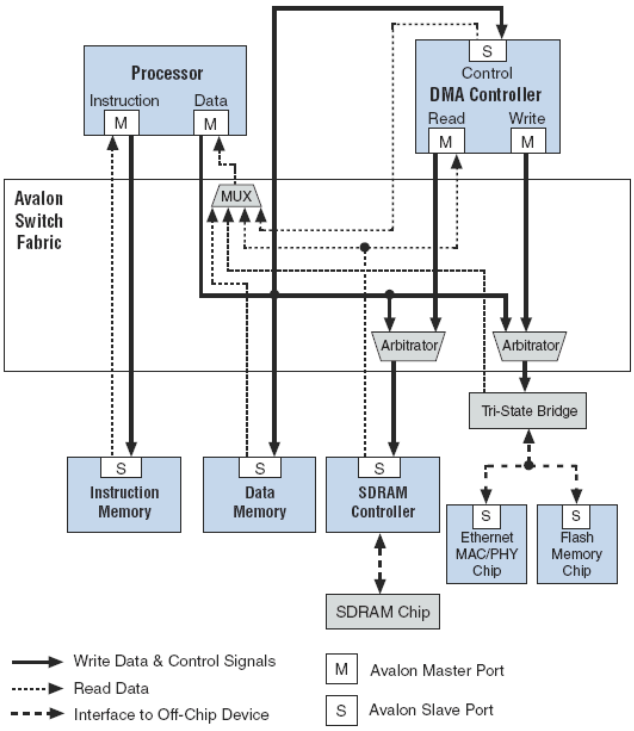

Figure 2.3: Avalon Switch Fabric[3]

2.3 C to Hardware Automation

Certain algorithms implemented in software can take far more time than can be afforded for a certain application. Typically in such a case, a system designer would have to offload this algorithm into a custom, hand-crafted, coprocessor designed specifically for the application. To reduce engineering time spent on such hardware design, companies have been offering C to hardware translation software. Such programs can take ANSI C code and compile them to HDL or RTL [2]. Figure 2.4 demonstrates the flow Altera's C2H compiler goes through in order to accelerate C code.

[3] Altera, *Quartus II Version 7.0 Handbook, Volume 4: SOPC Builder*

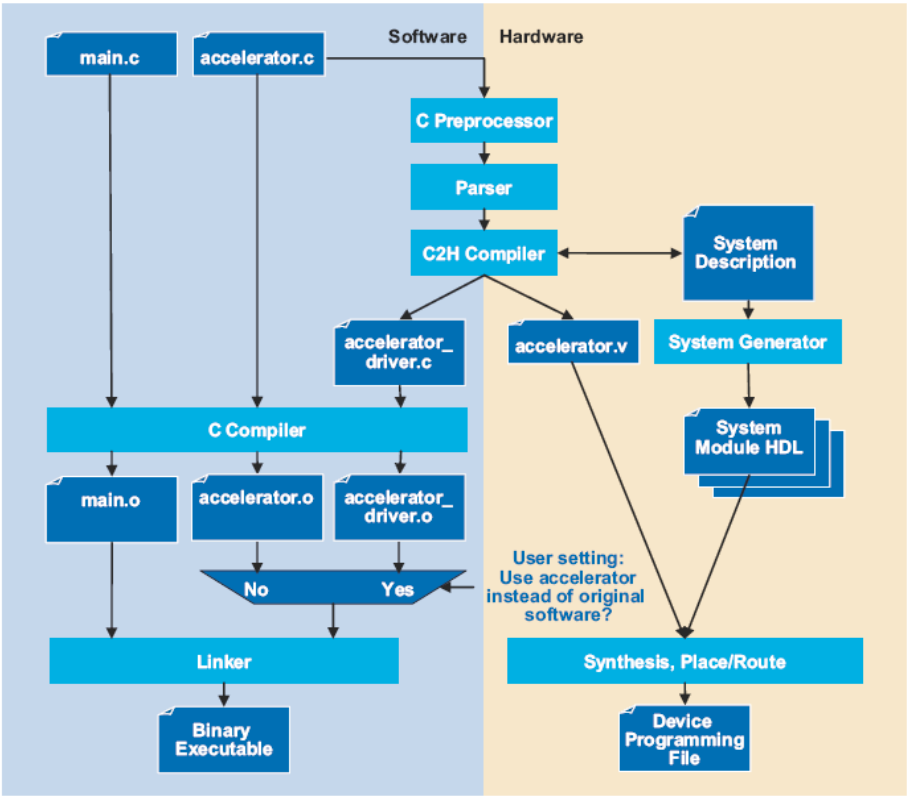

Figure 2.4: C2H Design Flow[4]

Automatic hardware generators such as the C2H compiler enable simple generation of hardware libraries. Instead of requiring laborious HDL coding, critical sections of code can simply be highlighted and translated into units for the system's accelerator pool. Such tools make generating a shared hardware library for any given application more feasible because the addition of accelerators becomes seamless.

2.4 Computationally Demanding Applications

Many applications require hardware acceleration in order to meet requirements for a particular application. For instance, many signal processing applications that require an FFT will not be able to run in real time implemented in software alone. Also, many applications in scientific computing will require evaluation of numerically complex

[4]Altera, Automated Generation of Hardware Accelerators With Direct Memory Access From ANSI/ISO Standard C Functions

equations. Implementing a specific function in hardware will cut down the runtime drastically.

DNA sequencing has become a critical task in the field of Bioinformatics. In order for scientist to understand the human genome billions of base pairs of DNA must be processed. The sheer amount of data overwhelms simple computers and custom hardware can greatly increase the rate at which the genome can be decoded [5]. Hence the ability to effectively process DNA rises to an immediate necessity as it is the limiting factor in genome research. Automation of sequencing relies upon the *edit distance* algorithm which efficiently computes the smallest number of shifts, inserts and deletes needed to align two text strings or DNA chains [1]. In the case of DNA, this process can be reduced to aligning the characters T, C, G and A.

The edit distance algorithm can be solved using a dynamic programming solution, where the overall problem is broken into smaller sub-problems [4]. These sub-problems are solved to get the overall solution. The algorithm relies upon a simple scoring matrix like the one shown in Figure 2.5. The algorithm proceeds as follows:

Algorithm: Edit_Distance

Input: String A of length N, String B of length M
Output: The edit distance of the two strings

let E be a MxN martix

for i=0..M
    E [i, 0] = 0

for j=0..N
    E [0, j] = 0

for i=0..M
    for j=0..N

```
                E[i, j] = min(  E[i-1, j] +1,
                                E[I, j-1] + 1,
                                match(A[i], B[j])+ E[i-1, j-1])

        return E[M, N]

function match(x, y)
        if x equals y
                return 0
        else
                return 1
```

		A	**T**	**G**	**C**	**A**
	0	1	2	3	4	5
A	1	0	1	2	3	4
T	2	1	0	1	2	3
C	3	2	1	1	2	3
T	4	3	2	1	2	3
A	5	4	3	2	2	2

A T G C - A

| | | | |

A T - C T A

Figure 2.5: Edit Distance Scoring Matrix[5]

Figure 2.5 demonstrates how the strings ATGCA and ATCTA would be scored. The edit distance is simply the value in the bottom right corner of the matrix. The best distance can be found by starting in the bottom right corner and working backward through the lowest scoring path. However this procedure is not needed to compute the scalar value of the edit distance. Given an *nxn* matrix, this algorithm would run in $O(n^2)$ time because it has two nested loops of length *n*. With a hardware accelerator that computes one of these loops in a single cycle, this algorithm runs in *O(n)* time. This gives an impressive speedup over software alone. While sequential sequencing of strings can be computationally complex when processed sequentially, time can be save at the expensive of specialized sequencing hardware.

[5]Kent, K. B., Proudfoot, R. B., and Zhao, Y. 2006. Parameter-Specific FPGA Implementation of Edit-Distance Calculation. In *Proceedings of the Seventeenth IEEE international Workshop on Rapid System Prototyping (Rsp'06) - Volume 00* (June 14 - 16, 2006). RSP. IEEE Computer Society, Washington, DC, 209-215.

Chapter 3: Design of a Shared Hardware Library

3.1 Architecture of Proposed Shared Library

In order to realize a Hardware Library, the system's cores must be connected to the accelerator pool and be controlled by an arbitrator unit. This control logic and data multiplexing is handled by a *Dispatcher* unit. The dispatcher is capable of queuing and prioritizing the cores requesting along with moving the data to the appropriate buses. Figure 3.1 illustrates a Dispatcher servicing two CPUs. The core logic of this unit is a Finite State Machine (FSM) which routes various control signals and data buses. Each Dispatcher directly controls any accelerators attached to it and they share a common clock. Furthermore, the Dispatcher takes cues from the CPUs that they request the accelerator using the *Start* signal and releases the data to them with the *Done* signal. Likewise, the accelerators must have a control logic interface with its own Start/Done control lines.

