A Lightweight Processor Core for Application Specific Acceleration

by

David Grant

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

(David Grant)

Abstract

Advances in configurable logic technology have permitted the development of low–cost, high–speed configurable devices, allowing one or more soft processor cores to be introduced into a configurable computing system. Soft processor cores offer logic–area savings and reduced configuration times when compared to the hardware–only implementations typically used for application specific acceleration. Programs for a soft processor core are small and simple compared to the design of a hardware core, but can leverage custom hardware within the processor core to provide greater acceleration for specific applications.

This thesis presents several configurable system models, and implements one such model on a Nios Embedded Processor Development Board. A software programmable and hardware configurable lightweight processor core known as the FAST CPU is introduced. The configurable system implementation attaches several FAST CPUs to a standard Nios processor to create a system for experimentation with application specific acceleration. This system incorporating the FAST CPUs was tested for bus utilization behaviour, computing performance, and execution times for a minheap application. Experimental results are compared to the performance of a software–only solution, and also with previous research results.

Experimental results verify that the theory and models used to predict bus utilization are correct. Performance testing shows that the FAST CPU is approximately 25% slower than a general purpose processor, which is expected. The FAST CPU, however, is 31% smaller in terms of logic area than the general purpose processor, and is 8% smaller than the design of a hardware–only implementation of a minheap for application specific acceleration. The results verify that it is possible to move functionality from a general purpose processor to a lightweight processor, and further, to realize an increase in performance when a task is parallelized across multiple FAST CPUs. The experimentation uses a procedure by which a set of equations can be derived for predicting bus utilization and deriving a cost–benefit curve for a coprocessing entity. They are applied to a specific system in this research, but the methods are generalizable to any coprocessing entity.

Acknowledgments

First and foremost, I would like to thank my supervisors Dr. Wayne Loucks and Dr. William Bishop. Even with their hectic schedules, they managed to always be around when I needed help or advice. Their support and guidance made my research possible.

Thanks to Science and Engineering Research Canada (SERC, formally NSERC) for providing funding for my research. Thanks to Altera Corporation for donating hardware and software to our research lab on such a regular basis that it was necessary to recompute many results as the synthesis tools and Nios software changed and improved.

To my friends, both new and old, thank–you for putting up with me in "school mode".

Most importantly, I would like to thank my parents and family for their constant encouragement and support. I could not have completed this degree without it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis investigates the use of a configurable and programmable application specific lightweight processor core in an embedded Nios environment. The goal of the thesis is to examine a configurable system containing lightweight processor cores to determine the feasibility of using such processors for application specific acceleration. Using multiple, stripped–down processor cores, to provide acceleration for several applications simultaneously, is part of the overall goal.

This research is part of a larger body of research investigating the most effective use of configurable resources, with an emphasis on the minimal use of resources, to solve a problem (or a series of different problems) as efficiently as possible. The efficiency may be based on the area, complexity, and speed of the solution, and may also include factors such as the ease of the development process, or the effort required to develop, debug, and maintain the solution.

1.1 Motivation

Advances in configurable logic technology have permitted the development of low–cost, high–speed configurable devices. These devices can be used to provide application specific acceleration to a system, but must be reconfigured with each new hardware design as the needs of the running applications change, which can take a significant amount of time. The fact that configurable devices

are becoming cheaper and faster [Alt02b, Xil04b] means that it is now feasible to introduce one or more processor cores into a system.

A processor core has several particularly attractive features to application specific acceleration:

- The implementation of a design is relatively small (on the order of kilobytes of compiled code), compared to a synthesized hardware design of an application specific acceleration core suitable for downloading into an FPGA (250 kB for small FPGAs [Alt99]).
- The time to reprogram a core is small, as it only needs to download a program then execute a jump to the beginning of the program.
- It is faster[1] to design, implement, and debug algorithms in software than in hardware [CN96, HS98].

This research into lightweight processor cores is further motivated by the following questions from the larger body of research:

1. A lightweight processor core can be used to implemented large, complicated algorithms for application specific acceleration. Can the lightweight processor core that is used for the implementation consume less chip area than the hardware core implementation of the same algorithm? If so, then it may be possible to use several processor cores in the same area to mirror the speed of a hardware implementation. Or, if speed is not an issue, to use only a single processor to reduce the area requirements.
2. Will the total storage required for a lightweight processor core and the programs for it be significantly smaller than the hardware implementations for each of the programs?

There are drawbacks to developing for processors. Foremost, it is known that a good hardware description of many algorithms will be faster than the software implementation [De 94]. Consider,

[1] And thus cheaper, since time is money.

however, an algorithm implemented with a re–usable hardware component (the lightweight processor), and an application specific software component. Such a system can leverage some of the benefits of hardware (speed) and many of the benefits of software (size, configuration time, development time). This realization serves as the purpose for developing a new configurable (hardware) and programmable (software) lightweight processor core for application specific acceleration.

1.2 Statement of Thesis

It is my thesis that lightweight processor cores can be used effectively as accelerators for application specific acceleration. The tradeoffs associated with moving some software–only designs into a configurable system containing processor cores will be positive. Furthermore, systems already employing custom–hardware for application acceleration may benefit from a migration to lightweight processors to reduce the complexity of the implementation and improve the switching time between designs.

The theory is to create a system using a procedure analogous to process and thread switching in modern operating systems. Using this analogy, a lightweight processor core that provides a basic set of opcodes tailored specifically for running an application is essentially a "process" within a configurable system. The program running on the lightweight processor is effectively a "thread".

Switching between the lightweight processor programs is like thread switching: the programs are small, so they can be switched quickly. For systems which require a form of custom–hardware acceleration to achieve the desired (or necessary) system speed, hardware can be added to the lightweight processor in the form of opcodes. A complete hardware–only implementation can be created within the processor core by designing an opcode that implements the complete accelerator, and by calling that opcode from a lightweight processor program.

Entire lightweight processors can be swapped depending on the opcode needs of the applications, which is akin to process switching. It takes a long time relative to the lightweight processor program ("thread") switching, so it is desirable to switch processor cores as infrequently as possi-

ble. Advantages can be realized if the basic set of opcodes remains constant across all processors, allowing any application that only uses the basic opcode set to be scheduled on any lightweight processor, even if the processor contains additional opcodes.

Developing an algorithm to effectively schedule the processor cores and the applications within the cores is necessary, but is beyond the scope of the thesis, and will be reserved for future work.

1.3 Thesis Contributions

My thesis makes the following contributions to the larger body of research encompassing configurable computing for application specific acceleration:

1. Introduces the FAST CPU, a configurable and programmable lightweight processor core for application specific acceleration, useful for performance and tradeoff testing between performing certain tasks in hardware and software.

2. A method to derive a set of equations to predict the bus utilization of a specific implementation of a configurable system.

3. A method to derive a cost–benefit curve to determine when it is beneficial to migrate a software algorithm on a general purpose processor to a lightweight processor from a purely computation standpoint. The method is valid for algorithms that permit fairly course–grained parallelism.

4. A comparison with experimental results from various configurable computing designs using a similar Nios embedded system architecture.

5. Highlights several problems encountered when pushing the resource limits of an Altera APEX FPGA and the design tools.

1.4 Outline

Chapter 2 provides a summary of configurable computing and defines the terminology related to configurable computing used in this research. Several models of configurable computing are also presented. Chapter 3 describes the Nios Embedded Processor Development Board, which is used in this research. One model presented in Chapter 2 is selected for implementation and testing on this board. Following the platform description, Chapter 4 presents the design of a specific system for the Nios Embedded Processor Development Board. It also introduces and describes the FAST CPU as a lightweight processor core for application specific acceleration. Chapter 5 covers the testing and verification of the system containing several FAST CPUs. Chapter 6 summarizes the conclusions of the thesis research, highlighting several challenges encountered and presenting the directions for future research. Raw results, and sample source code for the FAST CPU and Nios processors are given in the appendices.

Chapter 2

Background

2.1 Terminology

It is generally accepted in current research that terms containing the root "configurable" (*configurable, reconfigurable*) refer to a piece of hardware that can have its behaviour changed in some way. Whereas terms based on "programmable" (*programmable, reprogrammable*) can refer to either hardware or software. When referring to hardware it is used synonymously with *configurable*, and when referring to software it refers to the ability of a programmer to control and change the behaviour of the device by changing the software.

There are many terms to describe systems involving varying degrees of hardware and software, and the degree to which the hardware and software can be changed. Some terms are distinct, and some have been redefined by the various groups and fields which have emerged in the research. To distinguish between hardware and software in this research, *configurable* and all derivative words are used to refer to hardware, whereas *programmable* and all derivative words refer to software. Terms of interest in this research are only those related to hardware configuration and software programming, no references are made to hardware changes that involve altering the physical system (i.e., by physically rewiring components or moving components around).

2.1.1 Configurable Computing

Traditionally, the term *configurable computing* is used interchangeably with the term *reconfigurable computing* [CH02, DG97]. Both terms refer to a system containing a *configurable* device (or *reconfigurable device*, again used interchangeably) that implements a hardware design, and that can be changed without changing the physical hardware. The *configuration* (or *reconfiguration*) of the device refers to the event of downloading a new hardware design, usually represented by a bitstream, into the device to change the behaviour. Once downloaded, the device (or system) is sometimes restarted so the changes take effect. Earlier systems, such as [GKC+94], [Cha94], and [CR93] required this restart, whereas systems using more recent technology [Alt97], [Xil02], and [Mic02] can handle the device being configured without resetting the entire system. In all of these systems the process of configuring the device effectively resets it, since the configuration causes the loss of all state information.

Recent advances in FPGA technology have created hardware devices that support configuring part of the device, and leaving the rest of the hardware design untouched. In this set of definitions, these devices have been called *partially–configurable* (or *partially–reconfigurable*). A *partially–configurable computing* system is a *configurable computing* system which contains a partially–configurable device.

The terminology used in this research, however, is from a second set of definitions which have been gaining popularity in recent research [Hau98]. The term *configurable computing* is taken to mean what it has traditionally meant: A method of computing that contains some way of changing the hardware in the system without physically modifying it. However the term *reconfigurable computing* is used to refer to a disjoint set of systems which contain devices that do not lose all state information when configured, hence, they can be *reconfigured*.

Traditional FPGAs are configurable since they expect to be given their complete hardware design information through a serial bitstream; To download a new hardware design the device must be reset. In contrast, the set of devices classified as *reconfigurable* do not need to be reset. The clock

to the device can be stopped, new design information can be given to the device to change part (or all) of the hardware, and the clock can then be resumed. Any part of the hardware not reconfigured continues exactly as it was before the reconfiguration. The key advantage to reconfigurable devices over configurable ones is that the state information is left untouched on the unchanged parts of the device.

Even more recent advances in FPGA technology have necessitated a need for a further sub–classification of reconfigurable devices. A *run–time reconfigurable* device is one that can be reconfigured while the unaffected parts of the hardware are not interrupted due to a clock stoppage or any another method of suspending the device for reconfiguration. The term *Run–time reconfigurable computing* has thus been created to refer to systems employing *run–time reconfigurable* devices.

Chapter 4 presents the design of a system that requires *reconfigurable* hardware for implementation. The platform in Chapter 3, however, contains only a *configurable* FPGA. Consequently, no testing is done that involves changing parts of the hardware while the system is operational.

2.1.2 Programmability

Unlike the various classifications of configurable hardware devices, the term "*programmable*" is more clearly defined by the literature. Programmability, in the context of computing, refers to the flexibility and control a programmer has over the behaviour of hardware by changing the software in the device[1]. A hardware design can be viewed as forming a continuum between *not programmable* and *programmable*:

1. *not programmable* – These devices contain no software, and thus cannot be controlled by software. The direction the hardware takes can only be influenced by changing the external inputs to the device. Custom ASICs and FPGA hardware designs that do not read any instruction stream fall into this category.

[1]There is one subtle exception to this definition. In the context of PLDs (Programmable Logic Devices), "programmable" refers to a configurable hardware device.

2. *programmable* – These devices use an instruction set architecture (ISA) to provide the programmer with a model of the hardware [DG97]. In the extreme sense, a programmable device gives the programmer control over every aspect of the hardware through the ISA. These devices read program instructions and act on them accordingly. Virtually all modern day processors are almost completely programmable. A programmer has control over most of the hardware but some components like the cache, prefetch, and branch prediction units operate independently of the software.

A configurable computing system may use hardware designs that are programmable to various degrees. A soft processor core[2], for example, is quite programmable, and a programmer can implement virtually any algorithm in it. In contrast, consider a hardware encryption algorithm that is implemented using a minimal processor and a small amount of software to direct the flow of the hardware (as opposed to a state machine to control the flow). The encryption system would be situated closer to the middle of the programmability continuum, since it consists mostly of the hardware implementation of encryption routines, which are independent of the software, but does contain a small programmable element. It can still be viewed as *programmable*, but not to the same degree as the soft processor core.

2.2 A Brief Introduction to Configurable Computing

The concept of using changeable hardware to expedite processing is not new. It was first proposed by Estrin in 1960 as a "fixed plus variable structure computer" [Est60]. His implementation (actually developed in [EBTB63]) consisted of a standard processor which would have control over an array of "reconfigurable" hardware. His belief was that when the "reconfigurable" hardware was setup to perform a specific task, thus removing the burden from the standard processor, the performance of the entire system could be enhanced.

[2]A soft processor core is a processor core which is downloaded to an FPGA. The Nios and FAST CPU are both examples of soft processor cores.

Modern configurable computing platforms have benefited from significant improvements in technology and design tools. However, the basic principle of configurable computing has remained unchanged. A configurable computing system incorporates some form of configurable (or reconfigurable) device that can be changed to perform various tasks, especially processor bound tasks [DW99]. Many vendors currently produce configurable computing systems for research, educational, and commercial applications [Guc00].

A configurable computing system for application specific acceleration can be viewed as something between a pure software implementation of an algorithm, and the custom hardware circuit of the same design. The system attempts to leverage the benefits of both hardware(fast) and software(small, easy to implement and debug) to create an overall "better" implementation. To accommodate multiple applications, the hardware portion of the design uses configurable logic to allow hardware changes as the demands of the applications change. The configurable hardware can be used to implement custom hardware designs for specific applications, or can be treated as a more general purpose resource and used to implement more generic designs[WK01], like soft processor cores. There are also several ways to attach configurable logic to the system that can alter the behaviour of the system, as discussed in Section 2.3.

2.3 Configurable Computing System Architectures

For any particular implementation of a configurable system (including *reconfigurable and run–time reconfigurable* systems), there are many possibilities for the system layout. Many systems have a general purpose "main" system processor, a configurable entity, and usually also contain memory and peripherals. It is the location of these components relative to the configurable entity that define and limit how the overall system behaves.

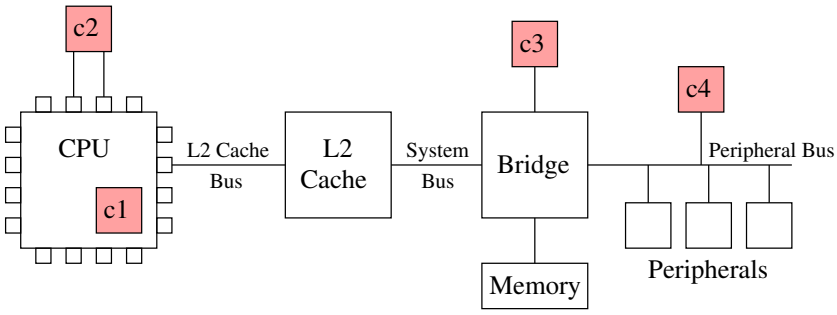

Figure 2.1: Coupling in Configurable System Architectures

2.3.1 Coupling

Coupling refers to the way in which a configurable entity is attached to the main processor and the system [CH02]. A tightly–coupled system has the configurable entity directly attached, or even inside the main processor, whereas a loosely–coupled system has the configurable entity far away from the main processor. Figure 2.1, adapted from [CH00] and [Bis03], demonstrates the varying degrees of coupling, which are shaded and labeled "c1" through "c4". As the configurable hardware moves away from the main processor, it generally becomes larger due to the reduced cost to implement it, but incurs a higher communication penalty for any communication with the processor.

c1. When the configurable hardware is within the main processor it is referred to as a tightly–coupled. The hardware is generally small due to the high cost of situating it within a processor. It is usually located on, or has access to, the main datapath of the processor, and is used to partially or completely implement custom opcodes. For example, it may be used to implement a new function for the Arithmetic and Logic Unit (ALU). There is a very low latency for communication allowing the main processor to be in frequent communication with the configurable hardware (several times per instruction).

c2. This configuration moves the configurable units outside the main processor, but leave them directly connected by dedicated pins. The configurable hardware is in direct communication

with the main processor but not under its direct supervision. This means that the hardware is effectively a coprocessor for the main processor and can be used, for example, to implement complete opcodes such as floating point arithmetic instructions or hardware multiply and divide. This particular configuration is referred to as *instruction level coupling* in [Bis03], meaning that while the hardware is independent, the results from the hardware are used as results from an instruction, or a series of instructions, in the main processor.

c3. In this configuration, the reconfigurable hardware behaves like an additional processor. It is either connected directly to the bridge (as shown in Figure 2.1), or can be directly connected to the high speed system bus. It does not have access to the L2 cache of the main processor. It becomes increasingly likely, as the system moves away from the tightly–coupled system, that there may be more than one configurable device present. This is referred to as *system bus level coupling* [Bis03]. There is a moderate penalty for communication, so the main system processor and the configurable hardware communicate infrequently. For example, each configurable entity may be active in rendering part of a scene, and only communicate with the main processor to exchange scene data or to report results.

c4. This is referred to as a *loosely–coupled system* or as *peripheral bus level coupling* [Bis03]. The configurable resources are stand–alone processing units (and there are often more than one of them), connected to the peripheral bus or even across a network. These units rarely communicate with the main processor, usually only to download a "task" to be computed, and then to give the result back to the main processor. An example in this case would be if each entity were rendering a complete scene. They would each download the complete scene data, and only report back when the final rendering was complete.

2.3.2 Memory Location

The location of memory in relation to the configurable device is also important. The *data memory location* and *instruction memory location* refer, respectively, to where the configurable device

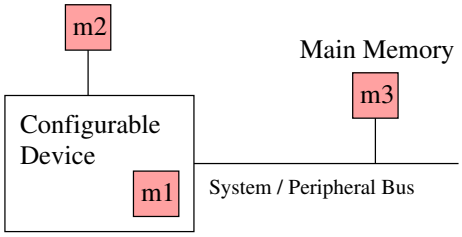

Figure 2.2: Memory Locations in Configurable System Architectures

fetches program data and instructions. Each of the coupling configurations in Section 2.3.1 may require the use of either instruction or data memory, or both. Just as there are several ways to connect a configurable resource to a system, there are several ways of connecting a memory to a configurable resource. Figure 2.2 shows the three locations of memory (numbered and shaded) in relation to a configurable device. A configurable system may use the same method for implementing instruction and data memory, or may use two different methods.

m1. The memory for the configurable device is located inside the device itself. Since the device is implemented in configurable logic, the memory is most likely part of the device design and is also implemented in configurable logic. In this case, the memory is quite small but fast. It uses on–chip resources, so it is size limited, but it can be connected directly to the hardware design within the configurable resource.

m2. The memory is external to the device, but the configurable device has exclusive access to it. The memory is most likely not implemented with configurable logic, and is still fairly small since it is not cost–effective to outfit each configurable entity with a large dedicated memory. However, it can be quite fast since the hardware can communicate directly with the memory without any bus contention issues.

m3. In this location, the configurable device uses the main memory, and thus shares memory with the main processor. Memory requests need to be routed across the system or peripheral bus and into the main memory. Bus collisions become an issue as the device no longer has

exclusive access to the memory (as it did in the first two cases). Multiple devices may now be making concurrent requests to the memory, so a method of synchronization and arbitration is needed on the bus shared between all the devices.

2.4 Systems for Application Specific Acceleration

Taking a step back from all the permutations of configurable systems possible by combining the various architecture layouts presented in Section 2.3, it is further possible to replace the configurable entity with a non–configurable entity. This begins to define coprocessing systems in the broader sense which are used for application specific acceleration. Such systems cover everything from ones where dedicated custom hardware provides acceleration to full symmetric–multiprocessing (SMP) [Int97] systems.

At one extreme of such systems, there is a dedicated custom ASIC providing a specific function targeted to a specific application. The ASIC is not configurable, so it can only provide assistance to an application which requires the function implemented. Moving slightly away from this extreme, the application specific hardware can be implemented in an FPGA, or another configurable resource. While not as fast as the custom ASIC solution, the hardware resource is reusable by other applications.

The other extreme also contains a non–configurable ASIC, in the form of a general purpose processor. An application targeted for acceleration would simply use both processors to achieve a higher execution rate. Again, moving away from the absolute extreme, it is also possible to implement a general purpose processor in an FPGA (a soft processor core).

It is the entire range involving configurable hardware that is of interest to the PADS Research Group at the University of Waterloo, Ontario, Canada. The topic of lightweight soft processor cores for application specific acceleration is what is studied in this research.

Chapter 3

Configurable Computing Platform

This chapter describes the hardware and software used in the design and implementation of a configurable computing system for this research in using lightweight processor cores for application specific acceleration. The hardware and software permit the development of a system that is representative of the configurable computing systems that are currently used by academia.

3.1 Nios Embedded Processor Development Board

The hardware used for the implementation of the system in this research is an Altera Nios Embedded Processor Development Board [Alt03c] depicted in Figure 3.1. The board contains an APEX 20K200EFC484-2X FPGA which is the only user configurable logic device on the board. This FPGA contains $8,320$ Logic Elements (LEs) which translates into approximately $526,000$ total usable gates. It also contains 52 Embedded System Blocks (ESBs) providing a total of $106,496$ usable bits of internal RAM.

