Cache Coherency for Symmetric Multiprocessor Systems on Programmable Chips

by

Austin Hung

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

Abstract

Rapid progress in the area of Field-Programmable Gate Arrays (FPGAs) has led to the availability of softcore processors that are simple to use, and can enable the development of a fully working system in minutes. This has lead to the enormous popularity of System-On-Programmable-Chip (SOPC) computing platforms. These softcore processors, while relatively simple compared to their leading-edge hardcore counterparts, are often designed with a number of advanced performance-enhancing features, such as instruction and data caches. Moreover, they are designed to be used in a uniprocessor or uncoupled multiprocessor architecture, and not in a tightly-coupled multiprocessing architecture. As a result, traditional cache-coherency protocols are not suitable for use with such systems. This thesis describes a system for enforcing cache coherency on symmetric multiprocessing (SMP) systems using softcore processors. A hybrid protocol that incorporates hardware and software to enforce cache coherency is presented.

Acknowledgements

This work was supported in part by the Science and Engineering Research Council (SERC) of Canada Discovery Grant 203763-03 and by an Ontario Graduate Scholarship (OGS). I would like to gratefully acknowledge the support provided by Altera Corporation, in the form of Nios development boards and related software. In particular, my thanks go to Dr. Andrew Kennings and Dr. William Bishop, without whom this work would not be. Finally, my thanks also go out to, in no particular order, my family, Julie, and the Contraplex Systems group.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis investigates the use of multiple processors in system-on-chip (SOC) systems when targetting programmable logic devices (PLDs), specifically field-programmable gate arrays (FPGAs). The goal of the research described in this thesis is to qualify and analyze the problems associated with implementing symmetric multiprocessor (SMP) systems on FPGAs. In particular, cache coherency is a major focus. Proof-of-concept and second-generation implementations are presented.

1.1 Motivation

Recent advances in FPGA technologies have resulted in programmable devices with significantly improved features and capabilities, including density, speed and functionality. Some of these improvements include the following:

- Improved Performance - A state-of-the-art FPGA can be clocked at 250 MHz, with an internal frequency of up to 500 MHz.

- Improved Capacity - A recently released FPGA family supports designs with up to 2.2 million application-specific integrate circuit (ASIC) logic gates and 9,383,040 on-chip RAM bits.

- Advanced Features - These features, including programmable phase-locked loops (PLLs), digital signal processing (DSP) blocks, and diverse I/O capabilities, are embedded within the fabric of the FPGA.

- User-Friendly CAD Tools - New releases of vendor computer-aided design (CAD) tools, hardware description languages (HDLs), and graphical user interfaces (GUIs) have simplified the task of building entire systems of complex, reuseable, and custom digital circuits.

These advanced capabilities, coupled with configurability and falling costs, have rapidly increased the popularity of FPGAs. The ability to generate custom systems for each application and yet reuse the same hardware device is a compelling argument for the use of FPGAs. This is true not just in research, but also in industry, which takes advantage of FPGAs not only to lower costs, but also to gain the ability to easily and cheaply add fixes and new features to products without requiring a product recall.

1.2 Field-Programmable Gate Arrays

Programmable logic represents the ultimate form of flexible hardware. Each PLD is a semiconductor that consists of memory and logic elements (LEs). The memory is configured with a hardware design, which defines temporary physical connections that form complex digital circuits. Since the memory is writeable, the PLD can be configured with different designs over and over again. FPGAs, in particular, are a subset of PLDs that can be programmed in the field.

That is, reconfiguring an FPGA is not limited to the time and location of manufacturing or initial programming. Field programmability even allows for remote programming facilities.

An FPGA is based on static random-access memory (SRAM) technology, which is high performance, but also high cost (six transistors are required to implement each SRAM bit). SRAM is volatile, and thus retains its contents only as long as it is powered. Some of the SRAM is designed to be dedicated on-chip memory, but most of the memory serves to configure logic elements and to define the interconnections between various logic elements. The configured logic elements and their interconnections are what carries out the desired functionality, such as state machines or arithmetic units.

Each logic element, though different for each FPGA vendor, typically contains a programmable four-input look-up table and a one-bit register. This table allows each logic element to implement any four-input function. The output of the logic element is selectable between the table or the register. Each vendor also adds other custom hardware to more efficiently implement common functionality, such as adders[1].