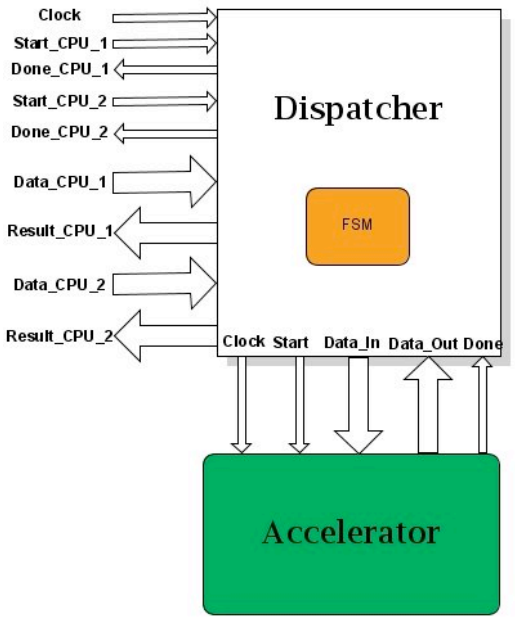

Figure 3.1: Dispatch Unit

Sharing Configurations

While each core must have access to some accelerator, there are many different topologies to realize this sharing. Four different configurations are explored for sharing accelerators between cores. Each configuration uses six cores and varies the ratio of cores to accelerators. The first configuration is six cores to one accelerator. As seen in Figure 3.2, all six CPUs have access to the same accelerator, but must wait if it is being used. This configuration requires a large dispatcher to process all six cores at once. The six to one configuration's *interconnection complexity*, the amount of routing used to synthesize, will be the highest of the four configurations for the Dispatcher, but not the system. It will also require a large amount of control logic to traffic the cores.

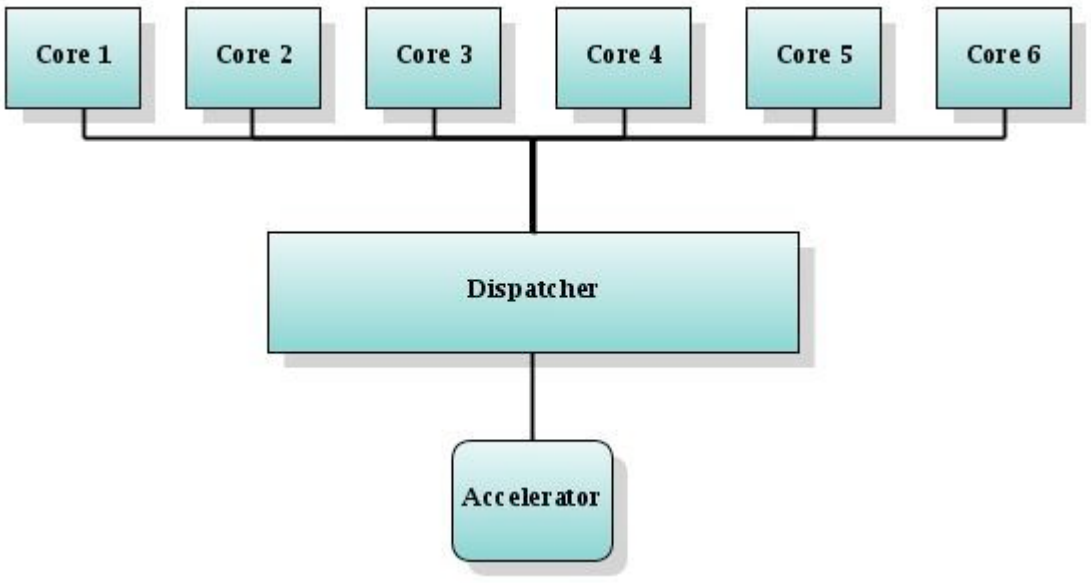

Figure 3.2: Six to One Sharing Configuration

By doubling the number of accelerators, the ratio of cores to accelerators becomes halved. Figure 3.3 shows the topology for a three to one sharing ratio. While this setup incurs additional resources for an additional accelerator, it alleviates the load on each

accelerator because they need to service half as many cores. Furthermore, the interconnect complexity and control signal logic are simplified.

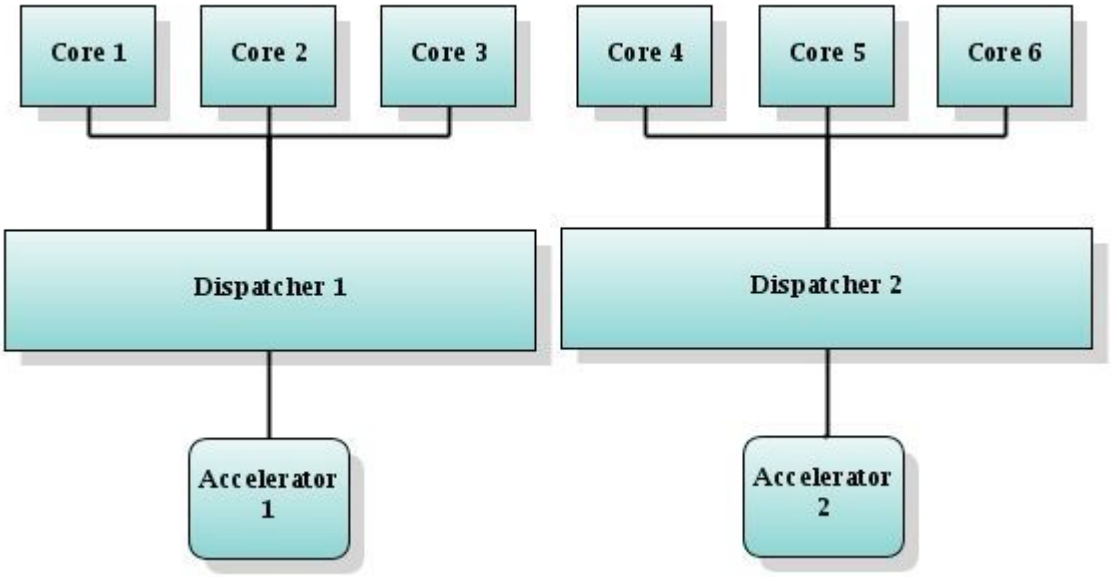

Figure 3.3: Three to One Sharing Configuration

Another possible configuration with six cores and two accelerators is given in Figure 3.4. This could potentially be a superior design to the three to one design in Figure 3.3 because even if one accelerator is being used it will not lock out the other. However, this increases the complexity of the dispatcher significantly. Furthermore, it requires more complicated routing to make sure all the units have access. This interconnect issue could be a burden if the cores or accelerators are placed significantly far away on the chip. For these reasons we choose to investigate the simpler design in evaluating the Hardware Library.

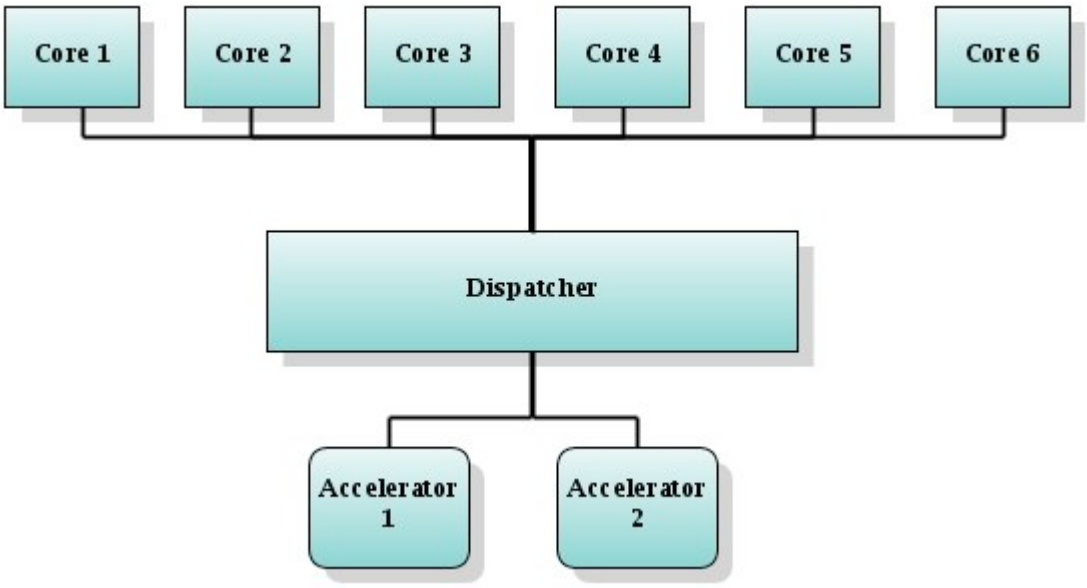

Figure 3.4: Alternative Configuration

By adding yet another accelerator, the topology shown in Figure 3.5 is achieved. This layout has a sharing ratio of 2 to 1 and is the smallest ratio still requiring a Dispatcher unit. Once again the same benefits are seen by reducing the sharing ratio. The interconnect complexity and control logic are reduced. Moreover, the accelerators only have two cores attempting to access them. At this point the dispatchers are quite less complicated than the initial configuration and there will be significantly less contention for them to resolve.