Along with the APEX FPGA, the board also provides 256 kB of onboard SRAM for bulk data storage and processor instruction memory, a UART for downloading code and as a means to interact with the configured hardware and software, a JTAG connector to configure the FPGA with a new core, several buttons and LEDs (primarily used for debugging), and other features not used in this

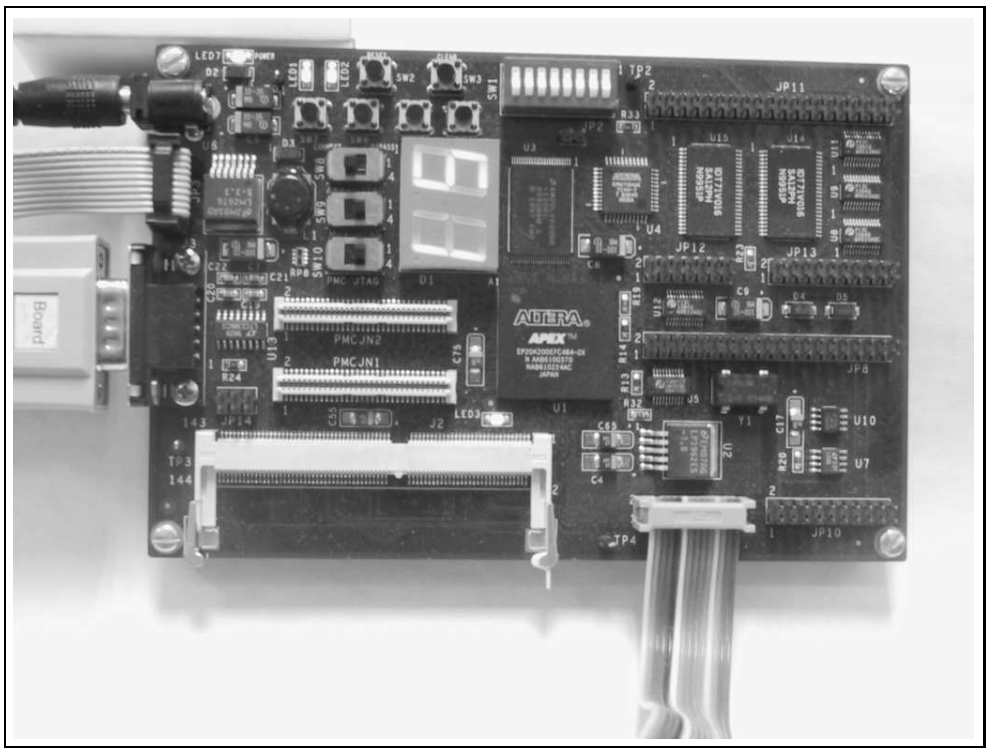

Figure 3.1: Nios Embedded Processor Development Board

research. A complete list of board features can be found in [Alt03c].

3.1.1 Nios Processor

While the Nios Embedded Processor Development Board can be programmed with any hardware design, it is geared towards use with the Altera Nios processor [Alt03b]. A Nios processor serves as the master (or main) processor in the embedded system for this research.

The Nios processor is a 33 MHz processor which has a 16–bit and 32–bit version. Both versions can be created with a *minimal*, *average* (default), or a *full* set of processor features, and can be optimized in synthesis for *speed* or *area*. For the purposes of this document, the "Nios32" processor refers to the 32–bit version of the Nios processor with the *average* set of processor features optimized for *area*. The *average* set of processor features was chosen because the minimal set eliminates the interrupt handling features of the Nios processor (which are required for peripherals to

give notifications to the Nios processor), and the *full* set provides extra hardware features (such as a hardware multiplier) that are not needed. The system is clocked at 33 MHz, and optimizing for *area* meets all timing requirements, so optimizing for *speed* is unnecessary. It should be noted that the feature set chosen and the optimizations used have little influence on the performance of any other system component, provided the entire system meets the 33 MHz clock frequency requirement.

3.1.2 Avalon Bus

At the core of a Nios system is the Avalon Bus [Alt03a], which interconnects all the processors, memories, and peripherals. The Avalon Bus really consists of many point–to–point links, one between each master and slave device. All slave devices, including main memory, appear in the global addressable memory space at offsets defined in the System On a Programmable Chip Builder (SOPC Builder, see Section 3.1.4).

An Avalon Bus master device can only make requests to slave devices. That is, it can read data from, and write data to, a slave device by simply reading/writing to offsets in the global memory space. Other master devices cannot initiate communication with a bus master unless the target also has a slave interface.

The Avalon Bus slave interface defines the number and width of registers within the slave device that are available to any bus master. No slave device may initiate communication with another slave or master device. The slave interface exists entirely to serve read and write requests from bus masters.

Mastership arbitration on the Avalon Bus follows a fixed priority scheme. The Avalon Bus arbitrator has a built–in priority for each master, which defaults to the order components are defined in the SOPC Builder. If multiple requests for mastership are active at the arbitrator, the one with the highest priority is allowed to proceed when the bus becomes free. There is no need for a FIFO or any other queuing system with this scheme, however it does mean there is a possibility of starving low priority bus masters.

3.1.3 Host PC

The Nios Embedded Processor Development Board is connected to a host computer (host PC). The computer uses the parallel port to control the JTAG programmer, and a serial port to communicate with the UART on the development board. The host computer also contains most of the tool flow (see Section 3.1.4) so it was used to design, build, and test the implementations of the system. The host computer has an Intel Pentium 4 2.60 GHz CPU with 1 GB of RAM, running Windows Server 2003.

A second computer running Linux, however, was used as part of the development platform, mainly to edit VHDL and C files and to provide convenient access to a revision control repository. The second computer was also used to create the assembler for the lightweight processor core designed for the research (the processor is called the FAST CPU, and is introduced in Section 4.2), and to assemble files for the FAST CPU. The Windows Server 2003 host computer accesses files on the Linux workstation through a network share directory.

3.1.4 Tool Flow

The development tool chain is shown in Figure 3.2, and includes both the required flow to produce a system containing a Nios32 processor, and the components added to the flow for the purposes of this research. Altera provides the Quartus II application suite to target hardware designs for the Nios Embedded Processor Development Board. Quartus II provides an end–to–end tool flow from VHDL and schematic editors, through a synthesizer and timing analyzer, and to a programmer to download a synthesized design to a development board through a JTAG interface. Part of the Quartus II suite is the SOPC Builder (System On a Programmable Chip Builder) [Alt03e]. The SOPC Builder tool is used to construct an Avalon Bus [Alt03a] based system on a chip, by connecting processors, memory, and other peripherals on a configurable number of buses. The output of the SOPC Builder is a series of VHDL files which implement the designed system.

Quartus II synthesizes the entire system using user written VHDL files, schematic design files,

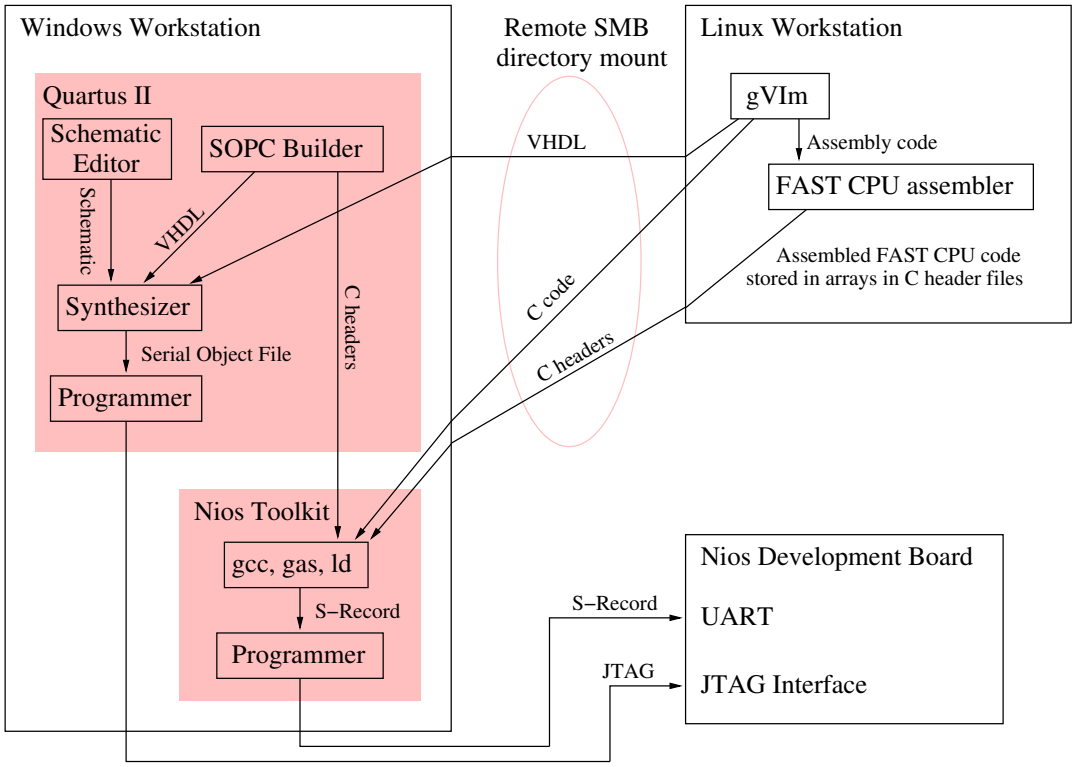

Figure 3.2: Development Tool Chain

and VHDL files produced by the SOPC Builder. When successfully synthesized, the output is a SRAM Object File (`.sof` file) which can be downloaded to the board through the Quartus II JTAG programmer.

The SOPC Builder also generates a series of header files that describe the existence and addresses of the hardware/peripherals in the synthesized system. Code written to execute on a Nios processor uses these header files to access the hardware so that memory offsets do not need to be hard–coded in the program. Application code for the Nios processor is built using the GNUPro toolkit [Cyg99]. The GNUPro toolkit uses a gcc cross–compiler and a Nios assembler (and linker) to build an S-Record suitable for execution on a Nios processor. The S-Record is downloaded to the Nios Embedded Processor Development Board through a UART connection.

The Nios32 processor code used in this research also includes header files generated by the lightweight processor (FAST CPU) assembler, which contain arrays of code built specifically for the FAST CPUs. The code running on the Nios32 processor instructs a FAST CPU core to download the code in one of these arrays when the FAST CPU functionality needs to be changed.

Chapter 4

System Design

4.1 System Model Used

Using the system platform described in Chapter 3, it is not possible to implement all the combinations of system architectures presented in Sections 2.3.1 and 2.3.2. The tools for the Nios Embedded Processor Development Board generate a system in which all components are connected to a single Avalon Bus. Without designing custom implementations of all the system components, which is beyond the scope of the thesis, this behaviour cannot be changed. Figure 4.1 shows a modified version of Figure 2.1 representing the types of coupling implementable with a Nios system. The changes are as follows:

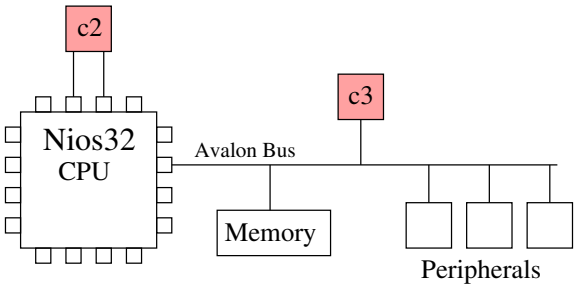

Figure 4.1: Coupling in a Nios Embedded System

- Configurable entity number "c1" is removed – The Nios32 processor is a standard component from Altera, and is not generally modifiable. Adding the facilities to support an internal configurable logic component would require an in–depth knowledge of the Nios processor. This is beyond the scope of the thesis.

- The "L2 Cache" component is removed – The Nios32 processor used in this research does not support a cache. A cache, specifically an instruction cache, would decrease the bus demands of the Nios processor by reducing the number of accesses to the main memory. The presence of a cache would significantly impact any bus–bound activities in the system, but would not change the computational performance of each system component with respect to the others.

- The "Bridge" component is removed – All components and peripherals in many Nios systems are connected directly to a single Avalon Bus. A bridge is normally used to switch data between buses of differing bandwidth. Although it is possible to implement a multi–bus system on a Nios Embedded Processor Development Board, investigating this type of system is beyond the scope of the thesis.

- Configurable entities "c3" and "c4" are combined. The system and peripheral buses in Figure 2.1 are really the same physical bus in a Nios system, so configurations "c3" and "c4" result in the same implementation.

Coupling configuration "c2" can be implemented in a Nios processor by using Nios custom instructions [Alt02a]. The Nios custom instruction interface automatically modifies the hardware description of a Nios processor to add the necessary external dedicated pins[1] for connection with external logic. However, this does not scale well to a multiple processor implementation (which is one of the goals of the research). There is a limit of 5 custom instructions in a Nios processor, limiting the number of directly connected processors to that number. A single instruction could be used for all processors to overcome that limit, however the Nios custom instructions only support

[1]These are really *virtual pins* since the entire system is being synthesized within a single FPGA.

one argument (perhaps the target processor number), meaning any other data would need to be read from an external connection to memory. This creates the need for extra logic within the custom instruction to route the data to the appropriate processor and necessitates an Avalon Bus connection since all data cannot be passed through the instruction. Coupling configuration "c3" already contains an Avalon Bus interface, so the extra logic required in configuration "c2" would only add to the size of the design, without adding any features that could not be realized by using the Avalon Bus directly (i.e., for all communication).

Configuration "c2" is not implementable on a system with a general purpose processor that cannot be resynthesized to support the necessary custom instructions or external interfaces. Previous research into using custom hardware designs for application specific acceleration has used configuration "c3" [Bis03], so for these reasons, coupling configuration "c3" is used in this research.

Figure 2.2 shows the three possible memory locations within the Nios Embedded Processor Development Board for a piece of configurable logic. All three locations are possible with coupling configuration "c3". The choice of memory location for the data (chosen to be location "m3") and the instructions (chosen to be "m1") is justified as the system design is described in Section 4.2.

Figure 4.2 shows the resulting system layout which is defined in the SOPC Builder. The system consists of a Nios32 processor, an interface to the 256 kB onboard SRAM, a UART for communication with the host PC, and a several FAST CPUs. All components communicate across a single Avalon Bus. The shaded area of Figure 4.2 is a standard Nios system, described in the Nios Software Development Tutorial [Alt03d].

4.2 The FAST CPU

The Flexible Application Specific Tiny CPU (FAST CPU) was designed to be small and fast, yet allow for the easy insertion of opcodes to test the tradeoffs of performing tasks in hardware or software. The FAST CPU model was written entirely in VHDL, as are the hardware modules for it.

The idea behind this modular approach, is that the FAST CPU offers a limited set of op-

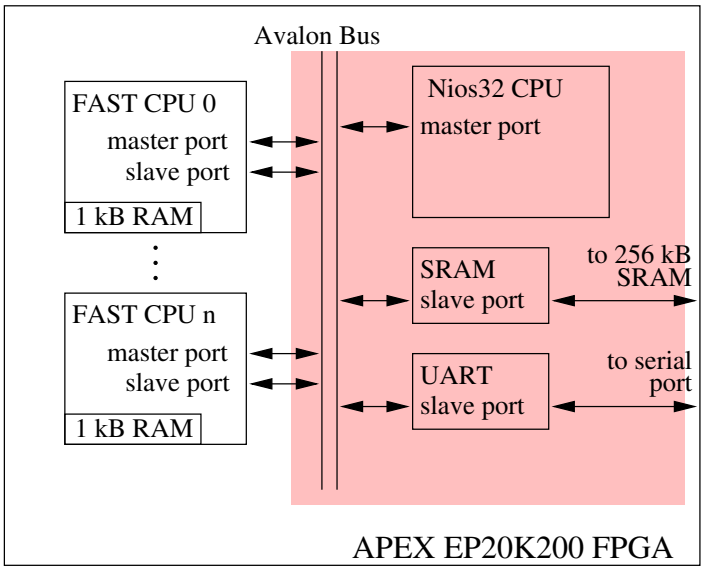

Figure 4.2: Nios32 System with FAST CPUs

codes which are sufficient to implement most algorithms. If further hardware acceleration is desired/required however, hardware modules can be inserted in the FAST CPU to add opcodes to the instruction set. Essentially, this ability to add instructions provides a means to accelerate part, or all, of an algorithm in hardware. The process of "inserting a hardware module into the FAST CPU" involves synthesizing the VHDL to create a new core, which then can be downloaded into an FPGA while the system is running. Since the Nios Embedded Processor Development Board has only a single FPGA which is not *reconfigurable*, and which must implement the entire system, the possibility of testing dynamically changing the FAST CPU cores has been deferred for future work (see Section 6.3). A system which could support changing the FAST CPU cores would require a board which is either *run–time reconfigurable*, such as a Xilinx Virtex-II Pro development board[Xil04a], or a development board that has multiple FPGAs.

The major features of the FAST CPU evolved over the design phase of the CPU, and are chosen to meet the needs of the applications examined. The features are as follows:

- **32–bit processor** – The FAST CPU uses a 32–bit design to easily interface with a Nios32

processor on the Avalon Bus so that longwords can be transferred in single bus transactions. The ability to natively handle 32–bit operations also reduces the code size, complexity, and processing time compared to performing them on a 16–bit processor. In application specific acceleration, complex algorithms that involve large (32–bit) integers are excellent candidates for moving to hardware cores (or lightweight processor cores in this research) to improve processing speed.

- **16 General Purpose Registers** – Each register is 32–bits wide, and no register contains any special definition or meaning.

- **1 kB Internal RAM** – Contention on the memory bus was a concern for the FAST CPU if the main Nios processor and all FAST CPUs were constantly fetching program instructions from the main memory. To avoid any potential bottleneck here, the FAST CPU contains a small internal RAM which is the exclusive source of instructions for the FAST CPU. The RAM is longword addressable only, since each FAST CPU opcode is exactly 32–bits wide (see Figure 4.3) so a word or byte addressable instruction memory is unnecessary. This RAM may also be used for data storage if the 16 registers are insufficient, for example, in the storing of a small array of data. The RAM and the registers are implemented using the same technology in the FPGA, so they have the same access time.

- **4 addressing modes** – Although register to register operations are expected to be the most common, the FAST CPU was also designed to support several more complicated addressing modes. The source argument can be any of the four addressing modes, and the destination argument, if required by the opcode, is always a register. Figure 4.3 shows that there are 2 dedicated bits for specifying the addressing mode of the source argument in the opcode. The possible value of these two bits are:

  0. **Register to Register** – The source operand is a register, as is the destination operand.
  1. **Immediate to Register** – The source operand is a 16–bit immediate value encoded in

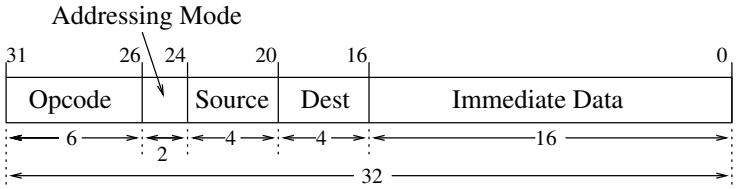

Figure 4.3: The FAST CPU Opcode Format

the opcode, sign extended to 32–bits. The destination is a register.

2. **Immediate–Indirect to Register** – The source operand is in the FAST CPU internal RAM location specified by the 16–bit value encoded in the opcode (only the lower 10–bits are used however to address the 1 kB internal RAM). The destination is a register.
3. **Register–Indirect to Register** – The source operand is in the internal memory location specified by the source register. The destination is a register.

One opcode reverses the definition of the source and destination operands so that values can be written to the internal RAM. The store opcode (`ST`) stores the value located in the destination register in either the source register, or an internal memory location specified by the source operand.

- **24 opcodes** – The FAST CPU contains a limited set of opcodes to keep the size of the processor small, and because an application specific processor that only runs a single program has no need for many instructions which would be found in a general purpose processor. Each opcode has an identical format to facilitate easily decoding and executing the opcode. Figure 4.3 shows the opcode format, and Table 4.1 describes the basic set of FAST CPU opcodes.

A processing unit dedicated entirely to the acceleration of a particular portion of a specific application does not need to be a full–featured general purpose processor. Such a processing unit only needs to execute a single program specifically designed to compute results used by the application running on the main system processor. The logic area required by the FAST CPU can be reduced by

Table 4.1: The FAST CPU Opcodes

Opcode	Mnemonic	Description
0x20	ADD src, dst	$src + dst \rightarrow dst$ – Unsigned addition
0x22	AND src, dst	src AND $dst \rightarrow dst$ – Bitwise AND
0x09	BEQ src	Jump to src if $src == dst$ in the last CMP
0x0B	BGE src	Jump to src if $src \geq dst$ in the last CMP
0x0A	BGT src	Jump to src if $src > dst$ in the last CMP
0x0D	BLE src	Jump to src if $src \leq dst$ in the last CMP
0x0C	BLT src	Jump to src if $src < dst$ in the last CMP
0x08	BNE src	Jump to src if $src \neq dst$ in the last CMP
0x25	CMP src, dst	Perform $src - dst$ and set branch flags according to the result
0x05	IRQ src	Set the interrupt status to the least significant big of src. A value of 1 means the FAST CPU is generating an interrupt for the Nios32 processor.
0x00	JMP src	Jump unconditionally to the address in src
0x28	LD src	$src \rightarrow dst$ – Load the destination register with the value in the source
0x01	NOP	No operation
0x23	OR src, dst	src OR $dst \rightarrow dst$ – Bitwise OR
0x26	SHL src, dst	$dst << src \rightarrow dst$ – Shift dst left by the number of bits specified in src. Fills the least significant bits shifted in from the right with zero
0x27	SHR src, dst	$dst >> src \rightarrow dst$ – Shift dst right by the number of bits specified in src. Fills any bits shifted in from the left with the value of the most significant bit, so the result remains sign extended
0x12	SLC src	Clear the Avalon Bus slave read buffer address src
0x11	SLR src, dst	Read the Avalon Bus slave read buffer address src, storing the value in register dst
0x10	SLW src, dst	Write the value in register dst to the Avalon Bus slave write buffer address src
0x29	SPC src	$PC + 2 \rightarrow src$ – Write the program counter value, plus 2, into the register or memory address src
0x04	SSEG src	Place the lower byte of src on the seven segment display
0x02	ST src, dst	Store the value in register dst into the address or register specified in src
0x21	SUB src, dst	$src - dst \rightarrow dst$ – Unsigned subtraction
0x24	XOR src, dst	src XOR $dst \rightarrow dst$ – Bitwise XOR

purposely omitting several features from the design which are found in general purpose processors:

- Exception support (interrupt handling) – The FAST CPU does not contain any support for externally or internally generated interrupts. A program executing on the FAST CPU cannot be interrupted by any means other than a reset, which is done by writing a value of `0xFFFFFFFF` to the first offset of the FAST CPU slave interface, causing the processor to restore the bootloader. The FAST CPU does, however, contain an opcode (`IRQ`) for generating an interrupt on the main system processor.

- Stack – A stack was deemed largely unnecessary due to the limited resources in the FAST CPU environment and the fact that the FAST CPU is not designed to run multiple applications concurrently, which would require memory separation between the applications. For calling functions, the `SPC` opcode saves the current `PC` into a register. The end of a function can `JMP` to the contents of this register to effectively execute a return–from–subroutine instruction.

- Register Windows – For the same reasons that a stack was determined to be unnecessary, register windows are unnecessary. All the internal resources of the FAST CPU should be simultaneously available to a program, not only a subset of them.