1.3 Custom Logic Versus Programmable Logic

Traditionally, due to the nature of custom logic versus programmable logic, FPGAs have been cost-effective only in small volumes. ASICs incur a large initial capital cost, or non-recurring engineering (NRE) cost, for a mask set, prototype wafers and respins. The NRE effectively eliminates custom logic as an option for low-volume applications. On the other hand a small unit price (a fraction of the cost of a PLD) combined with the amortization of capital costs over a large number of units, make ASICs ideal for high-volume applications. Additionally, until recently,

[1]The variety in logic elements (sometimes labelled logic cells or configurable logic blocks, depending on the vendor) often leads to varying methods for measuring the capacity of an FPGA [3] [19].

even moderately large or high-performance designs would not fit into an FPGA.

Today, it is possible to implement multi-million gate designs in a single programmable device that take advantage of both readily available third-party intellectual property (IP) as well as hard IP included within the FPGA itself. It has become relatively simple to build a large and highly complex system-on-programmable-chip (SOPC) rather that developing an ASIC. The lengthy design cycles, expensive software CAD tools and NRE costs associated with ASIC design can be avoided. Additionally, programmable logic allows for an unprecedented amount of flexibility since a single device can be reprogrammed to serve many different tasks, while an ASIC is only designed to perform one task.

1.4 Multiprocessors-On-Programmable-Chips

The enhanced programmability and the larger capacities of modern FPGAs have made it possible to create MultiProcessors-On-Programmable-Chip (MPOPC) systems that include either third-party or vendor-provided proprietary softcore microprocessors[2]. Examples of vendor-provided softcore processors include the Xilinx MicroBlaze [34] and the Altera Nios [10] softcore processors. The configurability of softcore processors make them excellent candidates for MPOPC systems.

Traditionally, multiprocessor systems have been implemented using discrete processors with traces on a printed circuit board (PCB) serving as the physical interconnect. By embedding softcore processors within an FPGA, no I/O resources are required to communicate with other embedded modules (whether peripherals, custom logic, or other processors). An unprecedented level of system-design flexibility is offered, as well as reductions in PCB requirements, power

[2]Some modern FPGAs include embedded hardcore processors, including Altera Excalibur [5] and Xilinx Virtex II Pro [36] devices. The number of hardcore processors, however, is fi xed and limited to a small quantity.

consumption, and electro-magnetic interference (EMI) as discrete modules are coalesced into one package [35].

MPOPC systems are generally tailored for particular computational tasks. Consequently, the systems tend to be somewhat heterogeneous (*i.e.*, processors are configured differently and are individually tailored to specific tasks), although this is not always the case. The processors in these systems also tend to be loosely coupled, or entirely independent, even in situations where different processors share memory or a common bus. MPOPC systems are relatively new, as traditionally each processor was implemented on a single FPGA and several FPGAs were combined to create a multiprocessing system (much like their discrete custom-logic cousins). Some examples of MPOPC systems include:

- A loosely coupled set of eight Altera Nios softcore processors on a single bus has been used to perform LU matrix factorizations for power flow analysis [32].

- A parallel data system, controlled by a central instruction stream, with up to eighty-eight custom processors taking advantage of on-chip hardware multipliers [23].

- A hardware-software co-configuration system developed to generate a multiprocessor Xilinx MicroBlaze system and standardized embedded real-time operating system. The system uses four independent SRAM banks to support up to four or five softcores in a simple shared-memory architecture processors. [31]

- SoCrates, a two-node distributed shared-memory machine. Each node consists of an ARM7TDMI [27] clone and 8 kB of memory. [17]

1.5 Statement of Thesis

This thesis describes the design and development of an easy-to-use cache-coherent Symmetric-Multi-Processor-On-Programmable-Chip (SMPOPC) system using vendor-provided IP. The goal is to implement the system with a minimum of user intervention and without any invasive alterations to the vendor-provided processor or bus. While other MPOPC systems provide relevant information, no recent MPOPC systems use an SMP architecture with caches.

The salient features of vendor-provided softcores and bus interfaces that contribute to the challenges in building an SMPOPC system and their corresponding solutions are highlighted. A generic MPOPC system based on the SMP architecture was chosen since there was no particular application in mind, and such an architecture offers a number of advantages, including:

- Softcore processors embedded into a single device represents an inexpensive way of increasing the overall performance of an embedded system. The number of processors is limited only by the device capacity.