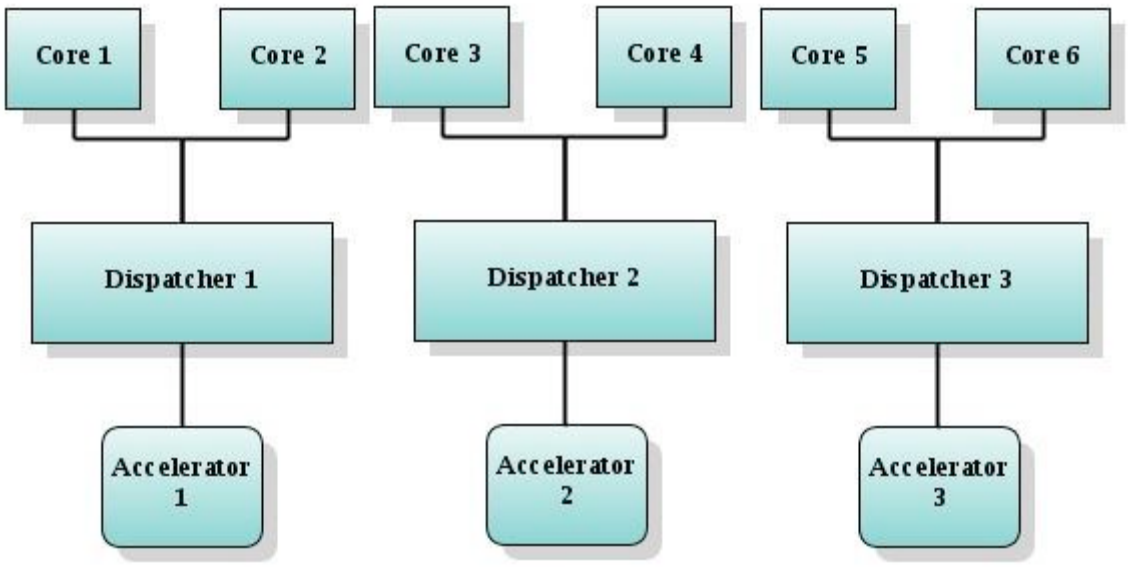

Figure3.5: Two to One Sharing Configuration

Lastly, the ratio of one to one must be explored. This setup is the simplest configuration and involves no sharing whatsoever. Each core has complete control over its own accelerator and contention will never be an issue. Dispatchers are not needed, because there's nothing to arbitrate. Because of the cores' ownership of the accelerators, they will never need to stall. On the other hand, this case ignores the entire concept of sharing and there exists a tremendous overhead in requiring an accelerator for each core. As is apparent in Figure 3.6, one would need six times as many accelerators without sharing than in the six to one topology.

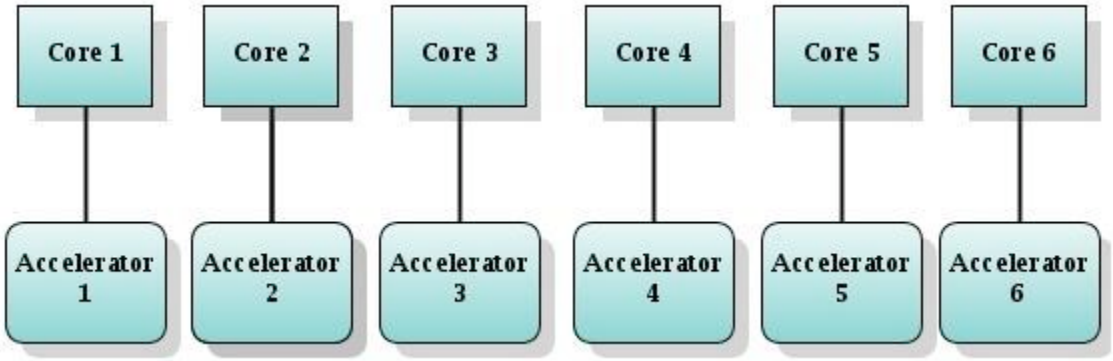

Figure 3.6: One to One Sharing Configuration

Table 3.1 summarizes the characteristics of each configuration. The general trend is that increasing the sharing ratio *might* decrease system performance and will add to the interconnect complexity and Dispatcher contention. However, there will be savings in terms of power and area gained by sharing. If the performance degradation is acceptable, the saving will be worthwhile.

	Guaranteed Performance	**Dispatcher Contention**	**Area Savings**	**Power Savings**	**Interconnect Complexity**
Six to One	Low	High	High	High	High
Three to One	Medium	Medium	Medium	Medium	Medium
Two to One	Medium	Medium	Medium	Medium	Medium
One to One	High	Low	Low	Low	Low

Table 3.1: Sharing Configuration Summary

Dispatcher Implementation

The Dispatcher's operation uses the Finite State Machine (FSM) model to construct its various operations. Specifically, the Dispatcher is a Moore machine as its outputs are dependent on its current state alone. These outputs control the multiplexing logic that route the data on the buses shown in Figure 3.1 to and from the CPUs and accelerator. Figure 3.7 illustrates the FSM diagram of the Dispatcher servicing two cores with a single accelerator and Table 3.2 explains the conditions needed for state transitions. The diagram shows that the Dispatcher will hold in the IDLE state until either start_cpu_1 or start_cpu_2 is asserted. At this point, one of the cores will have a *Lock* on the accelerator and will reside in the COMPUTEX state until computation completes. The other core will not be able to access it until computation is finished (Dispatcher goes back to IDLE).

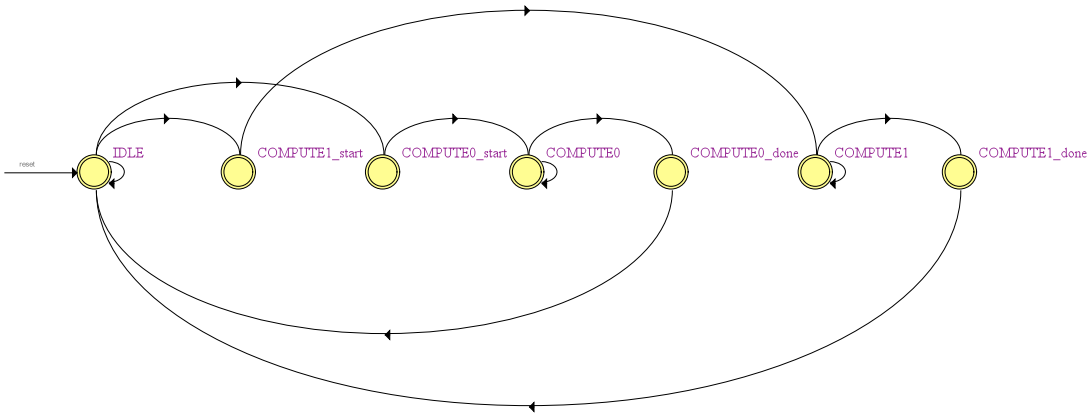

Figure 3.7: Diagram FSM Diagram

Source State	Destination State	Condition
COMPUTE1_done	IDLE	
COMPUTE1_start	COMPUTE1	
COMPUTE0	COMPUTE0	Accelerator => done
COMPUTE0	COMPUTE0_done	Accelerator => done
COMPUTE0_done	IDLE	
COMPUTE0_start	COMPUTE0	
IDLE	COMPUTE1_start	!start_cpu_0 & start_cpu_1
IDLE	COMPUTE0_start	start_cpu_0
IDLE	IDLE	!start_cpu_0 & !start_cpu_1
COMPUTE1	COMPUTE1_done	Accelerator => done
COMPUTE1	COMPUTE1	Accelerator => done

Table 3.2: FSM Transition Conditions

Figure 3.8 illustrates the timing of two CPUs being serviced by the Dispatch unit with a single accelerator. Once CPU_1 asserts its start signal, it gains control over the accelerator unit and no other requests will be acknowledged. Although start_cpu_1 goes high during computation, the FSM will ignore it. When computation is complete, the results will finally be valid and latched and the done_1 signal will be asserted. CPU_1 receives the data and can continue execution. When the second CPU dispatches a request, the previous results are invalidated and the process will repeat. To reduce the complexity

of the Dispatcher logic it does not have a non-blocking queue. Therefore, if CPU_1 is computing CPU_2, will stall until it gets ownership and finally done_2 is asserted. In the case of two requests from two cores at the same time, the Dispatcher acts as a priority queue. CPU_1 will get priority over all the cores, with CPU_2 being next and so on.