4.2.1 Avalon Bus Slave Interface

All FAST CPUs must contain an Avalon Bus slave interface to provide a common way to communicate with the Nios32 processor. The slave interface memory–maps 4 longwords (a longword is 4 bytes) of the addressable system memory to each FAST CPU. The slave interface implementation is split into a read buffer and a write buffer, as shown in Figure 4.4.

To a bus master, the interface appears to consist of only 4 registers, represented in Figure 4.4 by the 4 hatched squares on the FAST CPU–slave port boundary. The registers on the boundary are not for storage; they are virtual registers. The actual storage is implemented by 4 registers in the read buffer and 4 registers in the write buffer. When a bus master writes to an address which maps to the

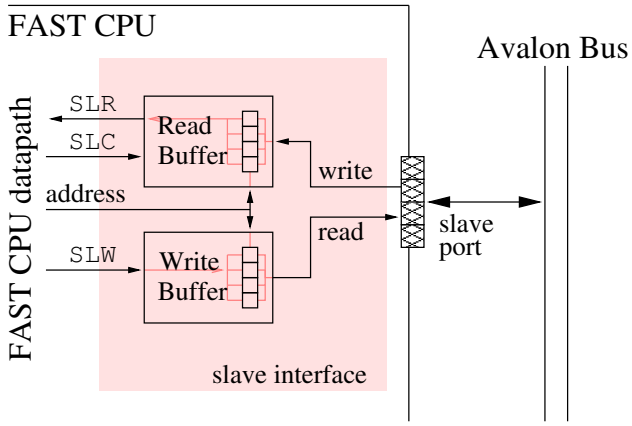

Figure 4.4: The FAST CPU Avalon Bus Slave Interface

FAST CPU, the value written is placed at the appropriate offset of the read buffer. The FAST CPU can read these values by executing a `SLR` instruction, to read from the bus read buffer. Similarly, when a bus master reads from an address mapped to the FAST CPU, the value is taken out of the write buffer. The FAST CPU can write to the slave write buffer by using the `SLW` opcode. A value written to the FAST CPU at an offset will not be the same value read back from the same offset.

This buffered approach was chosen because the FAST CPU does not support interrupt handling. As a result, the executing program cannot be interrupted when a bus master performs a read or a write on an address that is mapped to a FAST CPU. This means that, unlike most hardware peripherals, the FAST CPU cannot take immediate action when a particular address is written or read. The synchronization between the main system processor and the FAST CPU has been moved to software, and is done by probing the values in the read buffer and taking action based only on those values. The FAST CPU cannot tell if a bus master had read something from a write buffer, so the bus masters must inform the FAST CPU if it requires such a notification. The FAST CPU uses a third bus slave opcode, `SLC`, to clear the value at a particular address of the read buffer to detect when the same (or a different) value is written to an address by a master (other than a value of `0`). If the detection of multiple writes of `0` are required, then a hardware module could be used to

Table 4.2: The FAST CPU Bus Mastership Opcodes

Opcode	Mnemonic	Description
0x14	MAR src, dst	Master the Avalon Bus, and read the value at address src into register dst
0x15	MAW src, dst	Master the Avalon Bus, and write the value in register dst to the address in src

introduce a new SLC instruction which clears the specified address to a value other than 0, maybe 0xFFFFFFFF.

4.2.2 Avalon Bus Master Interface

The first, and only, additional hardware module designed for the FAST CPU provides Avalon Bus mastership. Table 4.2 shows the additional two opcodes which the mastership module adds to the FAST CPU.

The FAST CPU has a very limited internal RAM for all the program code and for limited data storage. Bulk data storage cannot be accomplished within the FAST CPU, and must be offloaded to the main memory. The FAST CPU Avalon Bus master interface was designed for this purpose. It is the responsibility of the program on the main system processor, and the code on the FAST CPU to negotiate the location of the data in main memory. There is no pre–defined dedicated memory region in main memory for the FAST CPUs to use.

4.2.3 Boot Loader

To facilitate the process of downloading code to the FAST CPU, the CPU is configured with one of two boot loaders during synthesis. If the FAST CPU contains the bus master module, then a bus mastering boot loader is used; if not, an alternative slave–only boot loader is used. Immediately after a processor reset, the boot loader becomes active, and waits to receive data from the main processor.

In the bus mastering boot loader, the main system processor (the Nios32) gives the FAST CPU

the location and size of a program in main memory. Once it has obtained this information, the FAST CPU copies the program out of main memory and into its 1 kB internal RAM. When complete, it signals the Nios32 processor with an interrupt. After the Nios32 acknowledges the interrupt, the FAST CPU jumps to the first instruction of the program.

With the slave–only boot loader, the Nios32 processor enters a loop writing the offset and data at that offset to the FAST CPU. The FAST CPU polls the slave interface register which stores the offset and when it changes, the FAST CPU reads and stores the associated piece of data at that offset. The FAST CPU then sends an interrupt to the Nios32 processor to indicate it is ready for the next piece of data. The download is complete when the offset is set to `0xFFFFFFFF` by the Nios32 processor, which is way beyond the size of the internal FAST CPU RAM. At this point, the FAST CPU jumps to the first instruction of the program.

4.3 FAST CPU Application Programming Interface

A programming model where programs communicate directly with the FAST CPUs at their individual memory offsets would not abstract well into a device driver and user library pair. Such an abstraction would be required if an actual operating system were to be used on this system. It is also not easy to develop or maintain user programs that use direct hardware communication. An API was developed to hide the hardware communication and to present the user with a friendly interface on which FAST CPU programs can be based. The code for the API is included in Appendix C.1.

The functions which implement the API communicate directly with the hardware. Normally they would use a series of memory mapped requests or I/O Control (IOCTL) calls to communicate with a kernel driver, and the kernel handles the direct hardware communication. The lack of an operating system, however, prevents such a design. The functions in the FAST CPU API are as follows:

```
int fastcpu_init(unsigned long base, unsigned long irq);
```

Probes for FAST CPUs beginning at the specified `base`, and initializes structures for all FAST CPUs found in the system. The `irq` is the interrupt number that will be generated if the FAST CPU executes the `IRQ` opcode, and is incremented as each FAST CPU is found.

```
struct _fastcpu *fastcpu_alloc(void);
```

Allocates a FAST CPU. Returns the FAST CPU structure to the caller. The caller must use the returned structure pointer in all further communication with the FAST CPU subsystem.

```
void fastcpu_program(struct _fastcpu *cpu,
     unsigned long *program, unsigned long len);
```

Loads the specified program code into the FAST CPU.

```
void fastcpu_free(struct _fastcpu *cpu);
```

Returns the FAST CPU given by `cpu` to the pool. It is automatically reset when this function is called.

```
void fastcpu_reset(struct _fastcpu *cpu);
```

Forces the FAST CPU given by `cpu` to reset. After a reset `fastcpu_program` can be called to download a program into it.

```
int fastcpu_num_cpus(void);
```

Returns the number of FAST CPUs that the probe function was able to find.

Using the API, the FAST CPUs appear to the programmer as an allocatable resource. To be used they must first be allocated, then programmed. If no FAST CPUs are available, the programmer has a choice of either waiting, or invoking an implementation of the program on the main processor. Once programmed, the interface to the FAST CPU takes on whatever format the FAST CPU program implements, with the exception that the FAST CPU can be reset at any time to return it to a known good state, and restore the boot loader. When the Nios program is finished using a

FAST CPU, it must be freed. Figure 4.5 is minimal C code for communication with a FAST CPU application, it illustrates how a programmer can use the API to gain access to the FAST CPUs.

All the programs in this research that use the FAST CPU are based on the code in Figure 4.5. As shown in System Testing (Chapter 5), this code may be a useful resource in deciphering what each test is attempting to do.

4.4 FAST CPU Integration

Since the FAST CPU is designed to appear as an Avalon Bus peripheral, the process of attaching a FAST CPU to a Nios32 system on the Nios Embedded Processor Development Board requires little effort. The bus slave and master interfaces are created using the SOPC Builder, which then automatically generates all necessary Avalon Bus VHDL code, signal exports, and header files for the interfaces. Figure 4.2 shows a graphical view of the SOPC Builder system definition of the Nios32 system with n–FAST CPUs attached to it. Unfortunately, the FPGA on the Development Board is relatively small by today's (2004) standards, so there are only sufficient resources to synthesize a system with 3 FAST CPUs. Consequently, all tests are done using a maximum of 3 CPUs, and larger systems are left for future work (see Section 6.3). It is known, though, that it is possible to synthesize a 10–FAST CPU system using a board containing the smallest available Stratix–II (the EP2S15 [Alt04]) FPGA.

Table 4.3 shows the resource requirements of the FAST CPUs before integration with a Nios system. The first row is a FAST CPU without the bus master module (so it is using the slave–only boot loader), and the FAST CPU in the second row is using the bus master module (so it is using the bus mastering boot loader). The resources used by a Nios processor (again, outside a system) are also included for comparison, and it can be seen that the FAST CPU is indeed quite small compared to a general purpose Nios processor. The "Time to Program" information represents the time to download each longword of a program into the FAST CPU. The slave–only boot loader is significantly slower since it goes through the process of an interrupt for each longword downloaded, whereas

```
/* FAST CPU example program */
#include <fastcpu.h>

/* include the output of the FAST CPU assembler */
#include "program_code.h"

void main(void)
{
    struct _fastcpu *cpu[3];
    int data;
    char done = 0;

    /* Probe for FAST CPUs starting at offset 0x1000 with irq 20 */
    fastcpu_init(0x1000, 20);

    /* Allocate a CPU */
    cpu[0] = fastcpu_alloc();

    /* If that fails, we may want to use a software implementation of
     * whatever we're trying to ask the FAST CPU to do */
    if(cpu[0] == NULL) {
        /* do something in software, or just wait, keep calling
         * fastcpu_alloc() to get a FAST CPU */
    }

    /* Program the CPU */
    fastcpu_program(cpu[0], program_code, program_size);

    while(!done) {
        /* Send something to the FAST CPU */
        *(cpu[0]->loc[0]) = 0x97071435;

        /* Wait for the interrupt from the CPU */
        while(!cpu[0]->done) ;

        /* Read something back */
        data = *(cpu[0]->loc[0]);

        /* Exit under some condition */
        if(data == 42) done = 1;
    }

    /* Free the CPU */
    fastcpu_free(cpu[0]);
}
```

Figure 4.5: Template for a Program using a FAST CPU

Table 4.3: Resource Requirements for an Individual FAST CPU

Processor	**Area (LEs)**	f_{max} **(MHz)**	**Time to Program** ($\frac{\textbf{cycles}}{\textbf{longword}}$)	$t_{overhead}$ **(cycles)**
FAST CPU	716	59.40	391	383
FAST CPU with bus master module	894	59.40	7	383
Nios32 Processor	≈ 1300	≈ 62		

the bus mastering boot loader reads the entire program directly out of the memory. The $t_{overhead}$ column is the command overhead, that is, the number of cycles required to write a command to the slave port of the FAST CPU and process the interrupt that the FAST CPU responds with when the command has finished. The overhead is measured by invoking a command that does nothing on the FAST CPU, and is used to confirm the experimental results in testing and verification.

Many of the FAST CPU tests in Chapter 5 require the bus master module to be installed. To avoid synthesizing different systems for different tests, all the testing and verification is done using the same system in which all the FAST CPUs have the bus mastership module.

Chapter 5

System Testing

This chapter presents the testing and validation done with the FAST CPU system from Chapter 4. A testing module is first presented as a means of facilitating system testing. The tests are divided into Avalon Bus (I/O) tests (Section 5.2), computation tests (Section 5.3), and a minheap application test (Section 5.4). Conclusions are drawn from each of these tests with respect to the expected behaviour of the test, and how the results compare to the Nios32 processor.

The flow of each test is illustrated in Figure 5.1. The Nios32 processor first instructs each FAST CPU in the test to download the program code related to the test. Once all downloads are complete, the testing module is activated, and the Nios32 processor sends a series of commands to the FAST CPUs. Each FAST CPU responds with an interrupt when each command is complete. When the entire test is complete, the testing module is deactivated and the results are read. Then either another test is started, or the testing process is complete and the results are recorded.

5.1 The Testing Module

The testing module measures the number of Avalon Bus (I/O) cycles required by various components in the system, and also counts the total number of clock cycles required to complete the test. By strategically counting the clock cycles used by various activities, instead of only measuring the

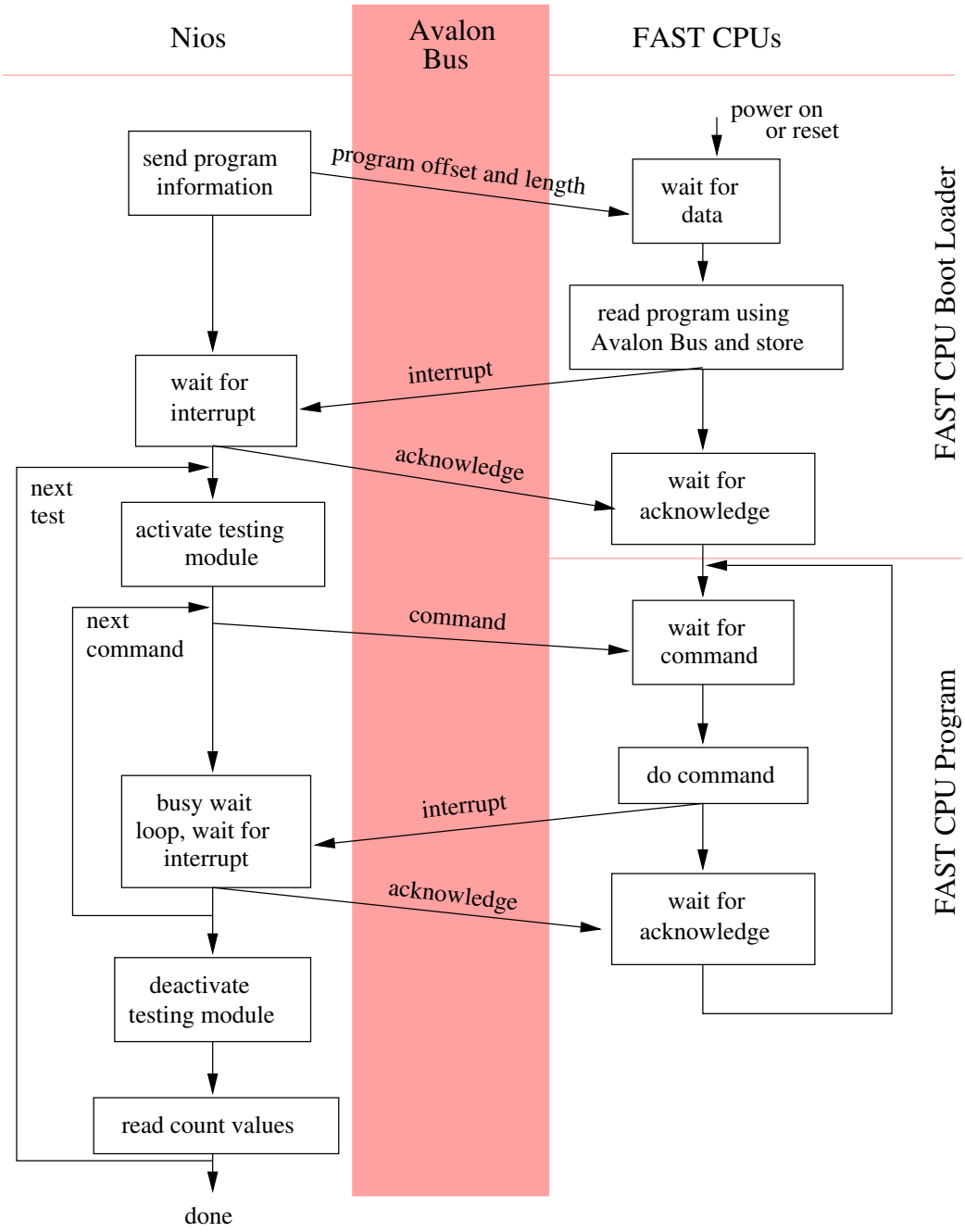

Figure 5.1: Nios and FAST CPU Test Program Flow

elapsed time (or total clock cycles) for a complete test, the cycles can be categorized into processing (CPU) cycles, and bus (I/O) cycles. A greater understanding of the interaction of the FAST CPUs with a Nios32 processor and the Avalon Bus can be developed with this categorization.

When the Nios32 processor gives a task to a FAST CPU, it enters a busy–wait loop, and waits for an interrupt from the FAST CPU to signal the task is complete. During this loop the Nios32 processor constantly fetches instructions from the main memory, making it challenging to measure the individual the sources of memory utilization in the system. Memory accesses can come from either FAST CPU data memory accesses, or Nios32 instruction fetching (in the busy–wait loop).

Ideally, probes could be inserted at strategic points in the system to directly count the bus accesses from each source, however, without modification of the system VHDL generated by the SOPC Builder this is not possible. The shaded area in Figure 5.2 shows the part of the system containing generated (and thus, not easily modifiable) code. The entire Avalon Bus and Nios32 processor fall withing this region.

If it were possible to collect these metrics with a single probe on the SRAM, it would be done by subtracting results of a measurement when the Nios32 processor is active during FAST CPU activity, and one where it is not. However, this approach is not possible, since the Nios32 processor cannot be halted or otherwise prevented from fetching instructions from main memory during the busy–wait loop. The Nios32 lacks an instruction that can halt or stop the processor until an interrupt occurs, so it simply cannot be “stopped”. Attempts at pointing the Nios32 to a small internal ROM containing the busy–wait code, to take it off the Avalon Bus while waiting for the FAST CPUs to respond, was equally not possible as the Nios32 processor automatically resets whenever the program counter is set to any address outside the main memory.

The solution, was to create a testing peripheral with several probes to count cycles at various points in the system. The module appears on the Avalon Bus as a peripheral so the Nios32 processor can start, stop, and read the values out of the module as desired. Figure 5.2 shows the testing module added to the original system of Figure 4.2. The probe are attached only to “external” signals to

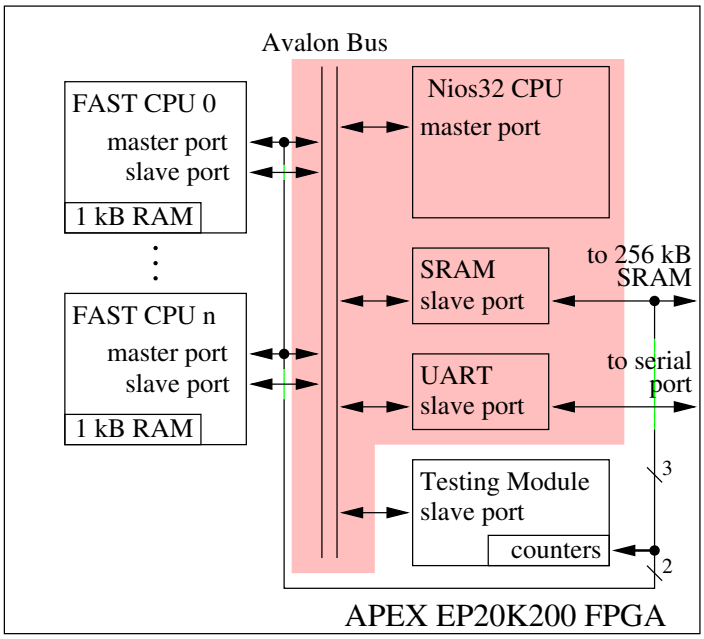

Figure 5.2: System with Testing Module Inserted

eliminate the need to alter the existing VHDL[1] for the Avalon Bus, the Nios32 processor, or any peripherals in the system.

The module consists of nine counters. One counter is connected to the clock signal so it always increments when the module is active. The remaining eight are triggered by external probes and only increment on a clock edge when the probe signal is low ('0'). The first seven probes trigger a count on the rising edge of the clock, and the 8^{th} probe count increments on a falling clock edge. The eight probes are connected as follows, and all test results are derived from these probe locations:

p_1. **FAST CPU 1 Avalon Bus Read Request** – This signal is low ('0') when FAST CPU 1 is either requesting or performing a read on the Avalon Bus.

p_2. **FAST CPU 2 Avalon Bus Read Request** – Same as p_1, but for FAST CPU 2.

p_3. **FAST CPU 3 Avalon Bus Read Request** – Same as p_1, but for FAST CPU 3.

[1]The existing VHDL is machine generated and is optimized. The VHDL is re–generated, resulting in the loss of any manual changes, when the system is modified. In addition, optimized machine generated VHDL is not easy to grok.

p_4. **FAST CPU Any Read Request** – This signal is low when any FAST CPU is waiting to perform a read request. It is the result of $(p_1$ AND p_2 AND $p_3)$. The purpose of this probe was originally to compute the number of bus collisions in the system. The value of the respective counters from $(p_4 - (p_1 + p_2 + p_3))$ was thought to be the number of bus collisions. Unfortunately, this does not take into account the number of stalls due to the Nios32 processor, which can artificially inflate the count of this probe. For example, if all FAST CPUs are reading, or attempting to read, on the Avalon Bus it cannot be determined if one is succeeding, or if they are all stalling while the Nios32 processor is performing a read. Consequently, it is quite possible for the value of this probe to exceed the total number of SRAM read cycles observed with probe p_7.

p_5. **FAST CPU All Read Request** – This signal is the result of $(p_1$ OR p_2 OR $p_3)$, so it is low when all FAST CPUs are either performing, or waiting to perform, a read request. The purpose of this signal is only to determine *if* collisions are happening between FAST CPUs. It does not add any other beneficial data to the analysis.

p_6. **SRAM Chip–Select** – This signal is low when a master on the Avalon Bus is performing a read or write operation to the SRAM. This signal should be the sum of the counters triggered by p_7 and p_8.

p_7. **SRAM Read** – This signal is low when a bus master is reading from the SRAM.

p_8. **SRAM Write** – This signal is low when a bus master is writing to the SRAM. The write signal to the SRAM is triggered on a falling clock edge, so the counter triggered by p_8 increments when p_8 is low on the falling edge of the clock. The SRAM write signal is always low on a rising clock edge.

The testing module uses an Avalon Bus slave interface similar to the one used the FAST CPUs to memory map internal registers to the total addressable memory space. Table 5.1 shows the memory

Table 5.1: Testing Module Memory Map

Address Offset	Mode	Width	Description
0x00	read	32	Read the clock cycle counter
0x04 → 0x20	read	32	Read Probe $p1 \rightarrow p8$
any even	write	32	Reset all counters and activate counters
any odd	write	32	Stop all counters

map for the testing module. To start a test, the Nios32 processor writes to any even address in the testing module, which causes all the counters to reset, and the testing module to become active. The system can then conduct whatever test is desired, and stop the testing by writing to any odd address in the testing module to stop the counters. After the counters are stopped, the results can be read from the various probes to get the cycle counts. The testing module can be started and stopped in 33 clock cycles (quantified by activating, then immediately deactivating the module, then reading the clock counter value).