- An N-way SMP architecture is flexible. Once a particular system is generated, any number of applications can be developed; more time and effort can be spent on application development rather than on generating hardware specialized to a particular task (which may not necessarily result in performance gains compared to an enhanced software solution).

- Since a particular application is not specified, SMP potentially offers performance improvements on a fairly general class of computational tasks. Embedded systems in particular would benefit from an increase in computational power.

- Using a known architecture immediately implies that proven algorithms for various computational tasks are available (*e.g.*, in the case of the LU factorization [32], software algorithms for SMP architectures are well-known).

- An operating system can be relatively easily written, leveraging on the knowledge base of known issues associated with SMP systems (*e.g.*, Linux natively supports SMP systems). Furthermore, existing SMP-oriented applications can be ported to new systems with little or no alterations.

A major objective in the development of this system was to leverage the best features of softcore processors and the available features of modern FPGA devices. The Nios processor (and associated Avalon bus) was chosen due to its popularity (the Linux operating system has been ported to run on Nios processors). Since it is vendor-provided, it is optimized for each of the different families of Altera devices. Finally, the use of a vendor-provided softcore processor implies excellent support, software and development tools.

The Nios processor supports the use of advanced on-chip memory to serve as cache to improve system performance. Unfortunately, in an MPOPC system (especially in the context of SMP), the use of individual caches for each processor raises issues; the Nios was not intended to be used within the context of an SMP architecture and this created *cache coherency issues*. Therefore, the issue of cache coherency in the context of the Nios softcore processor and the Avalon bus is addressed. This task is accomplished with no disruption to the Nios processor and Avalon bus designs. This implies the system can be used as an “add-on” to existing systems.

1.6 Thesis Contributions

This thesis makes the following contributions to the existing body of research:

- illustrates the challenges associated with implementing an SMPOPC system using vendor-provided softcores and bus interfaces;

- describes a generic hybrid snooping cache-coherency protocol;

- describes two non-intrusive hardware-software solutions: a prototype that shows that cache coherency can be maintained, but does not handle the case of multiple in-flight writes and a second-generation module, which addresses the critical flaw of the prototype and offers performance improvements with little additional hardware; and
- provides a performance analysis of a real cache-coherent SMPOPC system, showing that there is little impact on the system clock frequency (does not contribute to the critical path) while using few PLD resources to implement.

1.7 Outline of Thesis

Chapter 2 provides an introduction to multiprocessing, focussing on the area of symmetric multiprocessing, and discusses the issue of cache coherency in the context of SMP systems. Chapter 3 presents the Altera Nios softcore processor and associated Avalon bus, as well as the particular challenges they present when used in an SMP architecture. Chapter 4 describes an initial proof-of-concept solution that shows that the challenges can be overcome. A more complete second-generation design is presented in Chapter 5. Chapter 6 provides details on the development platform used, as well as an analysis on the experimental results conducted on the system. Finally, Chapter 7 concludes the thesis and outlines possible future research in this area.

Chapter 2

Multiprocessing

Modern small computers are dominated by uniprocessor systems. Uniprocessor systems feature powerful microprocessors that scale in frequency to beyond 3.5 GHz. This currently provides ample performance for all but the most demanding applications. The typical desktop user, running word processors, internet browsers, and audio/video applications, often has a hard time presenting a serious load to the processor, even when these applications are used simultaneously. Cutting-edge computer games, science, industry, and some business applications, however, still benefit from additional computing power. One of the most effective ways to improve performance beyond a single processor is to use multiple processors [22]. This is cost-effective, as multiprocessor systems often have a better cost-performance ratio than a uniprocessor system [33]. It is also significantly easier and less costly to add existing commodity processors, rather than creating a custom processor. The cost of a single processor design can be amortized when system vendors offer a wider range of computing platforms for applications with different computational demands [30].

A number of multiprocessor architectures exist. Most mainstream architectures feature fewer than one hundred processors [22]. Some supercomputer architectures incorporate thousands of processors [37]. Some specialized scientific applications have lead to the design of vector processors, and their associated multiprocessing architectures. Comparing these various architectures requires a taxonomy to better describe each alternative and the driving reasons behind each design. To this end, Flynn's taxonomy of parallel computer architectures [20] is often used, and is described below:

- Single instruction stream, single data stream (SISD) - This is a typical uniprocessor system. A single set of instructions is executed using a single stream of data.