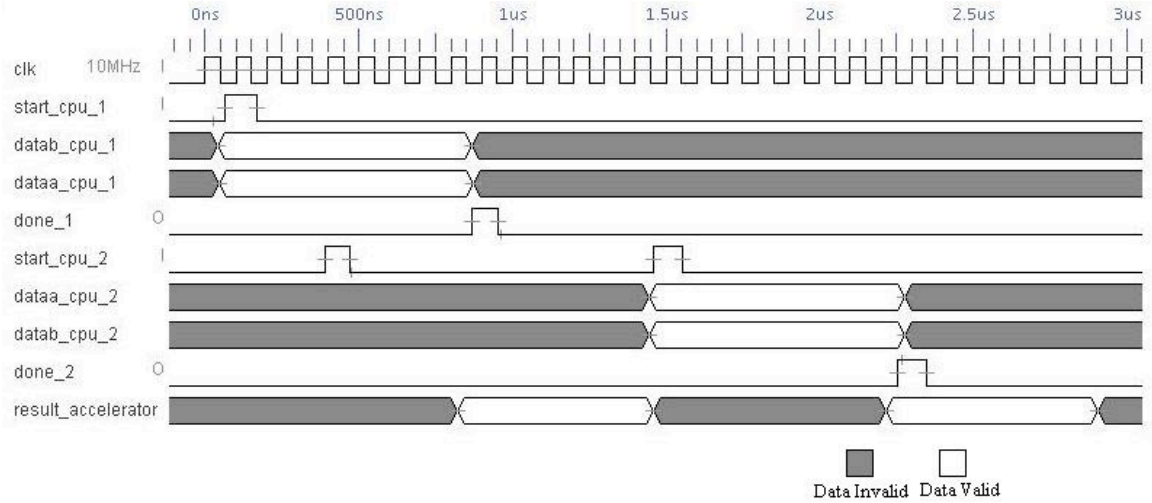

Figure 3.8: Timing Diagram for the Dispatcher

System Architecture

The hardware library relies upon a pool of hardware accelerators arbitrated by a dispatch unit. The NIOS II custom instruction interface is taken advantage of to incorporate the dispatcher with the NIOS II CPUs. For the prototype all data to and from the main memory goes through the CPU bus. This data will then be fed to the custom instruction and exported to the accelerator. Each accelerator unit reports to a single dispatcher which can service numerous CPUs as discussed previously. The block diagram for the system can be seen in Figure 3.9. The entities inside the “SOPC Builder” region

were designed and assembled with Altera's SOPC Builder tool inside Quartus II. Each system is composed of the following key components[6]:

- 6 NIOS II cores each with a Custom Instruction to access the Dispatcher
- A SDRAM controller which interfaces with 32MB of SDRAM. Each core has a 1MB region of this memory for its program to reside and run in.
- A performance counter for each CPU: This component acts as a binary counter that tracks clock cycles and can be turned on and off by software hooks. It is used to profile code execution times accurately.
- Each core has a JTAG UART to communicate with the programming PC and report data through standard printf calls.

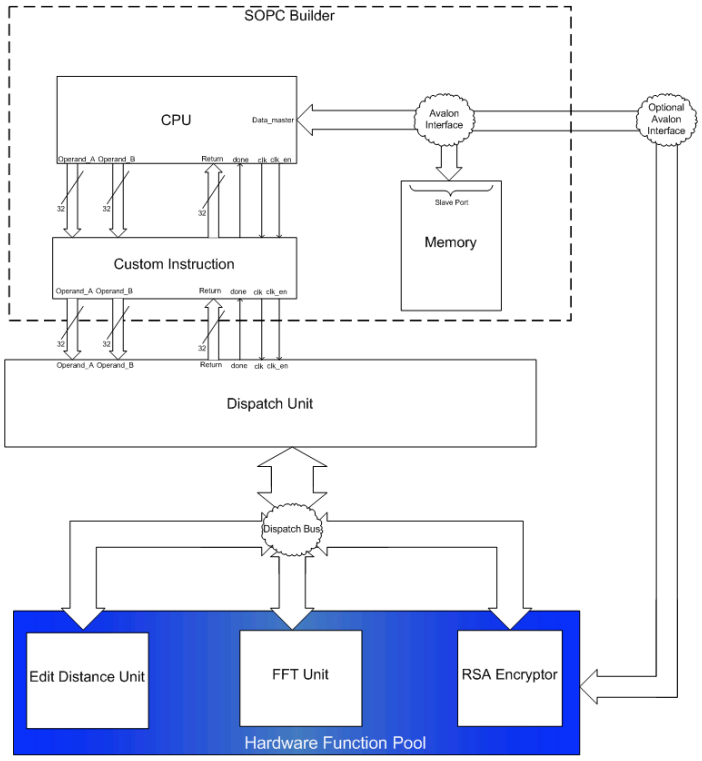

Figure 3.9: System Architecture

[6] The SOPC Builder has a quirk that it requires an On Chip Memory component for the CPUs to run even though they are not used. It is mentioned here only for completeness.

The components outside the SOPC Builder area in Figure 3.9 consists of modules coded in Verilog by hand. The Dispatcher acts as described previously and must be customized to accommodate the number of cores it will be serving. The hardware pool is not fully realized for the prototype, but the edit distance unit was hand translated from C to Verilog. As any unit interfacing with the Dispatcher unit, it has an interface with Start, Done, Data In and Data Out.

This diagram shows how one CPU is connected to the dispatcher. The dispatcher relies on two 32 bit data operands as well as clock control signals. The functions selector notifies the dispatcher which accelerator to send the data to and can potentially support up to 256 units. Although, not used in the prototype, this operand is important because it enables multiple accelerators to exist in the pool. The custom instruction entity itself acts as a *wrapper* as it simply passes the signals straight through and enables the NIOS II to talk to custom written Verilog.

Although our prototype only uses 64 total bits of data, this system can easily be extended. Application such as an FFT would require bulk access to the system's memory. This can be implemented by connecting the NIOS II Avalon bus. Future implementations can include such access; the only downside would be contention on the data bus that many CPUs would be using as well.

3.2 Performance Analysis & Results

Queuing Theory

Queuing theory is particularly relevant to the accelerator pools performance. Queuing theory models the flow of traffic through *servers* that service *customers* (for example customers waiting in a checkout line). In computing sciences, queuing theory has been applied to analyzing the scenario of a CPU accessing multiple devices such as hard drive disks. The concepts attempt to model the access times associated with such setups. For the proposed sharing system discussed however, we must model a single accelerator shared by multiple CPUs. The probability that an individual CPU will issue an accelerator request is given by the probability density function given in Equation 3.1 and illustrated in Figure 3.10 for various values of the parameter λ. 1/ λ is the issue rate for the CPUs or the arrival rate for the accelerator from one CPU.

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & , \ x \geq 0, \\ 0 & , \ x < 0. \end{cases}$$

Equation 3.1: Exponential Probability Function[7]

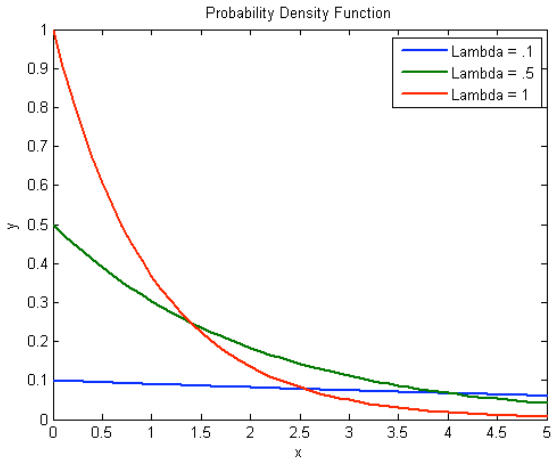

Figure 3.10: Exponential Probability Density Function

[7] http://en.wikipedia.org/wiki/Exponential_distribution

The arrival of accelerator requests can be modeled using this exponential distribution. Each request will be dispatched some number of cycles after the last one completes. Figure 3.11 shows the number of cycles between consecutive requests. As is expected, the cycle count decreases exponentially as the cycle time increases.

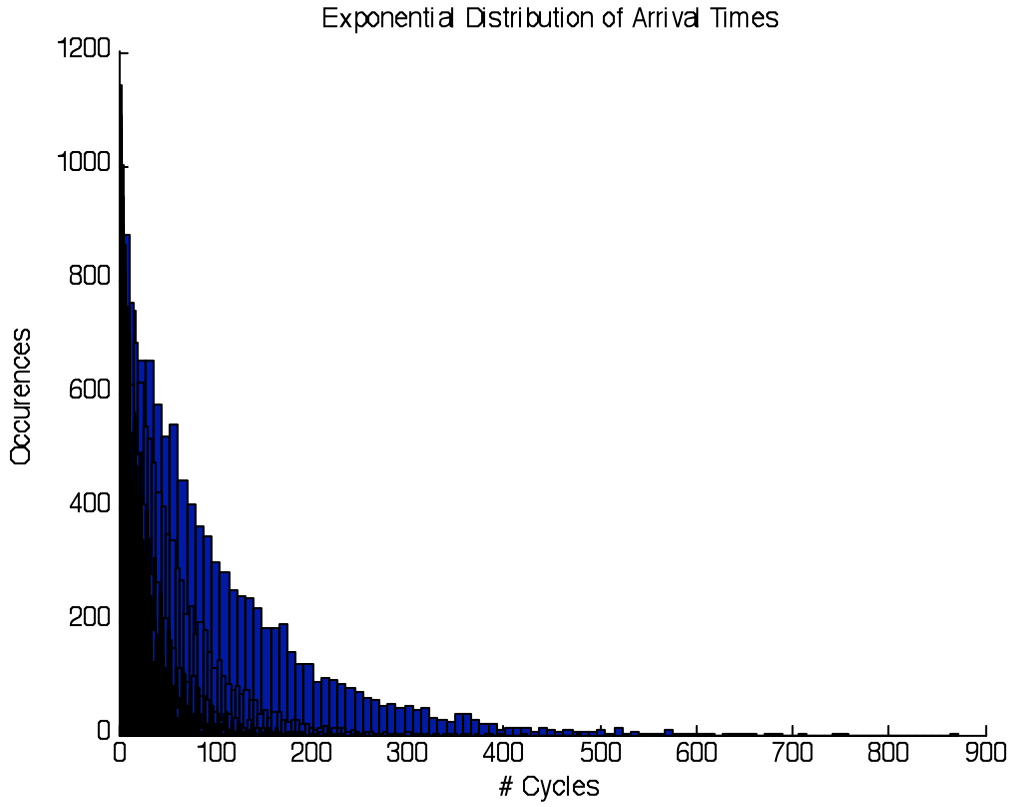

Figure 3.11: Histogram of Accelerator Requests Using an Exponential Distribution

Event Simulator

In order to properly access the behavior of our system without the overhead of creating and testing numerous designs, a simulator was written in C to calculate statistics regarding the Dispatcher. The hardware timing characteristics were captured in timing