5.2 Avalon Bus Utilization Testing

The Avalon Bus consists of many point–to–point links, one between each master and slave device. In the configurable system under test, all the FAST CPUs and the master Nios32 CPU access the main memory. The Nios32 fetches both instructions and data from the SRAM, and the FAST CPUs fetch only data from main memory. This convergence on main memory (the SRAM) may cause the system to bottleneck as the memory attempts to serve all the masters making requests.

The goal of this test is to find a relationship between the bus utilization demands of each FAST CPU processor, and the total number of cycles required to complete the task at hand. In the test, each FAST CPU runs a program that demands the Avalon Bus for a known percentage of the total cycles if there are no bus collisions[2] (the bus utilization is the *independent* variable in this test).

[2]It is computed, anyway, for the ideal case (no collisions). The actual individual FAST CPU bus utilization is difficult to measure since stalls are indistinguishable from active transfers. A probe would be required directly on the Avalon Bus to make this distinction (see Section 5.1)

The clock cycles (the *dependent* variable) required to perform the complete test concurrently on one, two, and three FAST CPU processors are measured for different bus utilization demands, then compared and analyzed for evidence of bus collisions.

5.2.1 Test Procedure

Figure 5.3 shows code for the FAST CPU which loops a desired number of iterations (stored in `R2`) to keep the Avalon Bus busy. The number of cycles required to execute each instruction on the FAST CPU is also shown. Ideally 3 of the 8 cycles required for each `MAR` opcode will use the Avalon Bus, however if there is a collision on the bus then the `MAR` may use more while it is forced to wait for the bus, so it is designated "$3^{\dagger}$". The extra 8 clock cycles required to exit the loop when R2 becomes 0 and the cycles required to setup R2 (not shown in Figure 5.3) are not included in the total clock cycle computation in Figure 5.3. Instead, these times are folded into a parameter called $t_{overhead,n}$ which is explained in Section 5.2.2.

A test on a single FAST CPU consists of a pair of parameters, m and i. m is the number of `MAR` instructions the FAST CPU should execute in sequence for each bus–busy loop iteration it performs, and has a range of $1 \leq m \leq 100$. i is the number of loop iterations to perform and ranges from $0 \leq i \leq 1000$. The lower bound of m is 1 because measurements using $m = 0$, where there are no bus transactions in the bus–busy loop, do not add anything beneficial to a bus analysis. The lower bound of $i = 0$ is used to measure the $t_{overhead,n}$ parameter.

The testing module from Section 5.1 has a probe for each FAST CPU to count the total number of Avalon Bus read accesses. This quantity is the sum of the $3^{\dagger}m$ values for all i iterations of the loop in Figure 5.3. It is possible to do the analysis using $3^{\dagger}m$, so the value of $3^{\dagger}$ does not need to be measured for each `MAR` instruction in each loop iteration.

The test follows the testing flow diagram in Figure 5.1. The Nios32 processor begins by dynamically rewriting the piece of FAST CPU code in Figure 5.3 to insert the desired number (m) of `MAR` opcodes into the bus test loop. It then instructs the FAST CPUs to download this code, and

	Clock Cycles
`lp:`	
`CMP 0, R2`	4
`BEQ lpdone`	4
`MAR R0, R1`	$5 + 3^{\dagger}$
Additional $(m-1)$ *MAR instructions inserted here each requiring* $5 + 3^{\dagger}$ *clock cycles*	
`SUB 1, R2`	4
`JMP lp`	4
`lpdone:`	
Total:	$16 + 5m + 3^{\dagger}m$

Figure 5.3: FAST CPU Bus Utilization Loop Code with m Bus Operations

waits for an interrupt from each FAST CPU. The Nios32 then activates the counters in the testing module, and writes the desired number of iterations of the bus–busy loop to perform to each processor which begins the test. The number of iterations is stored in register `R2` on the FAST CPU, which is decremented in the bus loop on the FAST CPU. The Nios32 processor enters a busy–wait loop, while the FAST CPUs execute the bus loop. An interrupt is used by the FAST CPUs to signal the completion of the task to the Nios32 processor, at which point the interrupting FAST CPU does not perform any further bus transactions. The testing module counters are disabled when all FAST CPUs in the test have responded with an interrupt so the results can be read.

5.2.2 Theoretical and Experimental Results

For any single FAST CPU, if the number of `MAR` instructions is increased (which adds 8 cycles to the loop for each `MAR`), then the total number of clock cycles required to complete the test should increase by the same amount if there are no bus collisions. If there are bus collisions, then the increase in the total number of cycles may be linear and greater than 8 (indicating a constant number of collisions added for each `MAR` instructions), or may be something beyond linear indicating that the `MAR` instructions have a cumulative effect on the bus collisions.

Comparing the 1–, 2–, and 3–processor case, if there are no bus collisions (hence, no memory bottleneck) the total time to execute the task on a single processor will be the same as that on three processors (allowing for $t_{overhead}$ clock cycles, from Table 4.3, to start the task on all processors, versus only starting it on one processor).

Recall that the bus utilization referred to in the testing is not measured. Bus utilization is the independent variable and it is computed from the code in Figure 5.3 assuming that the bus accesses never stall. The equation to convert the number of `MAR` instructions (m) to bus utilization is shown in Equation 5.1. A bus utilization of 1 indicates that the bus is 100% busy.

$$\text{bus utilization} = \frac{3m}{8m + 16} \tag{5.1}$$

With only a single `MAR` instruction in the loop, the bus utilization demanded by a single FAST CPU is $\frac{3}{8+16} = 0.125$, or 12.5% of the total number of clock cycles. At 12.5% per–processor, the Avalon Bus should be able to support three FAST CPUs without showing evidence of a bottleneck at the main memory, since the total usage ($12.5\% \times 3 = 37.5\%$) is much less than 100%. If it were possible to insert an infinite number of `MAR` instructions[3], however, then each FAST CPU could keep the bus busy $\lim_{m\to\infty} \frac{3m}{8m+16} \times 100 = 37.5\%$[4] of the total number of clock cycles. Even with 100 `MAR` instructions, the demanded bus utilization percentage is 36.67% for each FAST CPU, which is close to the limit of 37.5% so it serves as the maximum test point. This percentage is slightly over one third of the bus, per–processor, so it is expected that with three FAST CPUs ($37.5\% \times 3 = 112.5\%$ or $36.67 \times 3 = 110.01\%$) there would be many collisions on the bus. In addition, the Nios32 processor is constantly fetching instructions from main memory, which only furthers the expectation of collisions.

Due to the nature of the FAST CPU instruction set, it is not possible to construct a FAST CPU program that attempts to use more than 37.5% of the Avalon Bus. As each `MAR` instruction is added

[3]Unfortunately, the FAST CPU does not have sufficient RAM to store an infinite number of `MAR` instructions.

[4]The fact that this value is the same as the 3–FAST CPU case with a single `MAR` instruction is purely coincidental.

to the bus–busy loop, meaning that m is increased by 1, the total execution time should increase linearly under ideal conditions (no bus collisions). The bus utilization increase, however, is shown in Equation 5.2 and is dependent on the reciprocal of m and m^2.

$$\begin{aligned}
\Delta_{\text{bus utilization}} &= \text{bus utilization}(m+1) - \text{bus utilization}(m) \\
&= \frac{3(m+1)}{8(m+1)+16} - \frac{3m}{8m+16} \\
&= \frac{(8m+16)\cdot(3m+3)}{(8m+16)\cdot(8m+24)} - \frac{(8m+24)\cdot 3m}{(8m+24)\cdot(8m+16)} \\
&= \frac{(24m^2+64m+48)-(24m^2+64m)}{64(m^2+5m+6)} \\
&= \frac{3}{4(m+2)(m+3)}
\end{aligned} \tag{5.2}$$

It follows, then, that if the total number of clock cycles have a linear relation with the number of `MAR` instructions, and the bus utilization is dominated by the squared reciprocal of the number of `MAR` instructions (in such a way that the bus utilization never exceeds 0.375), then when the bus utilization is plotted on a linear axis against the total number of clock cycles, the total execution time will be non–linear, approaching positive infinity at an asymptote on 37.5% bus utilization.

Tables 5.2, 5.3, and 5.4 show theoretical and selected experimental results for the 1–, 2–, and 3–FAST CPU case respectively. The number of `MAR` instructions (m), and the number of loop iterations (i) are parameters to the program. The remaining variables are either measured or calculated, and all contain a second subscript (n, for example $t_{cc,n}$) which is either 1, 2, or 3. The second subscript denotes whether the test result refers to a 1–, 2–, or 3–FAST CPU test. The variables are as follows:

- $t_{cc,n}$ – Measured – The total clock cycles to complete the test, which consists of i iterations of a FAST CPU bus–busy loop containing m `MAR` instructions.
- $t_{rs,n}$ – Measured – Total number of read cycles observed at the SRAM. This is the counter value from probe p_7 in Section 5.1.

- $t_{r1,n}, t_{r2,n}, t_{r3,n}$ – Measured – Total number of read cycles for FAST CPU 1, 2, or 3 both waiting for the Avalon Bus, or using it (probes p_1, p_2, and p_3 respectively).

- $t_{overhead,n}$ – Measured – The number of clock cycles for 0 iterations of the bus–busy loop. It represents the amount of time to send a request to the FAST CPU, process the 0 iterations, and process the resulting FAST CPU interrupt. $t_{overhead,n}$ should be slightly larger than $t_{overhead}$ in Table 4.3 since $t_{overhead,n}$ includes the time to process the 0–iteration loop. $t_{overhead,n}$ will be constant across all tests with the same number of processors, and is used in the computation of $t_{ideal,n}$ and $t_{est,n}$.

- $t_{ideal,n}$ – Computed – Ideal total number of clock cycles for the test. Equation 5.3 shows this computation, which is the total number of cycles that are required if no Avalon Bus collisions occur. This value is also the minimum amount of time required to complete the bus test. The quantity $3^{\dagger}$ from the code in Figure 5.3 has been replaced with 3 in this equation because with no collisions, each bus transaction will only require 3 clock cycles to complete.

  Notice that $t_{ideal,n}$ for $n = \{1, 2, 3\}$ will have the same slope for a given m and i, because ideally, the bus transactions can be interleaved so that no collisions occur. Practically this will not be the case, especially in tests where collectively, the FAST CPUs attempt to use more than 100% of the bus.

$$t_{ideal,n} = t_{overhead,n} + (16 + 5m + 3m)i \tag{5.3}$$

- $t_{est,n}$ – Computed from Theoretical and Measured Values – Estimated number of total clock cycles, shown in Equation 5.4. This estimation replaces the theoretical $3^{\dagger}m$ parameter from the code in Figure 5.3 with a measured quantity, but leaves the remaining parts of the equation unchanged. The $3^{\dagger}m$ quantity may change for each of the i iterations, so it is not easily measurable, however $3^{\dagger}mi$ is a measurable quantity and is equal to $t_{r1,n}$, $t_{r2,n}$, or $t_{r3,n}$ for the individual FAST CPUs. Equation 5.4 replaces the $3mi$ parameter in Equation 5.3 with the

maximum of $t_{r1,n}$, $t_{r2,n}$, or $t_{r3,n}$ from the experimental results.

$$t_{est,n} = t_{overhead,n} + (16 + 5m)i + MAX(t_{r1,n}, t_{r2,n}, t_{r3,n}) \tag{5.4}$$

Any deviation from the ideal calculation ($t_{ideal,n}$) should be manifested in extra bus cycles. So the number of bus cycles in the `MAR` instruction will not be the ideal 3, but more. The decision to use the maximum of the three FAST CPU read quantities comes from the fixed priority arbitration scheme used by the Avalon Bus (see Section 3.1.2). If simultaneous requests for mastership are pending at the arbitrator, the one with the highest priority is allowed to proceed. Therefore, the 3^{rd} FAST CPU (defined last in the SOPC Builder) will always be serviced last, and will always have to wait the longest. Following this logic, then, the total number of clock cycles for the 2– and 3–FAST CPU case should be equal to the time it takes the slowest CPU to complete the task.

- $\Delta_{ideal,n} = t_{cc,n} - t_{ideal,n}$ – Computed – The difference from the actual number of clock cycles ($t_{cc,n}$) to the computed ideal number of clock cycles ($t_{ideal,n}$). This quantity removes the ideal portion of the total number of clock cycles, leaving a quantity which shows how the Avalon Bus collisions impacted the results. $t_{ideal,n}$ can be viewed as the minimum time required to complete the bus test, and collisions only increase the total number of cycles beyond $t_{ideal,n}$. If there are no Avalon Bus collisions, then $t_{cc,n}$ will equal $t_{ideal,n}$, and this quantity will be 0.

- $\Delta_{est,n} = t_{cc,n} - t_{est,n}$ – Computed – The difference from the actual number of clock cycles ($t_{cc,n}$) to the computed estimated number of clock cycles ($t_{est,n}$). This quantity should be 0 for all cases in the experimental results, since it is computed using all available knowledge of the Avalon Bus and how the testing code operates. If this value is not 0 (or close to 0), it means that the theory used in the calculations is incorrect.

Table 5.2: Selected Bus Utilization Test Results for 1 FAST CPU

MAR (m)	**Iter** (i)	**%Bus** (bus)	**Total Cycles** ($t_{cc,1}$)	**SRAM Read Cycles** ($t_{rs,1}$)	**FAST CPU 1 Read Cycles** ($t_{r1,1}$)	**Ideal Total Cycles** ($t_{ideal,1}$)	**Estimate Total Cycles** ($t_{est,1}$)	$t_{cc,1} - t_{ideal,1}$ ($\Delta_{ideal,1}$)	$t_{cc,1} - t_{est,1}$ ($\Delta_{est,1}$)
1	0	12.5	417[1]	231	0	417	417	3	3
	200	12.5	5220	2793	603	5217	5220	3	0
	400	12.5	10020	5353	1203	10017	10020	3	0
	600	12.5	14820	7913	1803	14817	14820	3	0
	800	12.5	19620	10473	2403	19617	19620	3	0
	1000	12.5	24420	13033	3003	24417	24420	3	0
10	0	31.25	417	231	0	417	417	3	3
	200	31.25	20274	11906	6656	19617	20273	657	1
	400	31.25	40134	23586	13317	38817	40134	1317	0
	600	31.25	59994	35266	19977	58017	59994	1977	0
	800	31.25	79855	46947	26637	77217	79854	2638	1
	1000	31.25	99715	58626	33297	96417	99714	3298	1
50	0	36.06	423	233	0	417	417	6	6
	200	36.06	87338	52589	33726	83617	87343	3721	−5
	400	36.06	174151	104896	67332	166817	174149	7334	2
	600	36.06	261189	157315	101177	250017	261194	11172	−5
	800	36.06	348113	209677	134901	333217	348118	14896	−5
	1000	36.06	434769	261906	168351	416417	434768	18352	1
100	0	36.76	426	234	0	417	417	9	9
	200	36.76	171529	103583	67911	163617	171528	7912	1
	400	36.76	342648	206942	135830	326817	342647	15831	1
	600	36.76	513771	310305	203753	490017	513770	23754	1
	800	36.76	684891	413665	271673	653217	684890	31674	1
	1000	36.76	856003	517017	339585	816417	856002	39586	1

[1]This is $t_{overhead,1}$, the time to do 0 iterations.

Table 5.3: Selected Bus Utilization Test Results for 2 FAST CPUs

MAR (m)	**Iter** (i)	**%Bus** (bus)	**Total Cycles** ($t_{cc,2}$)	**SRAM Read Cycles** ($t_{rs,2}$)	**FAST CPU Read Cycles**		**Ideal Total Cycles** ($t_{ideal,2}$)	**Estimate Total Cycles** ($t_{est,2}$)	$t_{cc,2} - t_{ideal,2}$ ($\Delta_{ideal,2}$)	$t_{cc,2} - t_{est,2}$ ($\Delta_{est,2}$)
					CPU 1 ($t_{r1,2}$)	**CPU 2** ($t_{r2,2}$)				
1	0	12.5	650[1]	358	0	0	648	648	2	2
	200	12.5	5743	3200	612	1087	5448	5935	295	−192
	400	12.5	11042	6138	1222	2186	10248	11234	794	−192
	600	12.5	16357	9079	1825	3301	15048	16549	1309	−192
	800	12.5	21669	12017	2437	4413	19848	21861	1821	−192
	1000	12.5	26966	14957	3038	5510	24648	27158	2318	−192
10	0	31.25	650	359	0	0	648	648	2	2
	200	31.25	22881	14735	6577	9225	19848	23073	3033	−192
	400	31.25	45291	29180	13155	18435	39048	45483	6243	−192
	600	31.25	67682	43603	19736	27626	58248	67874	9434	−192
	800	31.25	90084	58030	26294	36828	77448	90276	12636	−192
	1000	31.25	112500	72497	32915	46044	96648	112692	15852	−192
50	0	36.06	653	360	0	0	648	648	5	5
	200	36.06	99965	66445	33767	46305	83848	100153	16117	−188
	400	36.06	199370	132559	67681	92506	167048	199554	32322	−184
	600	36.06	298785	198722	101552	138725	250248	298973	48537	−188
	800	36.06	398215	264881	135419	184959	333448	398407	64767	−192
	1000	36.06	497700	331025	169255	231236	416648	497884	81052	−184
100	0	36.76	656	362	0	0	648	648	8	8
	200	36.76	196605	130971	67546	92949	163848	196797	32757	−192
	400	36.76	392854	262301	135370	185998	327048	393046	65806	−192
	600	36.76	589204	393391	203068	279148	490248	589396	98956	−192
	800	36.76	785529	524483	270778	372273	653448	785721	132081	−192
	1000	36.76	981733	655534	338502	465277	816648	981925	165085	−192

[1]This is $t_{overhead,2}$, the time to do 0 iterations.

Table 5.4: Selected Bus Utilization Test Results for 3 FAST CPUs

MAR (m)	**Iter** (i)	**%Bus** (bus)	**Total Cycles** ($t_{cc,3}$)	**SRAM Read Cycles** ($t_{rs,3}$)	**FAST CPU Read Cycles**			**Ideal Total Cycles** ($t_{ideal,3}$)	**Estimate Total Cycles** ($t_{est,3}$)	$t_{cc,3} - t_{ideal,3}$ ($\Delta_{ideal,3}$)	$t_{cc,3} - t_{est,3}$ ($\Delta_{est,3}$)
					CPU 1 ($t_{r1,3}$)	**CPU 2** ($t_{r2,3}$)	**CPU 3** ($t_{r3,3}$)				
1	0	12.5	885[1]	492	0	0	0	884	884	1	1
	200	12.5	5759	3413	666	666	1055	5684	6139	75	−380
	400	12.5	11014	6523	1329	1331	2109	10484	11393	530	−379
	600	12.5	16300	9637	1996	1996	3195	15284	16679	1016	−379
	800	12.5	21558	12749	2658	2658	4253	20084	21937	1474	−379
	1000	12.5	26861	15874	3329	3329	5356	24884	27240	1977	−379
10	0	31.25	885	493	0	0	0	884	884	1	1
	200	31.25	23447	16622	6866	6759	9742	20084	23826	3363	−379
	400	31.25	46435	32961	13730	13551	19530	39284	46814	7151	−379
	600	31.25	69712	49421	20539	20366	29607	58484	70091	11228	−379
	800	31.25	92923	65846	27386	27217	39622	77684	93306	15239	−383
	1000	31.25	115937	82142	34275	33955	49432	96884	116316	19053	−379
50	0	36.06	885	494	0	0	0	884	884	1	1
	200	36.06	101553	74737	33147	32898	47851	84084	101935	17469	−382
	400	36.06	202561	149034	66329	65709	95659	167284	202943	35277	−382
	600	36.06	303562	223484	99440	98717	143459	250484	303943	53078	−381
	800	36.06	404496	297919	132675	131740	191191	333684	404875	70812	−379
	1000	36.06	505024	372004	165481	164545	238519	416884	505403	88140	−379
100	0	36.76	885	496	0	0	0	884	884	1	1
	200	36.76	197967	145120	64087	63879	94262	164084	198346	33883	−379
	400	36.76	395648	289705	128064	127914	188743	327284	396027	68364	−379
	600	36.76	593463	435000	192559	192000	283358	490484	593842	102979	−379
	800	36.76	791575	581390	257920	256677	378270	653684	791954	137891	−379
	1000	36.76	988962	724677	320868	320164	472457	816884	989341	172078	−379

[1]This is $t_{overhead,3}$, the time to do 0 iterations.

5.2.3 Discussion of Results

Taking any column except $\Delta_{est,n}$ from the Tables 5.2, 5.3, or 5.4 and dividing by the number of bus iterations (i) gives approximately the same number down the entire column. This is particularly true for higher values of i where the effects of the constant overhead time for each test is minimized. This means that the number of iterations is largely inconsequential to the results, thus, all analysis is done for $i = 1000$.

The values of $\Delta_{est,n}$ for $n = 1$ (the single FAST CPU case) are approximately 0 for all i, indicating that the equations can almost perfectly predict the behaviour of the Avalon Bus. However, the equations over–predict the total required cycles by a constant 192 clock cycles for the 2–FAST CPU case, and by almost double that, 379 cycles, in the 3–FAST CPU case. Notice that the error in the prediction is constant and independent of m and i, for $i \neq 0$. When i is greater than 0, there is a non–trivial amount of work to be done by each FAST CPU in the test. To begin a test, each FAST CPU is given a "go" command in sequence, so it is possible that the first FAST CPUs in the test can be working while the remaining FAST CPUs are being given the instruction to begin the test. This parallelism is most likely reducing the observed value of $t_{overhead,n}$ by a near–constant amount for each processor added. At $i = 0$, where $t_{overhead,n}$ is measured, there is insufficient processing time in the task for this parallelism to significantly change the measured value of $t_{overhead,n}$. An alternative measurement of $t_{overhead,n}$ that compensates for this parallelism would likely reduce $\Delta_{est,n}$ to near 0 for all cases.