- Single instruction stream, multiple data streams (SIMD) - In this category, multiple processors execute the same set of instructions on multiple data streams. Each processor accesses its own data memory (multiple data), but there is one shared instruction memory and a control processor, which directs the other processors by fetching and dispatching instructions. Typically, these systems are special purpose machines. Modern uniprocessors, however, often include entire SIMD instruction sets, such as Intel's MMX, SSE, SSE2, and AMD's 3DNow!. These SIMD instructions target multimedia and communications applications, allowing uniprocessors to achieve new levels of performance by exploiting parallelism inherent in these types of applications.

- Multiple instruction stream, single data stream (MISD) - No system of this nature has been made commercially available. Example of applications include cryptographic processors and multiple independent frequency filters operating on the same signal.

- Multiple instruction stream, multiple data stream (MIMD) - A MIMD system features independent processors, each of which executes its own instructions and operates on its own data. Typically, commodity off-the-shelf processors are used in such a system [22].

MIMD machines have emerged as the dominant category for general-purpose multiprocessing. They can function equally well as single-user machines focusing on performing a single task with high efficiency, or as a multiprogrammed machine simultaneously running any number of tasks, or as some combination of the two [22]. Within the MIMD category, two architectures exist: centralized shared-memory architectures, and distributed-memory architecture.

Centralized shared-memory computers typically support a small number of processors (usually fewer than sixty-four). If the number of processors is small, it becomes possible for an interconnection network (often a bus) to provide uniform access to a single, centralized memory. Unfortunately, access to memory through a shared bus does not scale with the number of processors and therefore the bus becomes a performance bottleneck [21]. This problem can be somewhat mitigated through the use of cache (see Section 2.2). Symmetric multiprocessor systems are the most popular implementation of the centralized shared-memory architecture.

For completeness, distributed-memory architectures are mentioned briefly. These systems are often composed of self-contained computer systems (including one or more processors and local memory). These systems are connected via a high-speed interconnection network (such as Ethernet). Physically distributed memory allows the system to support a much larger number of processors. The Earth Simulator project [37], for example, uses 640 processor nodes, with each node including eight arithmetic processors and 16 GB of shared memory (for a total of 5120 processors and 10 TB of memory).

2.1 Symmetric Multiprocessing

MIMD symmetric multiprocessor systems are the most popular computer multiprocessor architecture. In an SMP system, a shared bus is used to interconnect processors to a single centralized memory. Figure 2.1 gives a high-level architectural overview of a typical SMP system [22]. Bus contention, combined with the additional operating system overhead required to coordinate multiple processors and the limited parallelism that can be achieved in applications, means that each additional processor provides diminishing returns, as described by Amdahl's Law [15]. The cost of an SMP system is incremental over that of a uniprocessor system; the increased cost being the additional processors and a slightly more expensive motherboard.