analysis and replicated in the simulator. The simulator can accept any number of cores as well as a value λ for the expected request rate generated by these cores. The request times are generated and added to an event queue. Each event represents the core's software calling the hardware library. They will contain data on the core which wants to access the accelerator, the time of the request and the request number. Furthermore, the list can be in one of three states:

- *Request*: The node will want to acquire the dispatcher, but not attempted to access it yet.
- *Wait*: The node has tried to acquire the Dispatcher but it wasn't available and will try again later.
- *Terminate*: The node has a lock on the Dispatcher and is waiting for it to finish.

When the first event from each core is initially inserted, they are all requests (no core has the Dispatcher). The first event will be pulled out and become a terminate node. This new node will be inserted back in the queue at the appropriate time (the number of cycles needed by the accelerator). If any nodes are pulled out before the terminate finishes, they will be inserted behind the terminate node because they will not be able to access the dispatcher until it finishes.

To illustrate this concept, an example is shown in Figure 3.12. This demonstrates an exaggeratedly short event queue. The first node (Terminate is denoted T, W for Wait and R for Request) will be a terminate event and the Dispatcher will be freed. The wait event comes next. The dispatcher is freer so this event becomes a terminate event and will be pushed back into the appropriate time slot.

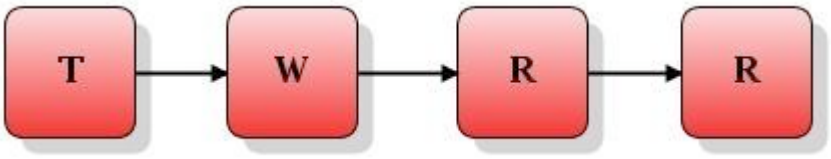

Figure 3.12: Simple Event Queue

If request is pulled out of the queue but the accelerator is occupied, the core will be stalled and be put in the wait state. The simulator will put the node (now a wait node) behind that last wait node as is shown in Figure 3.13.

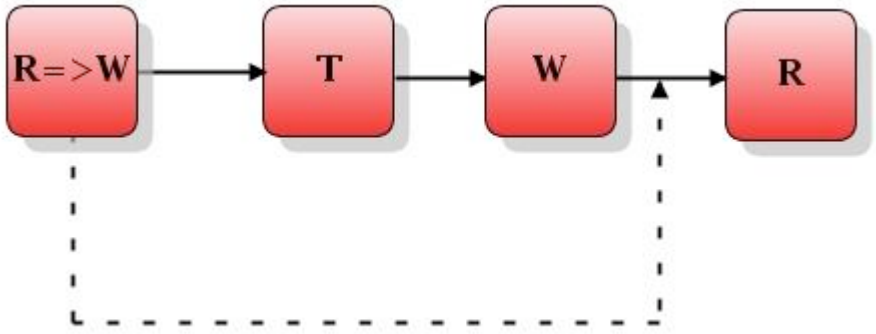

Figure 3.13: Request Being Stalled

The final situation arises when a wait node is pulled out of the front and the accelerator is now free. In this case the wait node will become a terminate node and be pushed back to the appropriate slot at which it will finish computation.

Stalling

The main performance barrier faced by sharing an accelerator is a core having to stall because another core has a lock on the unit. Therefore it is informative to analyze the trends in stalling due to different sharing configurations and traffic generation. Two factors directly impact the amount of stalling: (1) the sharing ratio and (2) the rate, λ, at which requests are being generated by the cores. Since for any given Dispatcher the number or cores determines the sharing ratio, we can simplify the investigation of stalling to its two independent factors: the number of cores per library and λ. Simulations are run

varying these two parameters and the results are shown in a contour plot in Figure 3.14. As would be expected, as the number of cores increases, so does the average stalling. Likewise, an increase in the rate of request generation also increases stalling. However, the number of stalls does not linearly increase with either of these quantities.

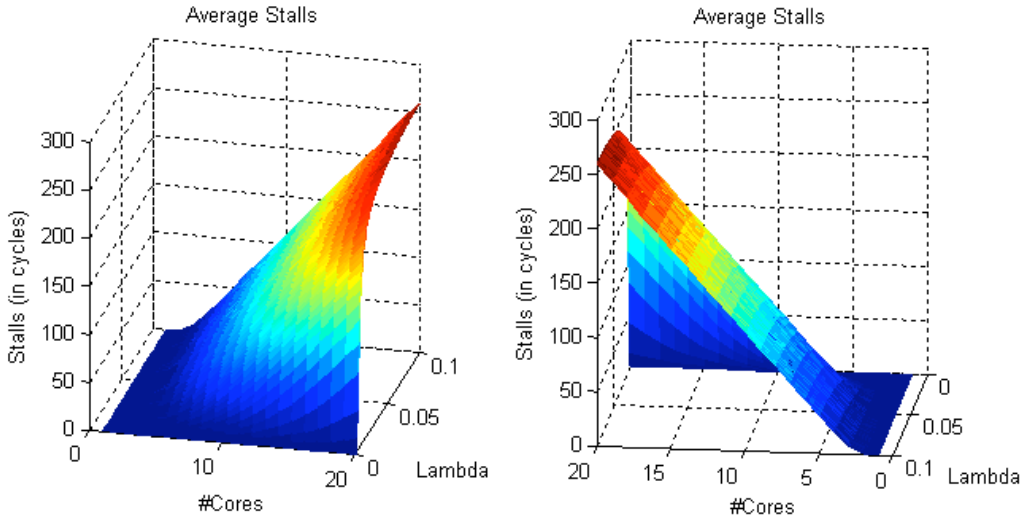

Figure 3.14: Average Stalls

In order to better understand the behavior of the stalling, Figure 3.15 presents slices of the contour plot at numbers of λ and cores. One can observe that the increasing number of cores drastically increases the stalling time for any value of lambda but especially for higher values. Also, increasing lambda for a given number of cores has a diminishing value after a certain point. What these trends infer is that the rate a core will need to access the accelerator can only affect performance up to a certain point. However, continuing to increase the core count can lead to a major slowdown.

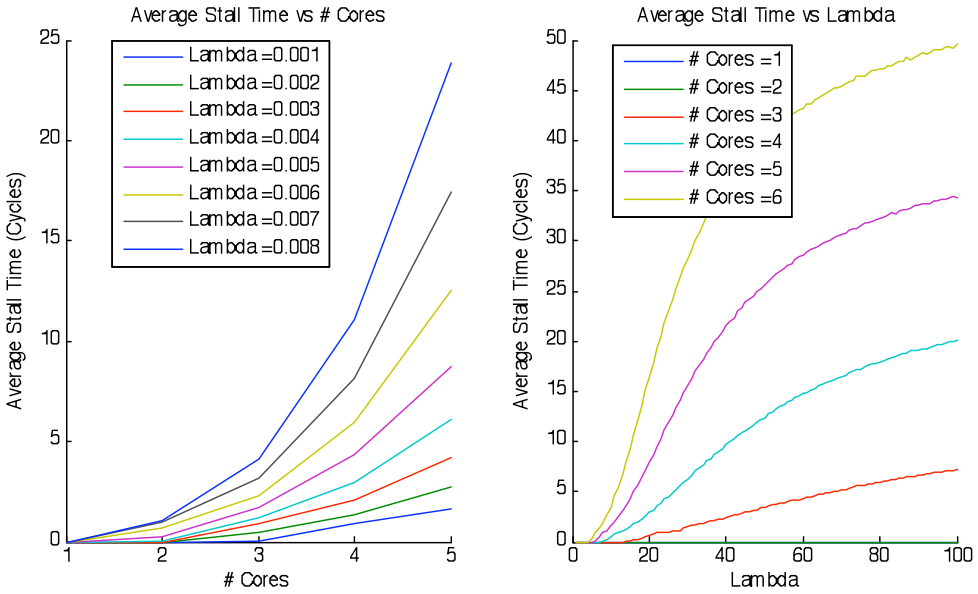

Figure 3.15: Select Values of Average Stalls

Dispatcher Queue

It is important to note that, most of the time, at least one of the cores will not be stalled. Figure 3.16 shows a sample of the number of cores trying to use the Dispatcher at once for a typical configuration. It is apparent that