Figure 5.4 shows a graph of $t_{ideal,1}$, $t_{cc,1}$, $t_{cc,2}$, $t_{cc,3}$, $\Delta_{ideal,1}$, $\Delta_{ideal,2}$, and $\Delta_{ideal,3}$ versus the number of `MAR` instructions in the bus–busy loop. The graph contains a datapoint for each m in $1 \leq m \leq 100$. $t_{ideal,1}$ is a linear function (defined in Equation 5.3). $t_{ideal,2}$ and $t_{ideal,3}$ are not plotted, as they follow the same slope as $t_{ideal,1}$ with just a different y–axis intercept. The graph shows that the three $t_{cc,n}$ and plots are approximately linear, which is verified by a linear regression and standard deviation calculation, shown in Table 5.5. Since the $t_{cc,n}$ curves are (approximately) linear the $\Delta_{ideal,n}$ must also be approximately linear, as they are computed by subtracting a linear

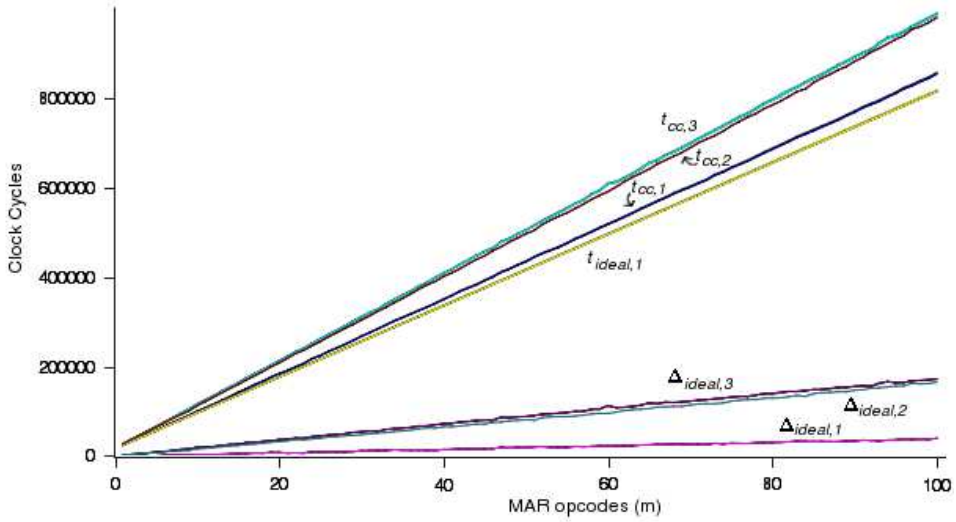

Figure 5.4: Number of MAR Opcodes versus Total Cycles

equation from the set of $t_{cc,n}$ data points. This means that the extra bus cycles due to collisions increases linearly as the number of MAR instructions (m) is increased.

It is evident from the graph in Figure 5.4 that the 2– and 3– FAST CPU cases execute in nearly the same amount of time, which is a somewhat surprising result. It suggests that the bus transaction interleaving in the 2–FAST CPU tests leaves sufficient free bus cycles with main memory that a third processor may be added which effectively uses these holes. Investigating this specific interleaving further through simulation or a digital scope would be a logical next step, and is discussed in Section 6.3. However for the goals of this research, only the results of multiple processors using the memory bus are useful, not how the bus specifically interleaves the requests to reach the observed values.

The data in Table 5.5 shows the results of a linear regression for the $t_{cc,n}$ and $\Delta_{ideal,n}$ data sets. The correlation index is very close to 1 (perfect correlation) for all cases, indicating that the linear regression is a good fit for the data.

Recall that the $t_{cc,n}$ data points are for $i = 1000$ iterations of the bus–busy code in Figure 5.3. For each MAR instruction added to the loop in the 1–FAST CPU case, there are approximately 8388

Table 5.5: Linear Regression Results

Line	**y–intercept**	**slope**	**Correlation Index r^2**	**Standard Error of the Regression (SE_y)**
$t_{cc,1}$	15630.08	8388.28	0.99999	796.43
$t_{cc,2}$	15770.39	9643.76	0.99998	1168.44
$t_{cc,3}$	18517.57	9727.86	0.99997	1623.48
$\Delta_{ideal,1}$	−786.91	388.29	0.99508	796.43
$\Delta_{ideal,2}$	−887.91	1643.76	0.99941	1168.44
$\Delta_{ideal,3}$	1633.57	1727.86	0.99896	1623.48

extra cycles required to complete all 1000 iterations (the slope of $t_{cc,1}$). Dividing by $i = 1000$ shows 8.388 extra cycles for each `MAR` instruction per iteration. A `MAR` instruction requires 8 cycles (see Figure 5.3), leaving 0.388 cycles cycles due to bus collisions for each iteration. The value of 0.388 extra cycles can also be found by taking the slope of the 1–FAST CPU case differential line, $\Delta_{ideal,1}$, and dividing by the number of iterations (1000).

For the 3–FAST CPU test, there are 1728 extra bus collision cycles added, or 1.728 cycles per added `MAR` instruction per iteration. The fact that the total time required to complete the test is not three times the total time required for the 1–FAST CPU tests, shows that there is indeed parallelization occurring (parallelization that is forced to serialize across all bus masters for bus accesses).

This analysis of the number of clock cycles versus the number of `MAR` instructions is valid only for the bus–busy loop code in Figure 5.3. For general applicability to any piece of FAST CPU code, an analysis of the percentage of extra cycles versus the percentage of bus utilization is required. First, however, Figure 5.5 shows a graph of $t_{ideal,1}$, $t_{cc,1}$, and $t_{cc,3}$ versus the bus utilization, which demonstrates the asymptote when the per–processor attempted bus utilization approaches 37.5%. It is evident from this graph that the experimental results agree with the theory. Also notice in this graph that even for the 3–FAST CPU case, the per–processor bus utilization goes well beyond 33%. Recall that the bus utilization refers to the *attempted* bus utilization for each processor, not the actual utilization. The attempted bus utilization is set by the number of `MAR` instructions in the code in

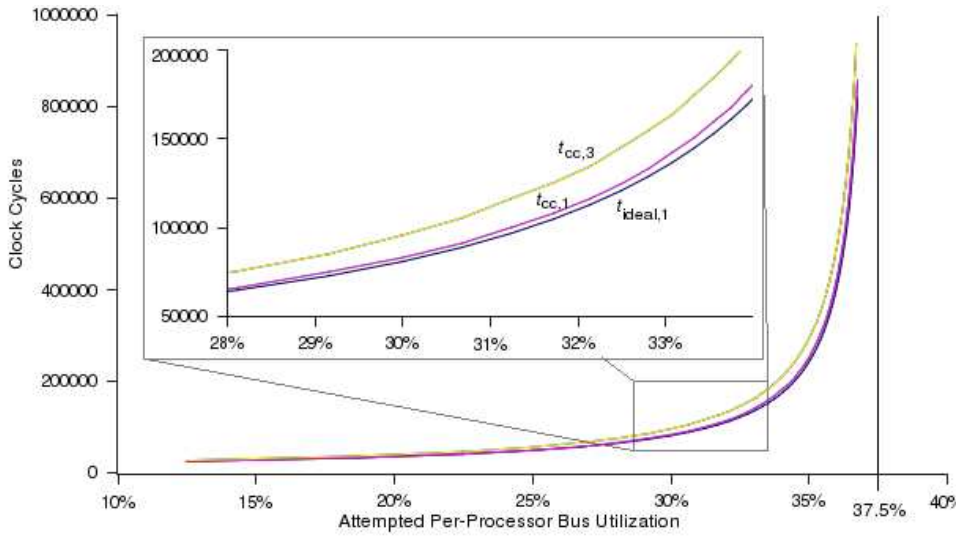

Figure 5.5: Attempted Per–Processor Bus Utilization versus Ideal and Experimental Total Cycles

Figure 5.3. For all cases, the actual total bus utilization is 100%, since the Nios32 processor (which is constantly fetching instructions while executing the busy–wait loop) is able to fill any idle bus cycles, and sometimes collides with the FAST CPUs in the process.

Figure 5.6 shows the extra cycles caused by Avalon Bus collisions for each 1000 total clock cycles at a given bus utilization. Equation 5.5 shows how the extra cycles are computed for each value of m.

$$\text{Extra Cycles} = \frac{t_{cc,n} - t_{ideal,n}}{t_{ideal,n}} \times 1000 \tag{5.5}$$

As in Figure 5.4 it is interesting to note that the results for the 2 and 3 FAST CPU systems are somewhat close. It is also interesting to note that the extra clock cycles level off around 45 cycles for the 1–FAST CPU case, and approach 200 and 210 for the 2– and 3–FAST CPU cases respectively, without showing much evidence of leveling. This figure represents the general result of the measurements and bus utilization analysis. It relates a measurable quantity for any FAST CPU program to a number of extra clock cycles. It can be used to determine how many extra clock

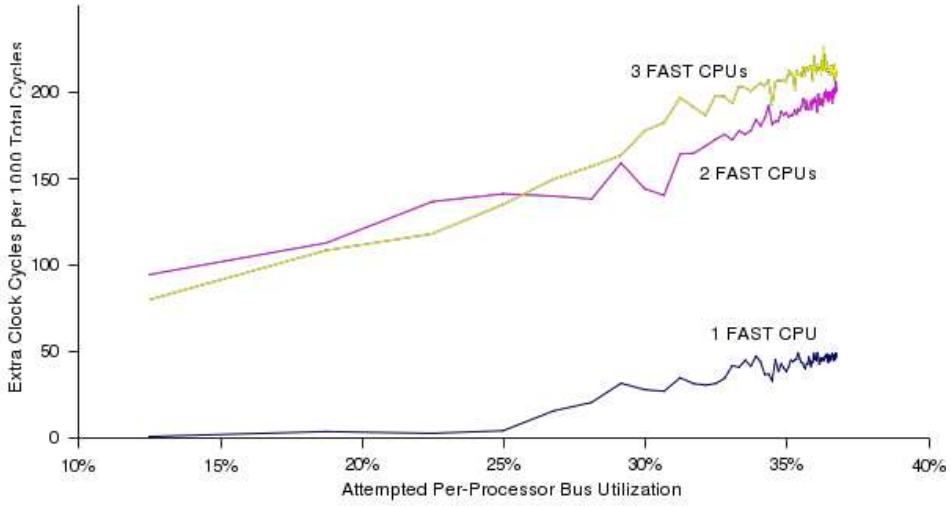

Figure 5.6: Attempted Per–Processor Bus Utilization versus Extra Clock Cycles per 1000 Total Cycles

cycles per 1000 total system cycles will be incurred for any FAST CPU program, given the number of FAST CPUs and the percentage of the Avalon Bus used by the FAST CPU program.

Recall from Section 4.2 that the FAST CPU was designed with a 1 kB internal RAM to avoid fetching instructions out of main memory. The average CPI of the FAST CPU is 4, and each request to the main memory requires 3 bus cycles (unless they are "pipelined", which is what the Nios processor does to achieve a CPI of 1 [Alt03b]). Assuming the CPI is to remain at 4, it means that a single FAST CPU would need to keep the Avalon Bus busy $\frac{3 \text{ cycles}}{4 \text{ cycles}}$, or 75% of the time. None of the collected data extends to 75% bus utilization, however extrapolating from Figure 5.5 or Figure 5.6 indicates that there would be a high frequency of bus collisions. Therefore, the decision to use the 1 kB internal RAM was a "good" design decision.

5.3 FAST CPU Computation Testing

To test the relative computational performance of the FAST CPU compared to a Nios32 processor and whether the Nios32 processor can act as a controller for several FAST CPUs, a factorization test was chosen. Factorization requires no memory accesses other than the initial bus operation to write the number to be factored into each FAST CPU, and a bus operation to read the result back from the FAST CPU once factorization is complete. The read–back is not strictly necessary, but useful to determine if the FAST CPU factored the number correctly. After a write to each CPU, the only activity on the Avalon Bus is the Nios32 CPU executing the busy–wait loop, waiting for a FAST CPU to finish and respond with an interrupt.

5.3.1 Test Procedure

The test procedure is based on the flow graph in Figure 5.1, and only adds a few minor steps. The Nios32 processor first programs each FAST CPU with a piece of code which factors a 32–bit number, given in Appendix A.2. Once complete, it picks two random prime numbers between $60,000$ and $65,535$, multiplies them, and writes the product (which is not more than 32–bits) to each FAST CPU. It then enables the testing module, and instructs each CPU to begin factoring through the use of a second command. While each FAST CPU is factoring a number, the Nios32 processor enters the busy–wait loop, and waits for all FAST CPUs to respond. Each CPU is an independent processing entity, and each is working on the same problem, so each of the CPUs will take exactly the same number of cycles to factor the number (the FAST CPU factorization routine is deterministic, meaning that there is no randomness in it). When all FAST CPUs have responded with an interrupt, the testing module counters are deactivated to read the values. The entire test is repeated with $100,000$ different 32–bit numbers.

Table 5.6: Factorization Test Results

	Total Cycles	Speedup
Nios32 factoring	30,353,043	1.000x
1 FAST CPU	39,718,061	0.764x
2 FAST CPUs	39,718,458	1.528x
3 FAST CPUs	39,718,855	2.293x

5.3.2 Experimental Results

Table 5.6 presents the average results of the factorization test. The clock cycle counts in the table represent the average clock cycles required to factor a single 32–bit integer. It is immediately evident that the Nios32 processor outperforms the FAST CPU by approximately 25% in factoring the number. For the 1, 2, and 3 FAST CPU case in Table 5.6, the Nios32 processor is only serving numbers to the FAST CPUs, it is not performing any of the factoring. It can be seen that that the Nios32 processor is capable of controlling all three FAST CPUs without slowing down the system at all. In the 3 FAST CPU case, the number is factored once by each CPU in approximately the same amount of time as on a single FAST CPU.

For convenience, Table 5.6 also shows the speedup using the Nios32 factor results as the baseline. The speedup is computed by normalizing the results, which is explained in the next section, and dividing each normalized value by the clock cycles required for the Nios32 processor to perform the factorization.

5.3.3 Discussion of Results

It is expected that the Nios32 processor can outperform a single FAST CPU in processing throughput, so a 25% faster time posted by the Nios32 processor is not surprising at all. The time difference is largely because the Nios32 is an optimized processor that contains a 5–stage pipeline that most instructions can complete without stalling, giving an average CPI of approximately 1 [Alt03b]. The average CPI on the FAST CPU (which is unoptimized, and contains no pipeline) is 4.

If the FAST CPU also had an average CPI of 1, then it would theoretically perform four times faster and complete the factorization tests in $10,000,000$ cycles (compared to $30,000,000$ for the Nios32 processor). This would mean the Nios32 processor is approximately three times slower than the FAST CPU which seems a bit dubious given both processors are somewhat simple and both are restricted to operating at 33 MHz (so the maximum pipeline stage size is essentially fixed). There are, however, two differences that can explain this theoretical result, and show that this is not an unreasonable expectation for a FAST CPU:

- **Programming Language Used** – Eventhough the same algorithm was used for both implementations, the code for the Nios32 processor was written in C, and compiled using all optimizations available in the compiler. The factorization code for the FAST CPU was written directly in assembly, since there is no C compiler which can target the FAST CPU (yet, see Section 6.3).

- **Instruction Set Differences** – The FAST CPU opcode set has been tailored specifically to run a single program. Time is saved by not using a stack, and by having addressing modes which support immediate operands. The FAST CPU is capable of retrieving a 32–bit constant operand as part of an instruction, whereas the Nios32 processor must load any non-register operands into a register first (requiring 2 additional instructions, and potentially more if the previous contents of the target register need to be saved on the stack.)

There is a difference of $39,718,458 - 39,718,061 = 397$ total clock cycles between the 1 and 2 FAST CPU case, and also between the 2 and 3 CPU case. This is the overhead required to write to additional processors, and is only slightly larger than the minimum overhead ($t_{overhead} = 383$ in Table 4.3). The $397 - 383 = 15$ extra cycles can be attributed to the fact that the FAST CPU factor code always retrieves the number to factor from the Avalon Bus slave interface before responding with an interrupt to the Nios32 processor. It can therefore be concluded that the extra 397 clock cycles incurred for adding additional FAST CPUs to the test is entirely attributable to the overhead

for the additional CPUs. This means that, as expected, there are no Avalon Bus collisions even in the 3–FAST CPU case.

Using the factorization results from Table 5.6, and assuming that the Nios32 processor can perfectly and completely use multiple FAST CPU factorization routines[5], is possible to construct a simple model to describe when it is beneficial to migrate code to a FAST CPU from a purely computational standpoint. Normalizing the factorization results, the Nios32 and the single FAST CPU processor can complete one factorization in the posted time of $30,353,043$ and $39,718,061$ cycles respectively. The 2 and 3 FAST CPU systems are actually doing two and three factorizations in the posted time, so they can complete a single factorization (in theory, under full parallelization) in one half and one third of the time shown in Table 5.6. Figure 5.7 shows the total cycles plotted against the number of FAST CPUs. Dividing each normalized data point by the result of the Nios32 processor gives the speedup, shown in Table 5.6.

In Figure 5.7, the intersection of the *Nios* curve and the *FAST CPU* curve is the breakeven–point, located at slightly less than 1.5 FAST CPUs. Concluding, if the algorithm in question is parallelizable across at least 2 FAST CPUs in such a way that at least 1.5 of the total cycles on the CPUs are dedicated to performing the algorithm, then the system will show a speed increase. This result, however, does not take into account any factor other than speed (other factors include communication delays, total area consumed, design time, and design complexity).

One additional note to the previous analysis is that if the computation can be completed on the Nios32 processor in less than the overhead cost (397 cycles for the factorization) then there is no point whatsoever in migrating the code to a FAST CPU. The cost of the communication will immediately negate any possible gain.

[5]It has been suggested that this be called “embarrassing parallelism”.

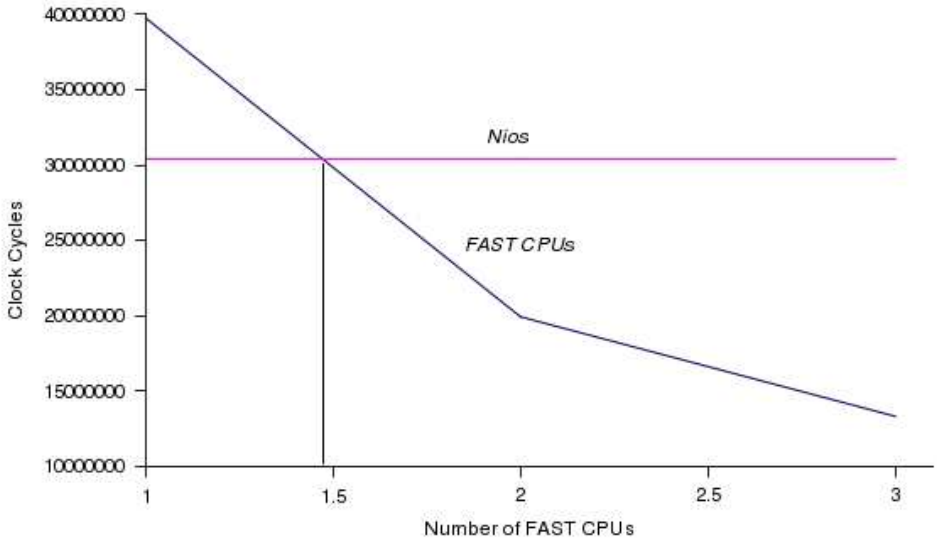

Figure 5.7: Total Normalized Clock Cycles versus Number of FAST CPUs

5.4 Minheap Testing

The minheap was chosen as a test for the FAST CPU system largely because of the availability of results from a custom hardware implementation available in [Bis03], and further, for three reasons given in [Bis03]:

1. Minheaps are used by many mainstream software applications including spreadsheets, databases, simulators, and operating systems.

2. Minheaps use memory resources efficiently.

3. Minheaps are relatively complex to manage with a coprocessor.

5.4.1 Test Procedure

The minheap test measures the amount of time required to fill a minheap to a specified number of entries and empty it over 5 iterations. The tests in [Bis03] actually measured the amount of time

required for 5000 iterations using the Nios timer peripheral. The timer, however, has an unknown start–stop time, and requires regular interrupt servicing which significantly impacts short tests. Consequently, in [Bis03], 5000 iterations were used to remove any impact these unknowns may have had on the results.

At 5000 iterations with the test module, the cycle counters used in this research would overflow. 5 iterations are sufficient for this test because the testing module counters can be started and stopped in 33 clock cycles, which is insignificant compared to the several hundred–million clock cycles required to complete the entire minheap test. It is therefore safe to assume that the entire measured time is for the minheap, and thus that it is safe to run the tests with only 5 iterations.

The Nios32 processor (running the code in Appendix C.2) begins by instructing all FAST CPUs in the test to download the minheap code given in Appendix A.3. If the test is for a Nios–only test, this step is skipped. Once complete, the Nios activates the testing module, and fills the minheap on each processor to the required amount for the test with random key–value pairs. Then, it executes the same number of delete operations to empty the heaps. The entire process is repeated 5 times, at which point the testing module counters are disabled so the results can be read.

5.4.2 Experimental Results

An analysis of the FAST CPU minheap code in Appendix A.3, similar to the clock cycle breakdown of the code in Figure 5.3, shows that the approximate Avalon Bus read utilization should be 4%. The Avalon Bus write cycles utilization will be the same as the read utilization (4%), since each `MAR` opcode has a paired `MAW` opcode in the minheap code. From the graph in Figure 5.6, it is expected that there will be almost no Avalon Bus collisions in the single CPU case. Even with all 3 FAST CPUs active at 8% bus utilization, there should be very few collisions.

Table B.1 in Appendix B gives the results of the test procedure for the Nios–only minheap implementation. Tables B.2, B.3, and B.4 are the results for the 1–, 2–, and 3–FAST CPU implementations respectively.

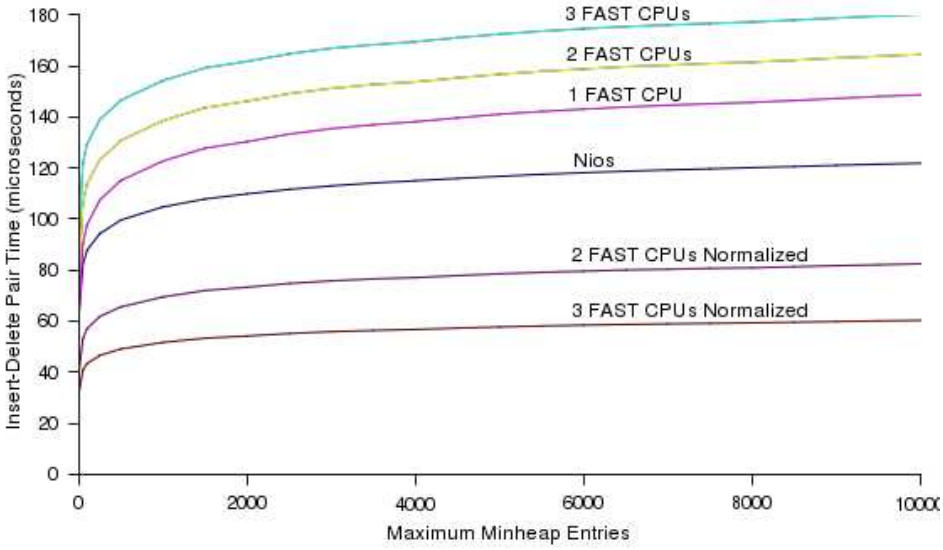

Figure 5.8: Insert–Delete Pair Time versus Minheap Size

It is expected that when the results are graphed, they will show a logarithmic relationship between the total time required to complete the test versus the number of entries in the heap. Inserting into a minheap and deleting from the heap are both $O(log\ n)$ operations. So as the heap increases in size, the total clock cycles required to insert, delete, and re–balance the tree will follow a $log\ n$ relationship, since $\sum_{i=0}^{h} \log(n_i) = \log(\prod_{i=0}^{h} n_i)$.

Figure 5.8 graphs the results from the four raw data tables, plotting the total test cycles against the number of items in the heap, and does indeed show the logarithmic relationship. This figure adds two additional "normalized" curves. Using the same normalization technique that was used in Section 5.3, the results from the two and three FAST CPU systems are normalized to reflect the fact that they are essentially doing two and three times the amount of work in the measured time.

5.4.3 Discussion of Results

The results appear to show no bus collisions whatsoever across all tests. While this is not unexpected, verification can be done several ways to ensure this assessment is correct. First, it is

important to notice that the FAST CPUs all take the same number of Avalon Bus read cycles to perform all tests (the write cycles were not measured, however they behave the same as read cycles, so the same is true) in Tables B.2, B.3, and B.4. This indicates that no FAST CPUs wait for the bus, or that they all wait the same number of cycles.

A better validation is to quantify the difference in clock cycles between the 1–, 2–, and 3–FAST CPU tests for a given number of minheap entries, to account for all the "extra" cycles incurred by adding an additional FAST CPU. For the $10,000$ entry minheap test, the difference between the 1–FAST CPU and 2–FAST CPU clock cycles is $271,069,870 - 244,880,487 = 26,189,383$. The extra clock cycles per insert/delete pair can be computed as follows:

$$26,189,383 \text{ cycles} \times \frac{1 \text{ iteration}}{10000 \text{ pairs}} \times \frac{1}{5 \text{ iterations}} = 523 \frac{\text{cycles}}{\text{pair}} \tag{5.6}$$

Similarly for the differences between the 2– and 3–FAST CPU systems for $10,000$ entry minheaps: $296,927,993 - 271,069,870 = 25,858,123$.

$$25,858,123 \text{ cycles} \times \frac{1 \text{ iteration}}{10000 \text{ pairs}} \times \frac{1}{5 \text{ iterations}} = 517 \frac{\text{cycles}}{\text{pair}} \tag{5.7}$$

From Table 4.3, the overhead time for writing a command to the FAST CPU and processing the interrupt response is $t_{overhead} = 383$ cycles. Each insert/delete pair in the minheap test requires 2 commands: insert and delete. When an additional FAST CPU is added, the inserts, which must be sent to each processor, are in fact parallelized. All the inserts are written, then all the interrupts are processed, allowing the interrupt processing to occur concurrently with any other FAST CPUs still performing the minheap operations. Because of this parallelism, the full overhead of $2 \times t_{overhead} = 766$ cycles can be reduced. The computed extra clock cycles for adding a second FAST CPU (523 cycles), and for a third FAST CPU (517 cycles) are consistent with the expected parallelism. It can be concluded then, that there are no bus collisions occurring, and thus, no bottleneck at the main memory.

The "normalized" curves on the graph in Figure 5.8 illustrate what has been shown in Section 5.3, that there is a performance increase between the 1–FAST CPU and the 2 or 3–FAST CPU implementations when the parallel work that was done is considered.

5.5 Comparison with Previously Tested Systems

This section combines the presented results from the FAST CPU with several other results that were obtained during testing, and compares them to known results from previous research. The hardware–only design used for comparison is from [Bis03], and uses a straight–forward finite state machine model to implement the minheap functionality. Of course, there are larger, faster, and more complex implementations of a heap, such as massively parallel priority queues [Sep02]. The results from [Bis03] are used because they are for a small heap implementation and are readily available. The goal is to illustrate where the system model used in this research fits into the larger body of research and show how the various systems compare using several metrics:

- **Hardware Size** – The Altera APEX series FPGAs measure hardware resources using LEs (Logic Elements) and ESBs (Embedded System Blocks, which implement RAM). The APEX FPGA on the Nios Embedded Processor Development Board contains 8320 LEs and 52 ESBs (providing a total of $106,496$ bits of RAM).

- **Maximum Frequency (f_{max})** – This is the maximum clock frequency of the hardware, which is a direct function of the slowest path through the circuit. This metric comes directly from the Quartus II synthesizer and timing analyzer.

- **Design Complexity** – There are several ways to measure the design complexity including time to design, lines of code in the design, and complexity of the code. The number of lines of code in the design is the only quantitative measure available, so the "complexity" is taken as the number of lines of C code, assembly code, and VHDL. The number of lines of each type of code in the design give a relative indication about the time and effort required for the

Table 5.7: Resource Requirements for Tested Configurable Systems

	Hardware Size				f_{max}	Design Complexity (lines of code)			Configuration
	LEs	**%**	**ESBs**	**%**	**(MHz)**	**VHDL**	**ASM**	**C**	**Time**
Nios only[1]	2385	28	16	31	52.32	0	0	0[2]	$< 300\ ns$
FAST CPU 1	3661	44	28	54	45.11	2072	58	276	Software: $< 300\ ns$ Hardware: $34.5\ ms$
FAST CPU 2	4868	58	40	77	45.11				
FAST CPU 3	6137	73	52	100	45.11				
Hardware only	4247	52	52	100	36.94	0	0	0	$34.5\ ms$

[1]Recall that Table 4.3 presented the size of individual processors, whereas the data in this table is for complete systems.
[2]This table compares only the resource requirements for a system, and does not include any application implementation on top of the design (which is why the Design Complexity is 0 for some designs). Table 5.8 shows the metrics for a minheap implementation on each system.

design. For example, writing, debugging, and testing C code is easier than assembly (ASM) code, and producing correctly functioning VHDL is more difficult (at least for the author) than both assembly and C code.

- **Minheap Reference Time** – The normalized time required to conduct the minheap tests described in Section 5.4.1. The time reported is the clock cycles to complete the $10,000$ entry test divided by the clock speed, 33 MHz.

- **Configuration Time** – This is the time required to change the design that is assisting with application specific acceleration. For software, it is the time required to change the software, and for hardware, the time required to configure the FPGA as specified by the manufacturer. For both hardware and software, the time to download the design to a location where the device can be reprogrammed (or configured) is omitted, it is assumed all designs are readily available or that the downloading can be done in parallel with system operation.

Table 5.7 shows a summary of the metrics for a Nios system, a 1–, 2–, and 3–FAST CPU system, and a hardware–only system, and Table 5.8 shows the relevant metrics for a minheap implementation on each system. For Table 5.7 the design complexity shows the requirements to bring the system to a point where any design can be implemented with it, whereas in Table 5.8 the design

Table 5.8: Minheap Results for Tested Configurable Systems

		Minheap Time (μs)	Design Complexity (lines of code) VHDL	ASM	C
Nios32		121.60	0	0	568
FAST CPU Minheap	1	148.41	0	351	82
	2	82.14			
	3	59.99			
Hardware		6.00	1572	0	0

complexity gives the code required only for the minheap implementation.

5.5.1 Hardware Size

As expected, in Table 5.7, the Nios only system is the smallest in terms of hardware area (ESBs) because all other systems include a Nios processor, Avalon Bus, and various system peripherals along with additional hardware. It is interesting however, that the single FAST CPU system, which can implement almost any algorithm, is approximately 8% smaller than the hardware–only minheap core, which only implements a minheap. Of course, this is only one data point for all the implementations of algorithms for application specific acceleration, but this one implementation shows that a lightweight processor core can produce an overall more area–efficient system than a hardware core (saying nothing about processing speed, see Section 5.5.4). Note that the 3 FAST CPU system has completely used all the available ESBs (memory) on the FPGA, so additional FAST CPUs cannot be added to the system even though there is available area.

5.5.2 Maximum Frequency (f_{max})

The f_{max} results show that there is little penalty in adding hardware to a Nios system. In the case of the FAST CPU systems, the longest delay path is from an automatically generated Avalon Bus slave module for the FAST CPU to the Nios32 master module. The maximum delay in the hardware implementation is through the minheap hardware, but the Nios Embedded Processor Development

Board operates with a 33 MHz clock, so all the maximum frequencies are well within the board specifications.

5.5.3 Design Complexity

In Table 5.7, the code required for the Nios32 processor and related peripherals, which is automatically generated by the SOPC Builder, is excluded. The Nios–only and hardware–only systems require no additional code. The FAST CPU systems require the development of the FAST CPU (VHDL), the boot loader (FAST CPU assembly), and the API (C code), before any application can be developed on top of a FAST CPU system. It is important to distinguish between this work, and the work required to implement an algorithm on each of the configurable systems, as the FAST CPU complexity reported in Table 5.7 has been done, and never has to be done again for any algorithm implemented on the FAST CPU system.

Table 5.8 shows the lines of code in the minheap designs for the Nios processor, the FAST CPU systems, and the hardware implementation. The code required for the hardware implementation is slightly smaller than the code required to implement the FAST CPU, which was not an insignificant design[6]. The FAST CPU design, while the smallest in terms of total lines of code, definitely required more effort than the Nios–only software design because of the language used[7]. A FAST CPU C compiler would bring the complexity down to that of the Nios–only software implementation (see Section 6.3 for comments about a FAST CPU C compiler).

5.5.4 Minheap Reference Time

The normalized minheap times have been discussed in Section 5.4.3, except for the hardware–only result. This value (6 μs [Bis03]) is significantly less than the Nios–only system and all the FAST CPU systems, which is expected. The hardware can manage a minheap approximately 20 times

[6]For a rough comparison, the FAST CPU took approximately 8 months to design, develop, and debug.

[7]Again, for a rough comparison, the Nios design was written and debugged, in C, in approximately half an hour, whereas converting that design to FAST CPU assembly took three hours.

faster than the Nios software implementation, and 24 times faster than a single FAST CPU, but only 10 times faster than the 3 CPUs, when parallelization is considered.

5.5.5 Configuration Time

The configuration time for the Nios–only system is taken to be the time to change from one program to another. On a Nios processor that involves simply changing the program counter to a new memory location, and maybe flushing the pipeline, the exact behaviour of the jump instruction is not known. The same is true for the FAST CPU software, provided both programs fit in the 1 kB internal RAM. A simple `JMP` instruction effectively changes the code that the FAST CPU is running. The FAST CPU is also designed to be reconfigured with a new core to add or remove hardware features. The complete configuration of an APEX 20K200EFC484-2X FPGA on the Nios Embedded Processor Development Board takes $34.5\ ms$ [Bis03]. The hardware–only system must always be reconfigured to change the design, so this is the time for changing between custom hardware designs.

5.6 Summary

This chapter has presented the results from several tests conducted with a FAST CPU system. The bus utilization tests have shown a linear relationship between the demands for the Avalon Bus and the resulting clock cycles required to complete the test using the bus. The linearity was verified by a linear regression. The results also show evidence of bus collisions which is expressed in a generalized plot that relates the number of extra bus cycles per 1000 system cycles to the ideal bus utilization of a piece of FAST CPU code.

The computation testing compared the factorization of 32–bit integers on a Nios processor, and on the FAST CPUs. The Nios processor is faster, as expected, because of the design of the processor.

A minheap application was used to test the FAST CPUs as well. The bus utilization results correctly predicted few–to–no bus collisions when the minheap algorithm was analyzed, and the

computation analysis showed that a single FAST CPU would be slower than the Nios, but 2 or 3 FAST CPUs could outperform a Nios processor when parallelism was considered. The minheap results are compared to a hardware–only implementation of a minheap, where it can be seen that the design complexity, design area, and configuration time are improved with a lightweight processor core, but the performance is approximately 24 times slower than the hardware–only core.

Chapter 6

Concluding Remarks

6.1 Thesis Conclusions

A system using programmable lightweight processor cores within a configurable system has been presented. The system design was based on known models of configurable systems. Using the designed system, containing 3 FAST CPUs, it has been shown that it is possible to move functionality from the general purpose processor into a lightweight processor with little penalty. Indeed, exploiting parallelism can actually improve the system performance of a multi–FAST CPU system compared to a Nios–only system. The speed improvements, however, come with a size penalty, as it takes more resources to implement additional processing units to achieve the speedup.

A single FAST CPU has been shown to be smaller than a hardware–only implementation of a minheap, which verifies that it is possible to make a system for application specific acceleration smaller, in terms of area, by using a lightweight processor core instead of a custom hardware core (which includes the 1 kB of internal RAM for instructions in the FAST CPU). In addition, the complexity of the code for the processor core is much less than the hardware description. The FAST CPU system, however was significantly slower than the custom hardware implementation (by factor of 24), which was expected.

Through the bus utilization analysis it was shown that for the particular configuration of the system studied that, even with the Nios processor constantly fetching instructions, congestion at the main memory is not an issue if the FAST CPUs use main memory for bulk data storage. The utilization was compared to the ideal bus usage of an algorithm to derive an expression for finding the number of extra cycles incurred in a program due to bus collisions given the number of FAST CPUs in the system and the average bus utilization. If the FAST CPU fetched instructions from the main memory, instead of from an internal RAM, then it is conceivable that each FAST CPU may be closer to utilizing 75% of the Avalon Bus. While the results from Section 5.2 do not extend beyond 37.5%, the trend indicates that at 75% utilization, bus collisions should occur frequently. The 1 kB of FAST CPU internal RAM (to help keep the FAST CPUs off the bus) was therefore was a good design decision.

A factorization test was also performed to compare the relative performance of the FAST CPU compared to a Nios processor. It showed that the FAST CPU was approximately 25% slower than the Nios processor, which was expected since the Nios processor was designed by professionals over a long period of time. The performance results were used to construct a cost analysis graph, which showed that it becomes advantageous to migrate code to FAST CPUs if the code can be parallelized across at least two processors, and in such a way that no more than 25% of the time on each processor is spent doing synchronization and communication.

The method used to perform bus and computation analysis is presented as a technique to derive equations for any specific configurable system, in general, to predict bus utilization and find a breakeven point in relation to the number of configurable lightweight processor cores required to show a system speedup. Based on the analysis of a minheap in the FAST CPU system, it was suggested that there should be no bottleneck at the memory, and that the 1–FAST CPU case should be slower than the Nios–only system. The normalized 2– and 3–FAST CPU systems, however, should be faster. This theory was verified by experimental observation.

6.2 Challenges Encountered

As with the development of any complicated system involving hardware or software (or in the case of this research, the interaction of both), there will be problems encountered. Some are solvable, some can be worked around, and some persist and must be strategically avoided.

6.2.1 Hardware

One particularly puzzling problem occurred during the bus utilization testing presented in Section 5.2. Figure 6.1 shows $\Delta_{ideal,2}$: the difference between the results of the 2 FAST CPU bus test and the ideal results. The oscillating and somewhat erratic curve represents the test results using CPUs $\{1, 2\}$ in the 3 CPU system, whereas the results from the other two combinations of CPUs ($\{1, 3\}$ and $\{2, 3\}$ as presented in Section 5.2) showed near identical results that agreed with the theory. All testing was done without resynthesizing the system, or even downloading a new system core to the FPGA on the Nios Embedded System Development Board. Resynthesizing and configuring the board again, however, did not change the results. Even in the case where CPUs $\{1, 2\}$ failed to run the test correctly, the single CPU tests all worked successfully (and returned identical results) on each of the 3 CPUs, and the 3 FAST CPU test returned data that is consistent with what is expected. This suggests that it is not a problem with the VHDL design, or with the software. The only theory (so far) to explain this phenomenon is that for the $\{1, 2\}$ test, there is some interference with the parts of the FPGA that are active and inactive that is not encountered in any other case.

A second problem was encountered where occasionally the system would not synthesize correctly. Most likely, this was the optimizer slightly misbehaving. However, this particular behaviour is to be expected with modern synthesis tools, so it is not a profound result, merely an annoying one.

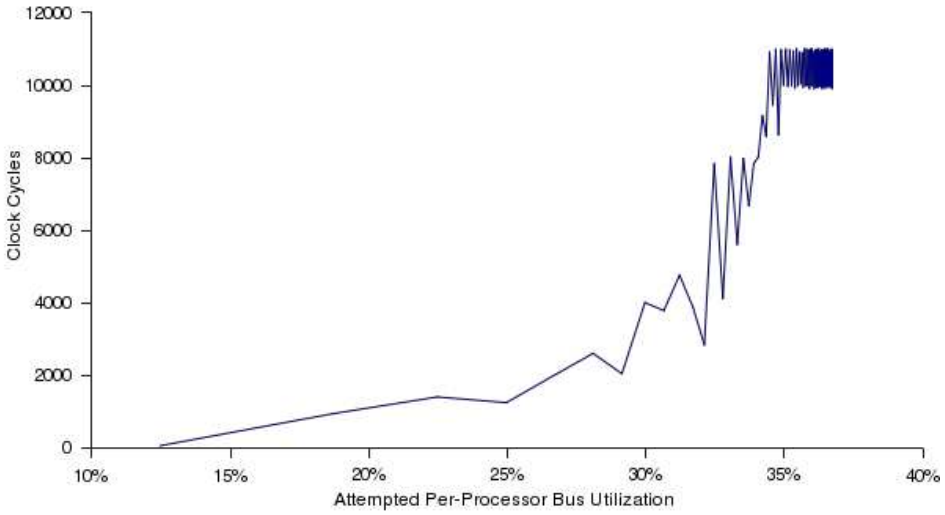

Figure 6.1: Problematic 2–FAST CPU Bus Utilization Results

6.2.2 Software

One notable problem was encountered as the testing module was being developed. The Nios peripheral timer was originally to be used for data collection, but it did not function reliably as an interval timer (see the next section on minheap related challenges). During the development of the testing module, and before the Nios timer peripheral had been removed from the system, if the C code drivers for both the timer and the testing module were included in the system (not called to initialize the hardware, or for any other purpose, just included), then the FAST CPU–2 would cease to function. If, however, either of the drivers were omitted, then without resynthesizing any parts of the system, the entire system would function as expected. This is similar to the aforementioned hardware problem, except that it can be reliably triggered by simply changing the software built and downloaded to the Nios processor.

An undocumented limitation was discovered with respect to global variables in the code for the Nios processor, or at least in the ability of the compiler to generate code for a Nios processor. There is an upper bound on the amount of data which can be placed in the global space, and if too much

data is compiled as global, then it trounces on program memory. This particular feature limited the size of the FAST CPU code which could be included in the Nios processor, as it it is required to be global so the FAST CPU can download it. This problem was worked around by moving all other data structures to the stack of the `main()` routine, and passing them as pointers to the various functions.

6.2.3 Minheap Related

Before the testing module was created (Chapter 5), the initial system testing was done using a minheap, however the results showed evidence of a significant memory bottleneck for the 3–FAST CPU system. That is, it showed little improvement from the three FAST CPU system compared to the single FAST CPU system. It is expected that with a Nios32 processor controlling multiple FAST CPUs, that the one–, two–, and three–FAST CPU tests should return nearly identical test results after compensating for the additional Avalon Bus transactions to communicate with additional processors. An analysis of the FAST CPU implementation of the minheap showed that it would use, in theory, less than 4% of all cycles for bus read operations, and even fewer for write operations. In the worst case, and assuming there are equal numbers of reads and writes, each FAST CPU could potentially consume 8% of the Avalon Bus. It does not make sense then, that any sort of slowdown would occur due to memory congestion, even in the 3–FAST CPU case.

Instead of proceeding with additional and more complicated tests, the testing module was developed and used in the Avalon Bus utilization test and the processor computation tests (in Chapter 5). This was done to investigate whether the system was achieving a bottleneck, or if something else was causing the slowdown from the first minheap test (or if the minheap test itself was flawed).

Comparing the original minheap tests to the testing presented in Chapter 5, the only difference in the tests is the actual measurement device used. The original tests used the Nios timer peripheral, whereas the tests in Chapter 5 used the testing module. The original tests showed similar results as Figure 5.8, with each minheap/delete pair measured as approximately $4\mu s$ faster, meaning the

curves were all shifted down by approximately $4\mu s$. It must be concluded, then, that the Nios peripheral timer was the source of the inaccuracies, which makes sense and can be explained.

The original attempt at measuring the overhead time in Table 4.3 using the Nios timer peripheral gave $t_{overhead} \approx 200$ clock cycles[1]. This number was computed by timing the delivery and response of 1000 commands, and dividing the resulting time by 1000. Equations 5.6 and 5.7 show a differential average insert/delete pair time difference of around 500 cycles between the 1– and 2–FAST CPU test and the 2– and 3–FAST CPU test respectively. Assuming that $t_{overhead} = 200$ cycles, and given a minheap insert/delete pair requires 2 commands, there are at least 100 "missing" cycles. The only source of these missing cycles is bus collisions caused by the `MAR` and `MAW` instructions waiting to use the bus, so they were attributed to that. An investigation of the code, though, showed that in theory there should not be bus collisions, so the tests in Sections 5.2 and 5.3 were performed. As it turns out, those particular tests were also extremely useful towards the goals of the thesis.

6.3 Future Work

There have been several paths highlighted throughout the discussion of the thesis for future work:

- Extend the tests to run on a configurable system that supports more than 3–FAST CPUs, to verify the results hold for additional processors. For example, it has been determined that the smallest Altera Stratix–II FPGA (the EP2S15 [Alt04]) is capable of synthesizing a system with 10 FAST CPUs.

- A C compiler for the FAST CPU would greatly assist programmers in creating code for the FAST CPUs. While writing assembly code can be invigorating for some, developing C code is generally a much faster procedure. The `lcc` [HF95] C compiler is a configurable compiler

[1]The measurement of $t_{overhead}$ using the testing module is known to be correct, since the testing module measurements agree with the computed values for the run–time required for various pieces of code. While it is known that the author has problems with simple integration, it has yet to be proven that the author is not proficient with addition and multiplication. It is therefore assumed that the computed values are correct.

which allows customization of the assembly output. `lcc` also supports calling a custom assembler which could turn the FAST CPU assembly code generated by the compiler into something downloadable into a FAST CPU.