a) the Dispatcher does not stay clogged for long

b) the number of cores accessing is rarely constant

c) sometimes the Dispatcher is unused

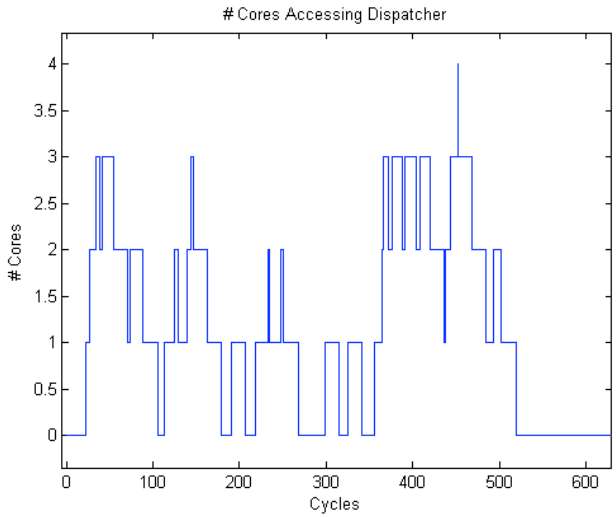

Figure 3.16: Cores Accessing Dispatcher

Utilization

Another important characteristic of the system to analyze is the *utilization* of the accelerator. The utilization is defined as the total time the system is busy divided by the total time spent. This is equivalent to:

$$U = \frac{time_{busy}}{time_{total}} = \frac{\#requests \cdot time_{request}}{time_{total}}$$

Hence if the unit receives many requests over a small period of time it will be heavily utilized. Conversely, if in a long time span, the accelerator barely gets any requests then it is not heavily utilized. Figure 3.17 shows how the request arrival rate (1/λ) and the number of cores affect how the accelerator is utilized. Clearly more cores yield a higher utilization percentage and likewise a higher rate of requests increases utilization. At a certain point, too many cores in the system or too high a request rate will completely saturate the unit and it will always be used throughout the execution. This scenario is unwanted and will probably require a lower sharing ratio.

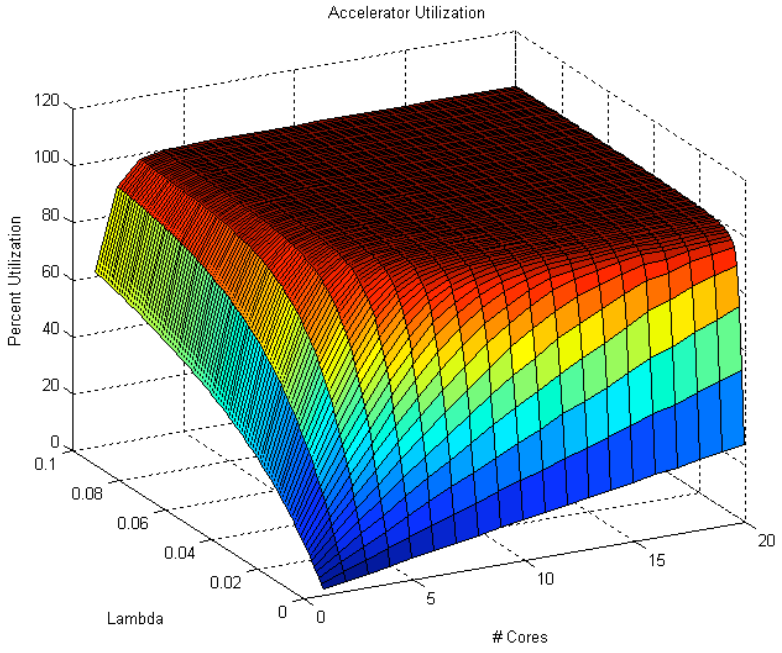

Figure 3.17: Average Accelerator Utilization

Execution Time

A metric for raw performance of a program on a microprocessor is its execution time. Since the clock frequency remains constant, the number of cycles is directly proportional to the time it takes to execute a program. The execution time in cycles can be seen in Figure 3.19 as determined by the Simulator; the number of cores and arrival rate is varied as before. The graph possesses a striking “performance wall”. Increasing lambda too much will create an overwhelming performance penalty. This particular data set was gathered with an accelerator execution time of sixteen cycles. It can be expected that this walls position will vary for different configuration; however, it shows that there exists a cutoff at which this system no longer remains useful.

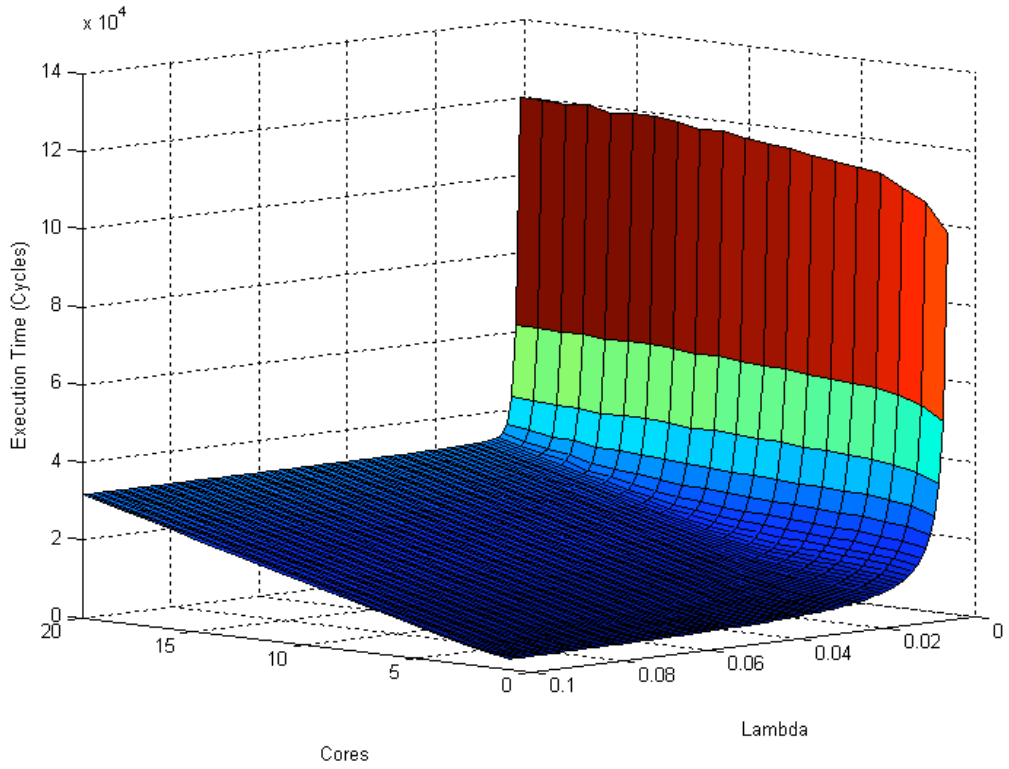

Figure 3.19: Execution Time

This data suggests that adding cores and therefore increasing your sharing ratio can be acceptable without a harsh performance penalty. Nevertheless, if all the cores start to execute a portion of code that continuously requests the accelerator the system may quickly suffer in performance.

Performance Limitations

The bounds for cycles spent in computation for accelerated and non-accelerated systems can be useful in evaluating a system. For this multi-core sharing environment, given *N* requests that take *t* cycles to complete, the upper bound on the number of cycles needed to complete on m cores from the perspective of an individual core is:

$$Cycles_{Accelrated} = m \bullet t \bullet N$$

This case would occur when all m processors are trying to use a unit at the same time. The upper bound would be applicable to the last core in this line.

In a design without acceleration, using software to compute the formerly accelerated sections of code, each core will be independent of each other. Hence from any one cores perspective, the number of cycles needed to compute would depend on the cycles needed l, and the number of requests N:

$$Cycles_{Sof\ tware} = N \bullet l$$

Therefore in order for there to be a speedup in a shared topology, one must obey the inequality that:

$$Cycles_{Accelerated} < Cycles_{Sof\ tware}$$

or more importantly:

$$m < \frac{l}{t}$$

This constraint means that, as one would expect, the complexity of the algorithm in software must be greater than it is in hardware for there to be any benefit of having

multiple cores. For example, suppose an algorithm that completes in software in n^2 number of steps. If one can accelerate it in hardware such that it become n, then the formula would become:

$$m < \frac{l}{t} = \frac{n^2}{n} = n$$

Hence, if n was 16, the number of cores could not exceed 15 or the system could run slower than if it had excluded a hardware library. This constraint sets up a good design metric engineers can use to design their system and determine the number of cores appropriate to use with their hardware library.