- Further investigation into the bus utilization may uncover additional interesting facts about the FAST CPU and bus interaction. The 2– and 3–FAST CPU bus access tests required approximately the same amount of time to complete. Determining specifically where all the cycles are allocated may show why the 2– and 3–FAST CPU bus test cycle counts are so close, and may allow prediction of behaviour with more FAST CPUs. For example, would there be another jump in clock cycles counts for a 4^{th} processor, would the 5^{th} processor be close to the 4^{th} processor, or is this particular behaviour of similar cycle counts a phenomenon unique to the 2 and 3–FAST CPU systems?

- The next family of Nios processors (Nios2) from Altera support a processor cache if synthesized on a Stratix FPGA. Experimentation on systems with a cache–enabled Nios processor would make it possible to keep the Nios processor "off the bus" as the FAST CPUs operated, and may lead to a more accurate prediction of bus utilization in the system.

- Beyond the immediate future, the goal is to extend the system onto multiple FPGAs, or onto an FPGA that supports reconfiguration, so that each FAST CPU can be contained in an individual configurable entity (as opposed to the research done for the thesis, where the entire system was implemented within a single configurable FPGA). When this occurs, a scheduling algorithm will be needed to schedule the FAST CPU cores, and the specific FAST CPU applications within the cores. The development of this algorithm should prove to be quite academically entertaining.

Appendix A

FAST CPU Code

A.1 Bus–Busy Code

This is the FAST CPU code which floods the Avalon Bus with Read Requests.

---

bus_util_test_fastcpu.s

```
; FAST CPU Avalon Bus Busy
;  David Grant, 2004

; Commands:
;     r0       r1        r2     desc
;  00000100    d2           iterations, and go

.org 0x20

JMP main

main:
    ; Wait for a command
    SLC 0
    SSEG R0
    command_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait

    ; Now, command is in R0, load commands and decide what command it is
```

```
    LD commands, R1
    LD 0, R2
    command_find:
        CMP (R1), R2        ; Compare with 0
        BEQ command_not_found   ; Yes, command not found.
        CMP (R1), R0        ; Check with the loaded command
        BEQ command_found   ; Yes, command match.

        ADD 2, R1       ; Proceed to next array member
        JMP command_find

    command_found:
        ADD 1, R1       ; Increment to the command pointer
        LD (R1), R2
        SPC R15         ; Save pc
        JMP R2          ; Run the function
                    ; Returns here
        JMP main

    command_not_found:
        SSEG 0xfe
        JMP command_not_found
        JMP main


    ; We'll never get here.

command_wait_done:

    SLC 0
    IRQ 1
    command_wait_done_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait_done_wait

    IRQ 0
    JMP R15

.long commands  0xFF000001
.long __c1  #command_reset
.long __c2  0x00000100
.long __c3  #command_set
.long __c_last  0x00000000

; Command FF000001
command_reset:
    SSEG 0xFF
    JMP 0x0

.long testlong 0x12345678
command_set:
```

```
    SLR 1, R0
    SLR 2, R2
    LD (testlong), R1
    lp:
        CMP 0, R2
        BEQ lpdone
        ; Nios code will rewrite this and insert desired number of
        ;  MAR R0,R1 operations here.
        SUB 1, R2
        JMP lp
    lpdone:
    JMP command_wait_done
```

A.2 Factor Code

This is the FAST CPU code for a deterministic factorization routine used to factor a 32–bit number.

---

factor32_fastcpu.s

```
; FAST CPU factor program
;  David Grant, 2004

; Commands:
;     r0        r1      desc
;  00000100      n      set longward to factor
;  00000104     --      start factoring

.org 0x20
JMP main

.long number 0x0
.long factor_count 0x0
.long msb1 0x80000000

.long pause 0x20000

main:
    ; Wait for a command
    SLC 0
    SSEG R0
    command_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait
```

```
    ; Now, command is in R0, load commands and decide what command it is
    LD commands, R1
    LD 0, R2
    command_find:
        CMP (R1), R2        ; Compare with 0
        BEQ command_not_found   ; Yes, command not found.
        CMP (R1), R0        ; Check with the loaded command
        BEQ command_found   ; Yes, command match.

        ADD 2, R1       ; Proceed to next array member
        JMP command_find

    command_found:
        ADD 1, R1       ; Increment to the command pointer
        LD (R1), R2
        SPC R15         ; Save pc
        JMP R2          ; Run the function
                    ; Returns here
        JMP main

    command_not_found:
        SSEG 0xfe
        JMP command_not_found
        JMP main


    ; We'll never get here.


; Command FF000001
command_reset:
    SSEG 0xFF
    JMP 0x0

; Set the number to factor
command_set:
    SLR 1, R1   ; data -> R1
    SSEG R1
    ST (number), R1 ; R1 -> number
    JMP command_wait_done

; Factor!
command_factor:
    ; Load numbers
    LD (number), R1 ; The number we're factoring

    ; div by 2
    LD R1, R3
    LD 2, R2
    AND 1, R3
    CMP 0, R3   ; If lsb is 0, it's divisible by 2
    BEQ factor_done_found
```

```
LD 1, R2    ; Current factor
SSEG R1
factor_loop:
    ; Add 2 to the factor
    ADD 2, R2
    ; Get the factor left aligned into R3
    LD R2, R4
    factor_setup_loop:
        CMP 0, R4   ; See if we're done.
        BEQ factor_setup_loop_done
        LD R4, R5   ; Make a copy, we need it in a sec.
        SHR 1, R3   ; Shift output and input
        SHR 1, R4
        AND 1, R5   ; Test the LSB on the back up
        CMP 0, R5   ; If NE, MSB R3 needs to be 1.

        BEQ factor_setup_loop ; Skip OR if the bit was 0
        OR (msb1), R3
        JMP factor_setup_loop
    factor_setup_loop_done:

    ; Shifted subtractor in R3
    LD R1, R5   ; Remainder in R5
    LD 0, R7    ; Answer in R7

    JMP factor_shift_loop_start

    factor_shift_loop:
        SHR 1, R3   ; Shift subtractor
    factor_shift_loop_start:
        SHL 1, R7   ; Shift answer
        CMP R5, R3  ; Check remainder with subtractor
        BLT shift_no_sub; R5 < R3?
        SUB R3, R5  ; R5 >= R3, subtract.
        OR 1, R7    ; Put a 1 in the answer.
    shift_no_sub:
        CMP R2, R3  ; Check what subtractor we used
        BNE factor_shift_loop

    ; Now, if there is anything left in R5, then R2 is
    ;  not a factor.
    CMP 0, R5
    BEQ factor_done_found

    ; Compare the answer with the factor being checked,
    ;  The answer should be bigger, if it isn't we've
    ;  passed the SQRT threshold, and should stop now.
    CMP R7, R2 ; R7 < R2 ?
    BLT factor_done_no_factor

    JMP factor_loop
```

```
    factor_done_found:
        SLW 0, R2
        JMP command_wait_done

    factor_done_no_factor:
        LD 1, R0
        SLW 0, R0
        JMP command_wait_done


; Fire an IRQ at the Nios, and wait for it to ack it
command_wait_done:

    SLC 0
    IRQ 1
    command_wait_done_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait_done_wait
    IRQ 0
    JMP R15

.long commands  0xFF000001
.long __c1  #command_reset
.long __c2  0x00000100
.long __c3  #command_set
.long __c4  0x00000104
.long __c11 #command_factor
.long __c_last  0x00000000
```

A.3 Minheap Code

This is the FAST CPU code for the minheap implementation.

---

minheap_fastcpu.s

```
; Minheap core, for keeping record of a single minheap
;  David Grant, 2004


; Commands:
;     r0       r1        r2     desc
;  00000100                    set minheap base
```

```
;  00000101 key       data  minheap insert
;  00000102             minheap delete (read result from r0)

.org 0x20
JMP main

.long heap_count 0x0
.long heap_base 0x0

main:
    ; Wait for a command
    SLC 0

    LD (heap_count), R0
    SLW 3, R0

    SSEG R0
    command_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait

    ; Now, command is in R0, load commands and decide what command it is
    LD commands, R1
    LD 0, R2
    command_find:
        CMP (R1), R2        ; Compare with 0
        BEQ command_not_found   ; Yes, command not found.
        CMP (R1), R0        ; Check with the loaded command
        BEQ command_found   ; Yes, command match.

        ADD 2, R1       ; Proceed to next array member
        JMP command_find

    command_found:
        ADD 1, R1       ; Increment to the command pointer
        LD (R1), R2
        SPC R15         ; Save pc
        JMP R2          ; Run the function
                    ; Returns here
        JMP main

    command_not_found:
        SSEG 0xfe
        JMP command_not_found
        JMP main


    ; We'll never get here.


; Command FF000001
```

```
command_reset:
    SSEG 0xFF
    JMP 0x0

command_minheap_base:
    SLR 1, R1
    ST (heap_base), R1
    SLW 1, R1
    JMP command_wait_done

:
; Command 00000101
;  R4 == iteration position, starts a heap count
command_minheap_insert:
    ; Load args
    SLR 1, R1       ; Key
    SLR 2, R2       ; Data

    LD  (heap_count), R4    ; Add 1 to the heap count
    ADD 1, R4
    ST  (heap_count), R4

    minheap_insert_loop:
        LD R4, R5
        SHR 1, R5   ; R5 == position / 2

        ; if position/2 == 0, break loop
        CMP 0, R5
        BEQ minheap_insert_loop_done

        ; Else, proceed with the insert
    minheap_insert_insert:

        LD (heap_base), R6  ; Load heap base
        LD R5, R7       ; Load r5 to r7
        SHL 3, R7       ; r7 == r5 * 4 * 2
        ADD R7, R6      ; find offset in ram
        MAR R6, R7      ; Read into r7

        CMP R1, R7      ;
        ; Now, if R1=key >= R7=minheap[p/2] we don't
        ;  want to swap up the tree anymore, we're done.
        BGE minheap_insert_loop_done

        ; Else, if the parent (array[p2]) is greater than the key
        ;  we're inserting, so we need to rearrange things
        ; Read the data for the last key
        ADD 4, R6
        MAR R6, R10

        LD (heap_base), R6  ; Load heap base
        LD R4, R9       ; Load p
```

```
        SHL 3, R9       ; p = p*4
        ADD R9, R6      ; Add to get location of m[p]
        MAW R6, R7      ; Write m[p/2] to m[p]
        ADD 4, R6       ; Move to data position
        MAW R6, R10     ; Write data too

        LD R5, R4       ; p = p/2

        JMP minheap_insert_loop

minheap_insert_loop_done:

    ; Store the value of key (R1) into p[m]
    LD (heap_base), R6
    LD R4, R9

    SHL 3, R9
    ADD R9, R6
    MAW R6, R1
    ADD 4, R6
    MAW R6, R2

    ; send an interrupt, and wait for the ACK
    JMP command_wait_done


; Min Heap Delete
command_minheap_delete:
    LD (heap_base), R0
    ADD 8, R0   ; Load value in m[1]
    MAR R0, R1

    SLW 0, R1
    ADD 4, R0   ; Load data in m[1]
    MAR R0, R1
    SLW 1, R1

    ; Now at this point we could trigger the interrupt and wait for
    ;  an ack.

    ; Fixup the table.
    LD 1, R0

minheap_delete_loop:

        ; position = p = R0

        ; Compute the left and right locations
        LD R0, R1
        ADD R1, R1  ; R1 = position * 2 = left
        LD R1, R2
        ADD 1, R2   ; R2 = left + 1
```

```
        ; If left is beyond the end of the heap, then we're done
        CMP (heap_count), R1
        BLE minheap_delete_loop_done ; b if (heap_count) <= R1=left

        ; If left(R1) == heap count, then we can jump right to the
        ;  final swap
        BEQ minheap_delete_loop_done

        ; Load left value
        LD (heap_base), R7
        LD R1, R8
        SHL 3, R8
        ADD R8, R7
        MAR R7, R5

        ; Load right value
        ADD 8, R7
        MAR R7, R6

        ; Compare left to right, we want to swap up the smallest value
        CMP R5, R6  ; left ?? right
        BLE minheap_delete_swap ; left <= right

        ; Else, right is smaller
        LD R2, R1   ; Load position and value
        LD R6, R5

    minheap_delete_swap:
        ; Assume postition is in R1, value is in R5

        ; Fetch the data
        LD (heap_base), R7
        LD R1, R8   ; Load location of m[l|r] + 4
        SHL 3, R8
        ADD 4, R8
        ADD R7, R8
        MAR R8, R10 ; Read data

        LD R0, R8   ; Load location of m[p]
        SHL 3, R8
        ADD R8, R7
        MAW R7, R5  ; Write key
        ADD 4, R7
        MAW R7, R10 ; Write data

        ; Store the new position
        LD R1, R0
        JMP minheap_delete_loop

minheap_delete_loop_done:
    ; Load m[p] <= m[heap_count]
```

```
    ; Load m[heap_count] and data
    LD (heap_base), R6
    LD (heap_count), R7
    SHL 3, R7
    ADD R7, R6
    MAR R6, R8
    ADD 4, R6
    MAR R6, R9

    ; Write it to m[p]
    LD (heap_base), R6
    LD R0, R7
    SHL 3, R7
    ADD R7, R6
    MAW R6, R8
    ADD 4, R6
    MAW R6, R9

    LD (heap_count), R6
    SUB 1, R6
    ST (heap_count), R6

    JMP command_wait_done


; Empty the heap
command_minheap_empty:
    LD 0, R0
    ST (heap_count), R0
    JMP command_wait_done


; Fire an IRQ at the Nios, and wait for it to ack it
command_wait_done:

    SLC 0
    IRQ 1
    command_wait_done_wait:
        SLR 0, R0
        CMP 0, R0
        BEQ command_wait_done_wait

    IRQ 0
    JMP R15

.long commands  0xFF000001
.long __c1  #command_reset
.long __c2  0x00000100
.long __c3  #command_minheap_base
.long __c4  0x00000101
.long __c5  #command_minheap_insert
```

```
.long __c6  0x00000102
.long __c7  #command_minheap_delete
.long __c8  0x00000103
.long __c9  #command_minheap_empty
.long __c_last  0x00000000
```

Appendix B

Minheap Results

This chapter contains the raw minheap test results. Each result is obtained by filling and emptying a minheap with the specified number of entries 5 times. Table B.1 gives the result for a Nios only implementation of the heap. In this test, no FAST CPUs are activated and the Nios processor manages the entire heap. Tables B.2, B.3, and B.4 give the result for a 1–, 2–, and 3–FAST CPU system respectively. In these tests, the Nios processor is instructing the FAST CPUs to insert or delete items from the heap, but is performing no heap management itself. The far righthand column of each table is computed to be the average time to insert an item into the tree and then delete it.

Table B.1: Minheap Results for a Nios Implementation

Entries	TotalClock Cycles	SRAM Cycles		FAST CPU Read Cycles			μs per ins+del
		Read	Write	CPU 1	CPU 2	CPU 3	@33 MHz[1]
1	10192	5942	325	0	0	0	61.77
50	678395	398510	18353	0	0	0	82.23
100	1443614	849738	37754	0	0	0	87.49
250	3876988	2288918	97572	0	0	0	93.99
500	8194648	4846813	200471	0	0	0	99.33
1000	17236251	10211760	411003	0	0	0	104.46
1500	26624215	15789445	625512	0	0	0	107.57
2000	36164717	21460020	842082	0	0	0	109.59
2500	45928638	27267090	1061142	0	0	0	111.34
3000	55795867	33138444	1281377	0	0	0	112.72
3500	65739460	39055691	1502671	0	0	0	113.83
4000	75714907	44993534	1724273	0	0	0	114.72
4500	85854123	51030615	1947798	0	0	0	115.63
5000	96082792	57121920	2172317	0	0	0	116.46
5500	106385330	63258338	2398058	0	0	0	117.23
6000	116694405	69400149	2623620	0	0	0	117.87
6500	127044921	75566946	2849716	0	0	0	118.46
7000	137405439	81740288	3075638	0	0	0	118.97
7500	147764718	87913767	3301467	0	0	0	119.41
8000	158215066	94141823	3529207	0	0	0	119.86
8500	168704562	100394539	3756569	0	0	0	120.29
9000	179344195	106736691	3986237	0	0	0	120.77
9500	189998744	113088430	4215939	0	0	0	121.21
10000	200644581	119435784	4445386	0	0	0	121.60

[1]This is computed by: $\frac{\text{Total Cycles}}{5\times\text{Entries}} \times \frac{1}{33\text{ MHz}}$

Table B.2: Minheap Results for a 1–FAST CPU Implementation

Entries	TotalClock Cycles	SRAM Cycles Read	SRAM Cycles Write	FAST CPU Read Cycles CPU 1	FAST CPU Read Cycles CPU 2	FAST CPU Read Cycles CPU 3	μs per ins+del @33 MHz[1]
1	10207	5372	175	20	0	0	61.86
50	744240	385497	8750	5750	0	0	90.21
100	1609661	831644	17500	13910	0	0	97.56
250	4422178	2278018	43750	42470	0	0	107.20
500	9474539	4871268	87500	97330	0	0	114.84
1000	20201276	10368659	175000	219540	0	0	122.43
1500	31556490	16179745	262500	353950	0	0	127.50
2000	42913993	21993520	350000	488950	0	0	130.04
2500	54873992	28108027	437500	635260	0	0	133.03
3000	66894174	34251517	525000	782760	0	0	135.14
3500	78939450	40407411	612500	930260	0	0	136.69
4000	90979722	46560544	700000	1077760	0	0	137.85
4500	103487623	52948789	787500	1235370	0	0	139.38
5000	116173803	59424018	875000	1395370	0	0	140.82
5500	128753536	65850980	962500	1555370	0	0	141.88
6000	141375009	72296079	1050000	1715370	0	0	142.80
6500	154052091	78767373	1137500	1875370	0	0	143.64
7000	166717151	85232992	1225000	2035370	0	0	144.34
7500	179338150	91678578	1312500	2195370	0	0	144.92
8000	191980758	98133989	1400000	2355370	0	0	145.44
8500	205018155	104786931	1487500	2523080	0	0	146.18
9000	218271652	111549258	1575000	2695580	0	0	146.98
9500	231614120	118352863	1662500	2868080	0	0	147.76
10000	244880487	125121551	1750000	3040580	0	0	148.41

[1]This is computed by: $\frac{\text{Total Cycles}}{5\times\text{Entries}} \times \frac{1}{33\text{ MHz}}$

Table B.3: Minheap Results for a 2–FAST CPU Implementation

Entries	TotalClock Cycles	SRAM Cycles		FAST CPU Read Cycles			μs per ins+del
		Read	Write	CPU 1	CPU 2	CPU 3	@33 MHz[1]
1	12973	6913	345	20	20	0	78.62
50	873709	458516	17250	5750	5750	0	105.90
100	1873782	980094	34500	13910	13910	0	113.56
250	5073895	2645340	86250	42470	42470	0	123.00
500	10766364	5600258	172500	97330	97330	0	130.50
1000	22819960	11842441	345000	219540	219540	0	138.30
1500	35482558	18390068	517500	353950	353950	0	143.36
2000	48152462	24942541	690000	488950	488950	0	145.92
2500	61417975	31791768	862500	635260	635260	0	148.89
3000	74710171	38655183	1035000	782760	782760	0	150.93
3500	88055062	45542309	1207500	930260	930260	0	152.48
4000	101313304	52391208	1380000	1077760	1077760	0	153.51
4500	115138927	59521832	1552500	1235370	1235370	0	155.07
5000	129099793	66717606	1725000	1395370	1395370	0	156.48
5500	143056776	73912241	1897500	1555370	1555370	0	157.64
6000	156946123	81078474	2070000	1715370	1715370	0	158.53
6500	171044791	88340986	2242500	1875370	1875370	0	159.48
7000	184835085	95458183	2415000	2035370	2035370	0	160.03
7500	198944417	102724149	2587500	2195370	2195370	0	160.76
8000	212772507	109862150	2760000	2355370	2355370	0	161.19
8500	227204377	117289989	2932500	2523080	2523080	0	162.00
9000	241851074	124822309	3105000	2695580	2695580	0	162.86
9500	256340526	132293022	3277500	2868080	2868080	0	163.53
10000	271069870	139861206	3450000	3040580	3040580	0	164.28

[1]This is computed by: $\frac{\text{Total Cycles}}{5\times\text{Entries}} \times \frac{1}{33\text{ MHz}}$

Table B.4: Minheap Results for a 3–FAST CPU Implementation

Entries	TotalClock Cycles	SRAM Cycles		FAST CPU Read Cycles			μs per ins+del
		Read	Write	CPU 1	CPU 2	CPU 3	@33 MHz[1]
1	15484	8342	515	20	20	20	93.84
50	1004503	532713	25750	5750	5750	5750	121.76
100	2132287	1127326	51500	13910	13910	13910	129.23
250	5721448	3013711	128750	42470	42470	42470	138.70
500	12066954	6339931	257500	97330	97330	97330	146.27
1000	25409983	13314001	515000	219540	219540	219540	154.00
1500	39343991	20588110	772500	353950	353950	353950	158.97
2000	53303777	27873225	1030000	488950	488950	488950	161.53
2500	67849690	35453341	1287500	635260	635260	635260	164.48
3000	82500083	43082425	1545000	782760	782760	782760	166.67
3500	97063295	50676490	1802500	930260	930260	930260	168.07
4000	111686050	58295970	2060000	1077760	1077760	1077760	169.22
4500	126844127	66177993	2317500	1235370	1235370	1235370	170.83
5000	142063855	74096215	2575000	1395370	1395370	1395370	172.20
5500	157332139	82037625	2832500	1555370	1555370	1555370	173.37
6000	172613748	89983893	3090000	1715370	1715370	1715370	174.36
6500	187861726	97915799	3347500	1875370	1875370	1875370	175.16
7000	203081046	105837125	3605000	2035370	2035370	2035370	175.83
7500	218326756	113764184	3862500	2195370	2195370	2195370	176.43
8000	233591312	121706041	4120000	2355370	2355370	2355370	176.96
8500	249274280	129844740	4377500	2523080	2523080	2523080	177.74
9000	265170336	138095491	4635000	2695580	2695580	2695580	178.57
9500	281074863	146344872	4892500	2868080	2868080	2868080	179.31
10000	296927993	154574163	5150000	3040580	3040580	3040580	179.96

Appendix C

Nios C Code

C.1 FAST CPU API and Interface Driver

This is the header file and source code for the FAST CPU API and driver.