3.3 Implementation Overhead

Logic Usage

Saving chip area is one of the primary goals of a sharing scheme. This also translates into reducing the amount of logic needed to synthesize the design. Decreasing the logic requirement opens the possibility for either added functionality, reduced power dissipation or even a smaller device to house the design. Figure 3.21 shows the trends in logic usage for the various sharing topologies. Clearly, the higher the sharing ratio, the less logic is needed to obtain the same functionality (but not necessarily the same performance). Remarkably, the six to one configuration uses nearly one third less of the total logic than no sharing at all. In fact, the one to one ratio nearly uses all of the FPGA's logic! While their may not be a linear tradeoff between the sharing ratio and logic usage, it remains exceedingly important that a significant amount of logic can be free from one's design. Interestingly, the number of registers needed remains the roughly the same for varying topologies. This can be attributed to the fact that neither the

Dispatcher nor the accelerators use these registers. Since the other components in the system remain constant, so will this statistic.

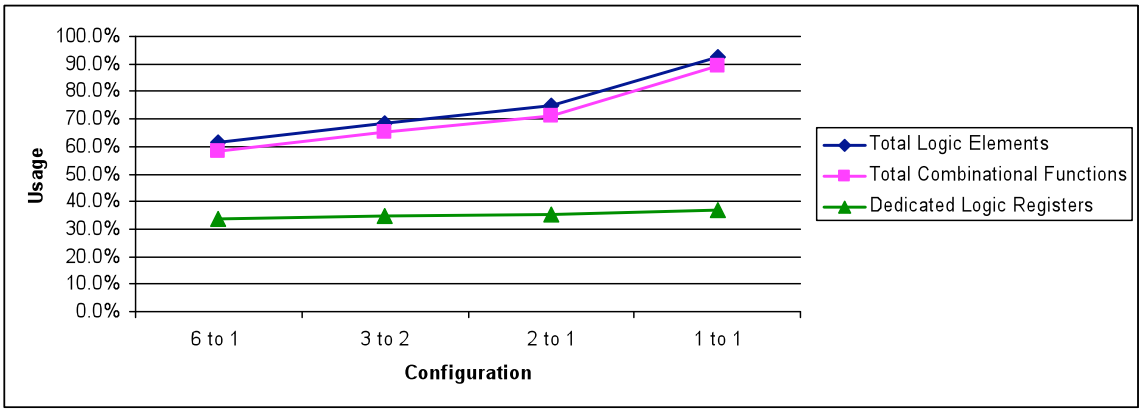

Figure 3.20: Logic Usage

Chip Area

Closely related to logic usage, the chip area used by the design must be considered by a designer. As with logic, saving chip area leaves room for other features. Furthermore, one could spread their design over the chip given more room to decrease power density which could greatly benefit cooling applications. Figures 3.22 and 3.23 are of the floor plan of the FPGA. Each small rectangle represents a CLB region. The coloring system is that darker the blue coloring, the denser the logic being used. The magenta represents the area used by the Dispatcher unit(s). Logically, the six to one design uses far less area than the two to one.

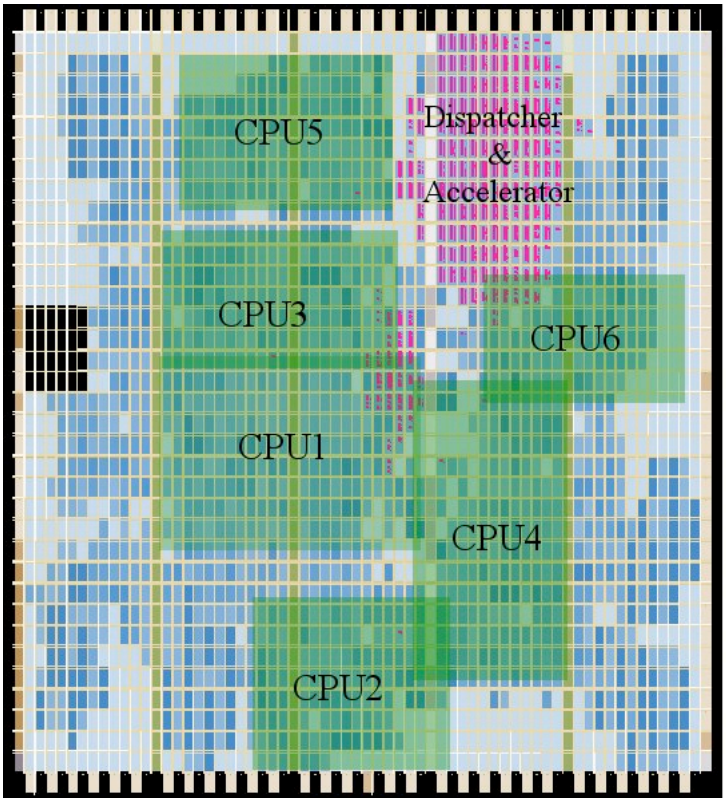

Figure 3.21: Six to One Chip Area Usage

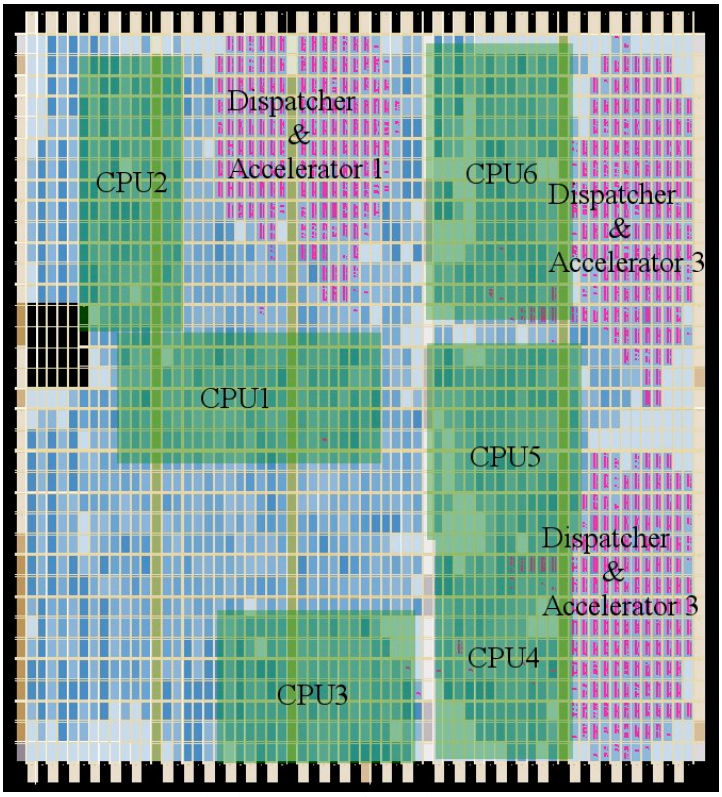

Figure 3.22: Two to One Chip Area Usage

Interconnect Complexity

The complexity and number of interconnections needed to realize a design can be a limiting factor. FGPAs have various varieties of interconnects that connect different regions of the chip. While it may not be likely, there exists a possibility that a design's logic becomes so convoluted that it simply cannot be routed. At this point, one must consider the number of connections needed. The percentage of the connections used can be seen to increase inversely to the sharing ratio in Figure 3.24. This trend is logical because the lower ratios require more logic and this increase implies increased connections between the logic.

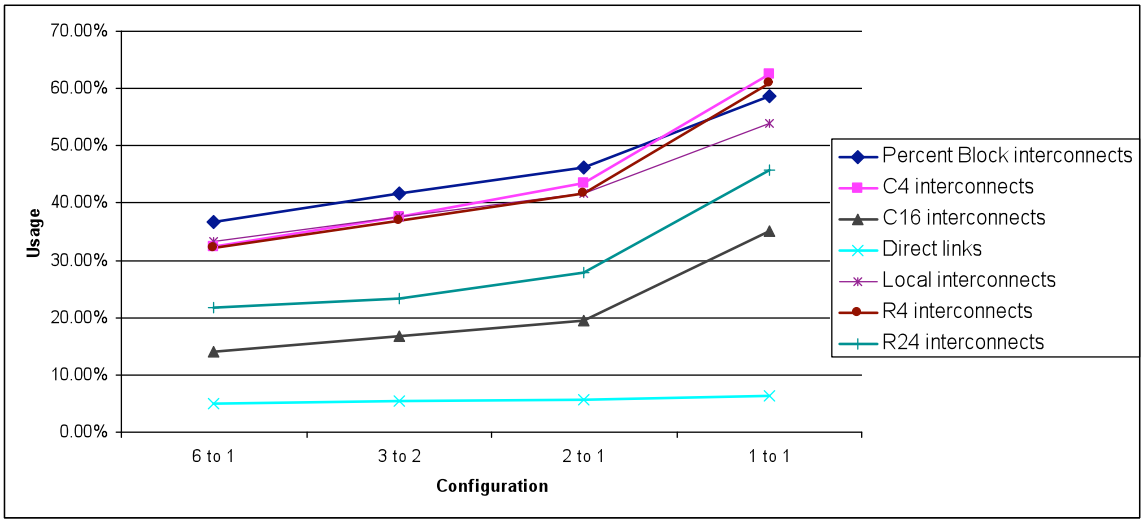

Figure 3.23: Interconnect Complexity

Delay Constraints

Finally, the delay considerations are crucial within any design. The worst case delays dictate the clock speed at which one's device may be run. FGPAs already possess a reputation for being slow devices, so minimizing the delays becomes critical in creating a competitive design. Figure 3.25 tabulates the worst case delays for setup (tsu), hold (th), contamination (tco) and propagation (tpd) times. Once again, the higher sharing ratio

improves the design by reducing setup delay. Propagation and contamination delay remain constant. This consistency can be attributed to the fact that these delay depend on logic depth and the compilation tools will optimize the circuit to minimize these delays. Finally, hold time does not have a clear pattern and most likely the compilation tool obscures the trend for this attribute.