---

fastcpu.h

```
/* FAST CPU API header file
 *  David Grant, 2004 */
#ifndef _FASTCPU_H
#define _FASTCPU_H

struct _fastcpu {
    unsigned long base;
    unsigned long irq;
    unsigned long busy;
    volatile unsigned long *loc[4];
    volatile unsigned long done;
} ;


int fastcpu_init(unsigned long base, unsigned long irq);
void fastcpu_fini(struct _fastcpu *cpu);
void fastcpu_load(struct _fastcpu *cpu, unsigned long *program,
                unsigned long len);
struct _fastcpu *fastcpu_alloc(void);
void fastcpu_free(struct _fastcpu *cpu);
void fastcpu_reset(struct _fastcpu *cpu);
```

```
int fastcpu_num_cpus(void);

#endif
```

---

fastcpu.c

```
/* FAST CPU API and hadware interface
 *  David Grant, 2004 */
#include "nios.h"
#include "fastcpu.h"

static struct _fastcpu fastcpu[2];
static unsigned long fastcpu_count=0;

static void fastcpu_isr(int cpu)
{
    struct _fastcpu *f= (struct _fastcpu *)cpu;

    /* Instruct the device to turn off the interrupt */
    *(f->loc[0]) = 1;

/*  if(done == 1) {
        printf("Spurrious interrupt!  called when done==1\n");
        printf("n=0x%08x, requested_n=0x%08lx\n", n+1, requested_n);
    }
*/

    /* Singal that the interrupt has occured to the running code */
    f->done = 1;
}


int fastcpu_probe(unsigned long base)
{
    volatile unsigned long *loc0 = (unsigned long *)base;

    /* Check the device for 'PADS' */
    if(*loc0  != 0x70616473) {
        return 0;
    }
    return 1;
}

int fastcpu_init(unsigned long base, unsigned long irq)
{
    int x, y;
    struct _fastcpu *f;
    unsigned long data;
    volatile unsigned long *loc1 = (unsigned long *)(base+4);
    fastcpu_count = 0;
```

```
    /* Check the device */
    for(x=0;x<8;x++) {
        unsigned long probe_base = base + (0x100 * x);

        if(!fastcpu_probe(probe_base)) continue;

        data = *loc1;
        printf("fastcpu%d: PADS uP, @0x%4x irq %d, "
                "loader firmware v%02x.%02x.%02x.%02x\n",
            fastcpu_count,
            probe_base, irq+x,
            (data & 0xff000000) >> 24, (data&0x00ff0000) >> 16,
            (data & 0x0000ff00) >> 8, data & 0x000000ff);

        /* Setup the structure */
        f = &fastcpu[fastcpu_count];
        f->base = probe_base;
        f->irq = irq + x;
        f->busy = 0;
        f->done = 0;
        for(y=0;y<4;y++)
            f->loc[y] = (unsigned long *)(probe_base + (y*4));


        nr_installuserisr(f->irq, fastcpu_isr,(unsigned long)f);
        fastcpu_count++;
    }
    return 1;
}

void fastcpu_fini(struct _fastcpu *cpu)
{
    nr_installuserisr(cpu->irq, 0, 0);
}



void fastcpu_program(struct _fastcpu *cpu, unsigned long *program,
                unsigned long len)
{
    /* write the length first, the cpu spins waiting for addr, then
     * assumes that all other data is available*/
    *(cpu->loc[1]) = (unsigned long)&program[0x20];
    *(cpu->loc[2]) = len - 0x20;
    *(cpu->loc[0]) = 0xFF000000;

    while(cpu->done == 0) ;
}

struct _fastcpu *fastcpu_alloc(void)
{
```

```
    unsigned long x;
    for(x=0;x<fastcpu_count;x++) {
        if(fastcpu[x].busy == 0) {
            fastcpu[x].busy=1;
            return &fastcpu[x];
        }
    }
    return NULL;
}

void fastcpu_free(struct _fastcpu *cpu)
{
    *cpu->loc[0] = 0xFFFFFFFF;
    cpu->busy=0;
}

/* reset the cpu, all client programs must accept this command */
void fastcpu_reset(struct _fastcpu *cpu)
{
    *cpu->loc[0] = 0xFFFFFFFF;
}


int fastcpu_num_cpus(void)
{
    return fastcpu_count;
}
```

C.2 Nios Minheap Code using the FAST CPUs

This is the code for the minheap tests run on the configurable system. This code uses the FAST CPUs (via. the FAST CPU API) to implement the minheap.

---

minheap.c

```
/* Minheap code that uses the FAST CPU
 * David Grant, 2003,2004 */
#include "nios.h"

#include "minheap_fastcpu.h"
#include "fastcpu.h"

#include "bus_util.h"
```

```
volatile unsigned long n;

//unsigned long *heap;

/*************************************************************************
 * Pseudo-random number generator */
unsigned long seed=56254254, A=9301, C=49297, M=233280;
unsigned short prand()
{
    unsigned long t;
    seed = (seed * A + C) % M;
    return (unsigned short)(seed & 0xffff);
}


/*************************************************************************
 * Minimum/Average/Maximum routines, to compute averages of results and
 * check to make sure no min/max data is too far off the average */
struct _mam {
    unsigned long count;
    unsigned long min;
    unsigned long max;
    unsigned long total;
};

void mam_clear(struct _mam *m)
{
    m->count = 0;
    m->total = 0;
    m->min = 0xffffffff;
    m->max = 0;
}

void mam_add(struct _mam *m, unsigned long value)
{
    if(value > m->max) m->max = value;
    if(value < m->min) m->min = value;
    m->count++;
    m->total += value;
}

void mam_print(struct _mam *m)
{
    printf("(%lu/%lu/%lu)", m->min, (m->total/m->count), m->max);
}


/*************************************************************************
 * Minheap interface with the FAST CPUs */

/* set minheap memory base , function 100*/
void minheap_base(struct _fastcpu *cpu, void *ptr)
```

```
{
    cpu->done=0;
    *(cpu->loc[1]) = (unsigned long )ptr;
    *(cpu->loc[0]) = 0x00000100;
    while(cpu->done==0);
}

/* Minheap insert, function 101 */
void minheap_insert_no_wait(struct _fastcpu *cpu, int k, int x)
{
    cpu->done=0;
    *cpu->loc[1] = k;
    *cpu->loc[2] = x;
    *cpu->loc[0] = 0x00000101;
}

/* Delete, function 102 */
void minheap_delete_no_wait(struct _fastcpu *cpu, int *k, int *x)
{
    cpu->done=0;
    *cpu->loc[1] = 1;
    *cpu->loc[0] = 0x00000102;
}
/* result from the delete */
void minheap_delete_get_result(struct _fastcpu *cpu, int *k, int *x)
{
    if(k) *k = *cpu->loc[0];
    if(x) *x = *cpu->loc[1];
}
/* wait for an interrupt */
void minheap_wait_complete(struct _fastcpu *cpu)
{
    while(cpu->done==0);
}

/* blocking insert using above routines */
void minheap_insert(struct _fastcpu *cpu, int k, int x)
{
    minheap_insert_no_wait(cpu, k, x);
    minheap_wait_complete(cpu);
}

/* Blocking delete, using above functions */
void minheap_delete(struct _fastcpu *cpu, int *k, int *x)
{
    minheap_delete_no_wait(cpu, k, x);
    minheap_wait_complete(cpu);
    minheap_delete_get_result(cpu, k, x);
}
```

```
/* Minheap count, available any time */
int minheap_count(struct _fastcpu *cpu)
{
    return *cpu->loc[3];
}

/* Minheap empty, function 103 */
void minheap_empty(struct _fastcpu *cpu)
{
    cpu->done=0;
    *cpu->loc[1] = 3;
    *cpu->loc[0] = 0x00000103;

    while(cpu->done==0);
}

/* forced reset */
void minheap_reset(struct _fastcpu *cpu)
{
    *cpu->loc[0] = 0xff000000;
}


/*****************************************************************************
 * Ensure the correct functionality of a minheap on a processor */
int functionality_test(struct _fastcpu *cpu, int max)
{
    int x, i;
    long *heap;
    int ret = 0;

    printf("functionality test: (programming cpu)");
    fastcpu_program(cpu, minheap_fastcpu_program_code,
                minheap_fastcpu_program_code_count);

    heap = (long *)malloc(0x20000);
    if(!heap) printf(" MALLOC FAILED\n");
    printf("(set to 0x%08lx)  ", (long)heap);
    minheap_base(cpu, heap);

    for(x=0;x<16;x++) {
        int k=0, d=0;
        for(i=0;i<max;i++) {
            minheap_insert(cpu, max - i - 1 + x, i);
        }
        if(minheap_count(cpu) != max) {
            printf("minheap_count=%d != %d\n", minheap_count(cpu), max);
            goto functionality_fail;
        }

//      heap_print(max);
```

```
        for(i=0;i<max;i++) {
            minheap_delete(cpu, &k, &d);
//          heap_print(max-i);
            if(k != i + x) {
                printf("value at %d is %d != %d\n", i, k, i+x);
                goto functionality_fail;
            }
            if(d != max - i - 1) {
                printf("data at %d:%d is %d != %d\n", x,i, d, max - i - 1);
                goto functionality_fail;
            }

        }
        if(minheap_count(cpu) != 0) {
            printf("minheap_count=%d, not empty(0)\n",
                    minheap_count(cpu));
            goto functionality_fail;
        }
    }
    printf("OK\n");
    ret = 1;
    goto done;

functionality_fail:
    printf("FAIL\n");
    ret = 0;

done:
    free(heap);
    return ret;
}

/*****************************************************************************
 * Test from Bill's PhD thesis, slightly modified */
void bill_test(int cpus, int entries)
{
    int i;
    long t;
    int x, y;
    struct _fastcpu *cpu[3];
    long *heap[3];
    struct _mam c_mam, s_mam[8];

    for(x=0;x<3;x++) {
        cpu[x] = fastcpu_alloc();
        /* program it */
        fastcpu_program(cpu[x], minheap_fastcpu_program_code,
                minheap_fastcpu_program_code_count);

        heap[x] = (long *)malloc(0x18000);
```

```
        if(!heap) printf(" MALLOC %d FAILED\n", x);
        minheap_base(cpu[x], heap[x]);
    }
    bus_util_reset_and_start();
    bus_util_stop();
    bus_util_count(&x, s_mam);
    printf("CYCLES=%d\n", x);


    for(x=0;x<5;x++) {
        unsigned long usage;
        unsigned long c, s[8];

        mam_clear(&c_mam);
        for(y=0;y<8;y++) mam_clear(&s_mam[y]);

        t = 0;
        if(cpus == 1) {
            bus_util_reset_and_start();
            for(i=0;i<5;i++) {
                int j, d, k;
                int r;
                for(j=0;j<entries;j++) {
                    r = prand();
                    minheap_insert_no_wait(cpu[0], r, j);
                    minheap_wait_complete(cpu[0]);
                }
                for(j=0;j<entries;j++) {
                    minheap_delete_no_wait(cpu[0], &k, &d);
                    minheap_wait_complete(cpu[0]);
                    minheap_delete_get_result(cpu[0], &k, &d);
                }
            }
            bus_util_stop();
        } else if(cpus==2) {
            bus_util_reset_and_start();
            for(i=0;i<5;i++) {
                int j, d, k;
                int r;
                for(j=0;j<entries;j++) {
                    r = prand();
                    minheap_insert_no_wait(cpu[0], r, j);
                    minheap_insert_no_wait(cpu[2], r, j);
                    minheap_wait_complete(cpu[0]);
                    minheap_wait_complete(cpu[2]);
                }
                for(j=0;j<entries;j++) {
                    minheap_delete_no_wait(cpu[0], &k, &d);
                    minheap_delete_no_wait(cpu[2], &k, &d);
                    minheap_wait_complete(cpu[0]);
                    minheap_delete_get_result(cpu[0], &k, &d);
                    minheap_wait_complete(cpu[2]);
```

```
                minheap_delete_get_result(cpu[2], &k, &d);
            }
        }
        bus_util_stop();
    } else {
        bus_util_reset_and_start();
        for(i=0;i<5;i++) {
            int j, d, k;
            int r;
            for(j=0;j<entries;j++) {
                r = prand();
                minheap_insert_no_wait(cpu[0], r, j);
                minheap_insert_no_wait(cpu[1], r, j);
                minheap_insert_no_wait(cpu[2], r, j);
                minheap_wait_complete(cpu[0]);
                minheap_wait_complete(cpu[1]);
                minheap_wait_complete(cpu[2]);
            }
            for(j=0;j<entries;j++) {
                minheap_delete_no_wait(cpu[0], &k, &d);
                minheap_delete_no_wait(cpu[1], &k, &d);
                minheap_delete_no_wait(cpu[2], &k, &d);
                minheap_wait_complete(cpu[0]);
                minheap_delete_get_result(cpu[0], &k, &d);
                minheap_wait_complete(cpu[1]);
                minheap_delete_get_result(cpu[1], &k, &d);
                minheap_wait_complete(cpu[2]);
                minheap_delete_get_result(cpu[2], &k, &d);
            }
        }
        bus_util_stop();
    }

    bus_util_count(&c, s);

    mam_add(&c_mam, c);
    for(y=0;y<8;y++) mam_add(&s_mam[y], s[y]);


    minheap_empty(cpu[0]);
    minheap_empty(cpu[1]);
    minheap_empty(cpu[2]);
}
/* num, total cycles, sel, rd, wr, f0cpu rd, 1,2,any,all */
printf("%d, %ld, "
        "%lu, %lu, %lu, %lu, %lu, %lu, %lu, %lu, %lu ",
            cpus, entries,
        c_mam.total/c_mam.count,
        s_mam[5].total/s_mam[5].count,
        s_mam[6].total/s_mam[6].count,
        s_mam[7].total/s_mam[7].count,
        s_mam[0].total/s_mam[0].count,
```

```
            s_mam[1].total/s_mam[1].count,
            s_mam[2].total/s_mam[2].count,
            s_mam[3].total/s_mam[3].count,
            s_mam[4].total/s_mam[4].count);

    printf("\n");

    for(x=0;x<3;x++) {
        free(heap[x]);
        fastcpu_free(cpu[x]);
    }
}

/* call for bill's sequence tests */
void sequence_test(void)
{
    unsigned long tries[] = { 1, 50, 100, 250, 500, 1000, 2500, 5000,
                10000, 2500, 0 };
    int x;
    unsigned long us;

    printf("cpus, entries, ttl_cy, sel_cy, rd_cy, wr_cy, ");
    printf("f0_rd_cy, f1_rd_cy, f2_rd_cy, fany_rd_cy, fall_rd_cy");
    printf("\n");

    for(x=0;;x++) {
        if(tries[x] == 0) break;
        bill_test(1, tries[x]);
        if(x>0) {
            if(tries[x-1] == 500) {
                tries[x] += 500;
                if(tries[x] > 10000) tries[x]=0;
                x--;
            }
        }
    }
    tries[5]=1000;
    for(x=0;;x++) {
        if(tries[x] == 0) break;
        bill_test(2, tries[x]);
        if(x>0) {
            if(tries[x-1] == 500) {
                tries[x] += 500;
                if(tries[x] > 10000) tries[x]=0;
                x--;
            }
        }
    }
    tries[5]=1000;
    for(x=0;;x++) {
        if(tries[x] == 0) break;
        bill_test(3, tries[x]);
```

```
        if(x>0) {
            if(tries[x-1] == 500) {
                tries[x] += 500;
                if(tries[x] > 10000) tries[x]=0;
                x--;
            }
        }
    }
}

/*************************************************************************
 * rand() time test, so we know how many cycles it takes */
int prand_test(void)
{
    unsigned long x, y, r;
    unsigned long tries[] = { 1000, 10000, 50000, 100000, 500000, 0 };
    struct _mam c_mam;

    mam_clear(&c_mam);

    printf("prand() time test: ");
    for(y=0;; y++) {
        unsigned long s[8],c;
        if(tries[y] == 0) break;
        bus_util_reset_and_start();
        for(x=0;x<tries[y];x++) r = prand();
        bus_util_stop();
        bus_util_count(&c, s);
        mam_add(&c_mam, c);

        printf("(%ldcy)  ", c_mam.total/c_mam.count);
    }
    printf("\n");
    return 1;
}


/*************************************************************************
 * main routine */

/* We can only synthesize 3 cpus max */
#define FAST_CPUS_MAX 3
/*#define FAST_CPUS_MAX 10*/

int main(void)
{
    int x,y,i;
    unsigned long t, tl;
    long delta;

    struct _cpu_device {
        struct _fastcpu *cpu;
```

```
    } cpu[FAST_CPUS_MAX];

    struct _bus_util_data bus_util_data;


    /* Init all CPUs, starting with the first one, and checking at
     * offsets + 0x100 for subsequent ones. */
    fastcpu_init(na_fastcpu_s0_base, na_fastcpu_s0_irq);
    printf("total of %d FASTCPUs activated.\n", fastcpu_num_cpus());

    bus_util_init(na_bus_utillization_base, &bus_util_data);

    printf("timer nasys_clock_freq = %lu (0x%08lx)\n",
            nasys_clock_freq, nasys_clock_freq);
    printf("timer nasys_clock_freq_1000 = %lu (0x%08lx)\n",
            nasys_clock_freq_1000, nasys_clock_freq_1000);

    for(x=0; x<fastcpu_num_cpus(); x++) {
        cpu[x].cpu = fastcpu_alloc();
        if(cpu[x].cpu == NULL) {
            printf("fastcpu_alloc returned NULL!\n");
        }
    }

    for(x=0; x<fastcpu_num_cpus(); x++) {
        functionality_test(cpu[x].cpu, 1000);
        fastcpu_free(cpu[x].cpu);
    }

    prand_test();
    sequence_test();

/*  nr_installuserisr(na_fastcpu_s0_irq,0,0);*/

    printf("done.\n\004");
}
```

Bibliography

[Alt97] Altera Programmable Hardware Development Program. World Wide Web Document, January 1997. `http://www.altera.com/html/programs/phd.html`.

[Alt99] Altera Corporation. Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices. Application Note A-AN-116-01, Altera Corporation, San Jose, California, August 1999.

[Alt02a] Altera Corporation. Custom Instructions for the Nios Embedded Processor v1.2. Application Note 188, Altera Corporation, San Jose, California, September 2002.

[Alt02b] Altera Corporation. Cyclone Product Backgrounder. Product Backgrounder, Altera Corporation, San Jose, California, November 2002.

[Alt03a] Altera Corporation. *Avalon Bus Specification - Reference Manual*. San Jose, California, July 2003.

[Alt03b] Altera Corporation. *Nios Embedded Processor 32-Bit Programmer's Reference Manual*. San Jose, California, January 2003.

[Alt03c] Altera Corporation. Nios Embedded Processor Development Board Data Sheet. Data Sheet DS-NIOSDEVBD-2.1, San Jose, California, July 2003.

[Alt03d] Altera Corporation. Nios Software Development Tutorial. Tutorial, Altera Corporation, San Jose, California, July 2003.

[Alt03e] Altera Corporation. SOPC Builder Data Sheet. Data Sheet DS-SOPC-2.0, Altera Corporation, San Jose, California, January 2003.

[Alt04] Altera Corporation. *Section I. Stratix II Device Family Data Sheet, v1.0*. San Jose, California, February 2004.

[Bis03] William D. Bishop. Configurable Computing for Mainstream Software Applications. Ph.D. Thesis, Parallel and Distributed Systems Group, University of Waterloo, Waterloo, Ontario, Canada, May 2003.

[CH00] K. Compton and S. Hauck. An Introduction to Reconfigurable Computing. *Invited Paper, IEEE Computer*, April 2000.

[CH02] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[Cha94] Pak K. Chan. A Field-Programmable Prototyping Board: XC4000 BORG User's Guide. Technical Report UCSC-CRL-94-18, Board of Studies in Computer Engineering, University of California, Santa Cruz, Santa Cruz, California, April 1994.

[CN96] J.M.P. Cardoso and H.C. Neto. A Co–Synthesis Environment for Embedding Digital Systems in a Sea–of–Gates IC. In *Proceedings of the Eleventh Conference on Design of Integrated Circuits and Systems (DSCIS'96)*, pages 411–416, Sitges, Barcelona, November 1996.

[CR93] Steven A. Cuccaro and Craig F. Reese. The CM-2X: A Hybrid CM-2/Xilinx Prototype. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 121–130, April 1993.

[Cyg99] Cygnus. GNUpro Tools for Embedded Systems. Technical Report version 99r1, Sunnyvale, California, 1999.

[De 94] G. De Micheli. Computer–Aided Hardware–Software Codesign. *IEEE Micro*, 14(4):6–10, August 1994.

[DG97] G. De Micheli and Rajesh K. Gupta. Hardware/Software Co–Design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.

[DW99] O. Diessel and G. Wigley. Opportunities for operating systems research in reconfigurable computing, 1999.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, 12:747–755, December 1963.

[Est60] G. Estrin. Organization of Computer Systems – The Fixed Plus Variable Structure Computer. *Proc. Western Joint Computer Conference*, pages 33–40, 1960.

[GKC+94] David Galloway, David Karchmer, Paul Chow, David Lewis, and Jonathan Rose. The Transmogrifier: The University of Toronto Field-Programmable System. Technical Report CSRI-306, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, June 1994.

[Guc00] Steve Guccione. List of FPGA–Based Computing Machines. World Wide Web Document, August 2000. `http://www.io.com/~guccione/HW_list.html`.

[Hau98] S. Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):349–365, April 1998.

[HF95] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison–Wesley Publishing Company, Reading, Massachussetts, January 1995.

[HS98] Samuel Holstrom and Kiasa Sere. Reconfigurable Hardware – A Case Study in Codesign. Technical Report No 175, Turku Centre for Computer Science, Turku, Finland, May 1998.

[Int97] Intel Corporation. *Intel MultiProcessor Specification v1.4*. Mt. Prospect, Illinois, May 1997.

[Mic02] Microtronix. *Stratix EP1S25 Development Kit*. London, Ontario, December 2002.

[Sep02] K. Seppanen. Massively Parallel Priority Queues for High–Speed Switches and Routers. *Advances in Communications and Software Technologies*, pages 71–76, 2002.

[WK01] G. Wigley and D. Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proceedings of the Ninth Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01)*, Napa Valley, California, 2001.

[Xil02] Xilinx, Inc. *Virtex-II Pro Prototype Platform User Guide*. San Jose, California, October 2002.

[Xil04a] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. San Jose, California, June 2004.

[Xil04b] Xilinx, Inc. Xilinx Virtex–4 Revolutionizes Platform FPGAs. World Wide Web Document (Press Release), January 2004. `http://www.xilinx.com/company/press/kits/v4_arch/v4_finalwhitepaper4.pdf`.