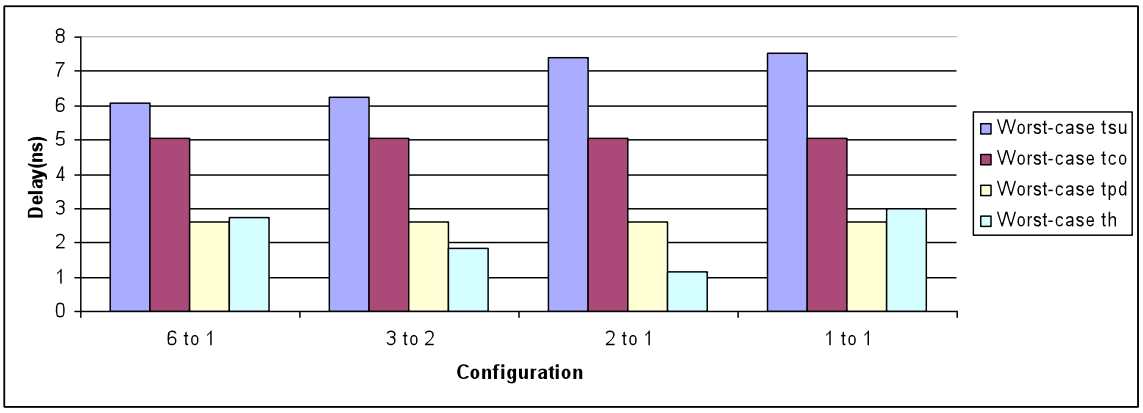

Figure 3.24: Delay Constraints

Chapter 4: Conclusion and Future Work

In emerging microprocessor designs, power density and chip area continue to increase in importance. Moreover, designers are no longer able to continue the trend of monotonically increasing clock speed to improve performance; nevertheless transistor feature size continues to shrink. In order to utilize the extra transistors, designers have added extra cores on the die to run sequentially.

At the same time, FPGAs have risen to prominence for their ability to rapidly enable product development without the cost of fabrications. However, FPGAs in particular need to overcome their power shortcomings to become replacements for ASICs. Perhaps for applications were performance is absolutely the dominant factor, ASICs remain a more viable option. However, when considerations such as power, engineering cost, and device size become important, the proposed sharing method enhances the abilities of FPGAs.

The concept of a Hardware Shared library has been proposed. This library consists of hardware accelerators that run commonly used algorithms faster than is possible in software alone. Recent tools have been developed which can automatically translate section of code that need this speedup to hardware automatically. Like software libraries these units are shared between multiple instances running programs. However, because hardware cannot be replicated without a cost, one tries to minimize the number of copies required for a given application.

The need to minimize the number of copies in the library naturally lends itself to sharing between cores. By creating a dispatching unit, multiple cores can use the same accelerator in turns. Because only one core may use the accelerator at a time, a

performance penalty will be incurred if a core needs the accelerator but another core already has the lock. I was shown that one can increase the number of cores sharing an accelerator only to a certain point after which the performance penalty will become overwhelming. However, with reasonable numbers of core, one gains significant logic and area savings. This will reduce dynamic power consumption and leave more room for additional functionality.

Future revision of this shared hardware library will most likely focus on enhancing the performance of the system. While there are clear advantages in terms of chip economy, performance does degrade slightly for nominal operation. The source of slowdown comes from the cores needing to stall when another core has a lock on the accelerator they wish to use. In this implementation these cores will simply stall until the lock is released.

A possible upgrade to this design would be to have a watchdog circuit that monitors the number of cores waiting for a hardware unit. Once a predefine limit of cores starts to wait, the watchdog will free the cores and have them execute the function in software thus avoiding stalling. This method would require software hooks in the processor architecture similar to an interrupt. With this feature the watchdog could send a control signal that would instantly divert the codes execution to a software implementation. In principle, there would be no software engineering overhead because the function would be written in C first to be compiled to HDL. The only penalty for such a feature would be the overhead of the watchdog logic and the software code size. The most logical limit for the watchdog to enforce would be the ratio m introduced of the software steps to that of the hardware. As discussed previously, if more cores are trying

to use the accelerator than this ratio, it would have been wiser to execute the code in software in the first place.

Even with lock contention remaining, the Dispatch system has great benefits for a multi-core design. The performance penalty is marginal until the discovered performance wall. Furthermore, the savings in terms of area, logic, and power may be preferred for many applications such as embedded systems. The proposed system shows a methodology for creating a Dispatcher to share a Hardware library and service multiple cores. Moreover, metrics to evaluate these systems were introduced. Like hardware caches, hardware designers will need to share accelerators between cores to effectively maintain chip economy. The Hardware Library has been established as an alternative to traditional accelerator design and has many benefits in terms of chip power, area, and logical complexity.

Chapter 5: References

1. Marzal, A. and Vidal, E. 1993. Computation of Normalized Edit Distance and Applications. *IEEE Trans. Pattern Anal. Mach. Intell.* 15, 9 (Sep. 1993), 926-932.

2. Altera, Automated Generation of Hardware Accelerators With Direct Memory Access From ANSI/ISO Standard C Functions, White Paper, May 2006, ver. 1.0

3. Lysecky, R. and Vahid, F. 2005. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society,, 18-23.

4. S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani, *Algorithms*, University of California, 22 May 2006

5. Roberts, L., *New Chip May Speed Genome Analysis*, Science, New Series, Vol. 244, No. 4905. (May 12, 1989), pp. 655-656

6. Yiannacouras, P., Steffan, J. G., and Rose, J. 2006. Application-specific customization of soft processor microarchitecture. In *Proceedings of the 2006 ACM/SIGDA 14th international Symposium on Field Programmable Gate Arrays*. FPGA '06. ACM Press, New York, NY, 201-210.

7. http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze

8. http://www.altera.com/products/ip/processors/nios2/ni2-index.html

9. http://www.opencores.org/browse.cgi/by_category

10. Altera, *NIOS II Processor Reference Handbook*, San Jose, 2006

11. http://www.altera.com/products/devices/cyclone2/features/architecture/cy2-architecture.html

12. Altera, *Quartus II Version 7.0 Handbook, Volume 4: SOPC Builder*, San Jose, 2007

13. Önyüksel, I. H. and Irani, K. B. 1990. Markovian Queueing Network Models for Performance Analysis of a Single-Bus Multiprocessor System. *IEEE Trans. Comput.* 39, 7 (Jul. 1990), 975-980.

14. Denning, P. J. and Buzen, J. P. 1978. The Operational Analysis of Queueing Network Models. *ACM Comput. Surv.* 10, 3 (Sep. 1978), 225-261.

15. Court, T. V. and Herbordt, M. C. 2004. Families of FPGA-Based Algorithms for Approximate String Matching. In *Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE international Conference on (Asap'04) - Volume 00* (September 27 - 29, 2004). ASAP. IEEE Computer Society, Washington, DC, 354-364.

16. C.W. Yu, K.H. Kwong, K.H. Lee and P.H.W. Leong, A Smith-Waterman Systolic Cell*, Proceedings of the Tenth International Workshop on Field Programmable Logic and Applications* (FPL'03)*,* Lisbon, pp. 375-384, 2003

17. Hoang, D. 1993. Searching Genetic Databases on Splash 2. *IEEE Workshop on FPGAs for Custom Computing Machines*. Brown University, RI

18. Sheldon, D., Kumar, R., Vahid, F., Tullsen, D., and Lysecky, R. 2006. Conjoining soft-core FPGA processors. In *Proceedings of the 2006 IEEE/ACM international Conference on Computer-Aided Design* ICCAD '06. ACM Press, New York, NY, 694-701.

19. Sheldon, D., Kumar, R., Lysecky, R., Vahid, F., and Tullsen, D. 2006. Application-specific customization of parameterized FPGA soft-core processors. In *Proceedings of the 2006 IEEE/ACM international Conference on Computer-Aided Design* ICCAD '06. ACM Press, New York, NY, 261-268.

20. Yiannacouras, P., Rose, J., and Steffan, J. G. 2005. The microarchitecture of FPGA-based soft processors. In *Proceedings of the 2005 international Conference on Compilers, Architectures and Synthesis For Embedded Systems* CASES '05. ACM Press, New York, NY, 202-212.

21. Jin, Y., Satish, N., Ravindran, K., and Keutzer, K. 2005. An automated exploration framework for FPGA-based soft multiprocessor systems. In *Proceedings of the 3rd IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis* CODES+ISSS '05., 273-278.