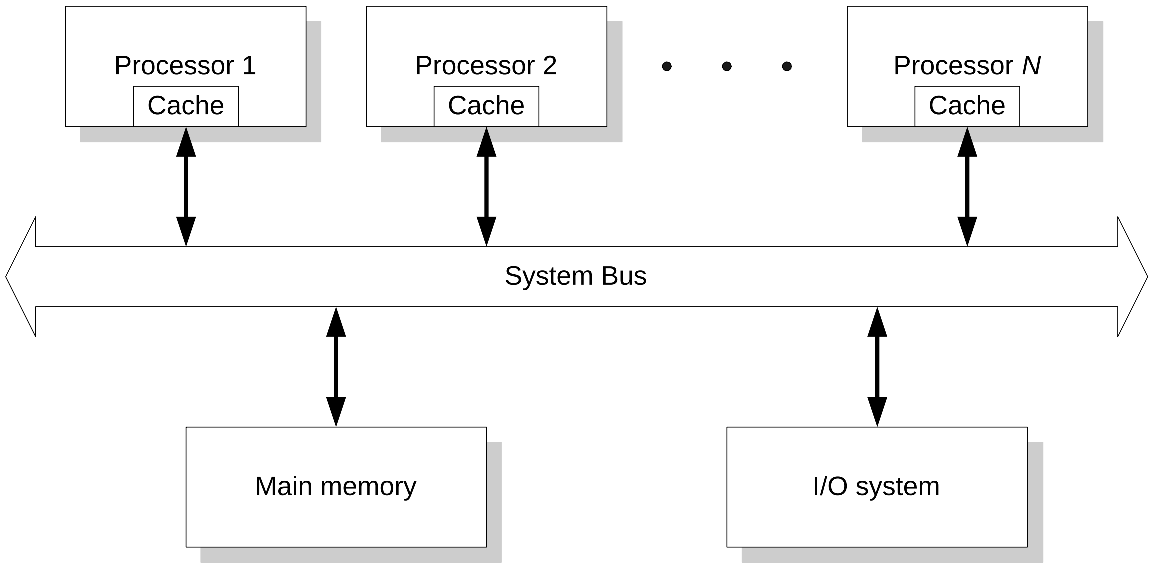

Figure 2.1: Basic Architecture of a Typical *N*-Way SMP System

The symmetry is three-fold in the system, and encompasses the processors, the memory, and I/O. All processors are functionally identical and are arranged in a flat hierarchy. That is, there are no master-slave relationships or geometry that limits inter-processor communication to particular processors. Memory symmetry refers to the ability of all processors to use the same addresses to share the same address space. I/O is symmetric when all processors share access to the same I/O subsystem, and any interrupt can be received by any processor. There are no dedicated processors for handling interrupts or I/O in this model. Memory and I/O symmetry are conducive to hardware scalability. The shared nature of symmetry helps to eliminate or reduce potential bottlenecks in critical subsystems. Additionally, symmetry leads to software standardization, as system developers can produce systems with differing numbers of processors that can all execute the same binaries. [25]

Though functionally identical, it is common to differentiate between a bootstrap processor (BSP) and an application processor (AP) [25]. This difference is only in effect during initialization and shutdown of the system, and is provided as a convenience. Any processor in the system may be the BSP, and is typically determined by hardware, or a combination of hardware and firmware. The role of the BSP entails initializing the system and booting the operating system (OS). During this process, the APs (all other processors) are held in reset to avoid any conflict that multiple uninitialized processors might cause.

To take advantage of multiple processors in an SMP system, both the operating system (if present) and the application must support multiple processors. If the operating system is not SMP aware, then only the BSP executes instructions, with the additional processors running idle. Most consumer applications, such as word processors and games, are not written to take advantage of multiple processors. These applications do not usually benefit from additional processors. The user will still notice a performance increase if the system is multiprogrammed, since more than one program can execute simultaneously (for example, a user could listen to music files while reading e-mail). These applications are not written with SMP in mind since they would suffer a performance loss on uniprocessor systems (their most common platform). The loss is caused by the operating system overhead of switching between threads, which does not accomplish any useful work on a single processor.

To truly take advantage of an SMP system, an application must be multithreaded. Scientific, industrial, and business programs are often designed to run on multiple processors, explicitly taking advantage of inherent parallelism in the application. Server applications and distributed computing projects can also benefit greatly from additional processors.

2.2 Cache

As mentioned, access to memory through a shared bus does not scale with the number of processors, therefore the bus becomes a performance bottleneck. This problem can be somewhat mitigated through the use of one or more levels of *cache*, which is one feature that is frequently used to increase processor performance, even in the uniprocessor case.

These caches are composed of fast memory that sit between the processor and the main memory to reduce latency by fulfilling repeated memory requests to the same location. Caches take advantage of the spatial and temporal locality characteristics of executing code to store recently used memory blocks. When the processor accesses memory that is cached, the cache is able to supply the data and no transaction occurs on the shared memory bus (reducing contention on that bus). Main memory is quite slow when compared to the speed of modern processors. A cache helps the memory subsystem supply instructions and data at the rate the processor consumes them.

Unfortunately, caching is not without its drawbacks. The speed of cache memory is a direct result of the increased number of transistors used to implement each bit of storage. This precludes designing a large amount of on-die cache for each processor, where it is the fastest and most effective. This leads to most systems implementing a hierarchical memory subsystem. The caches are fast and small memories. Memory devices become larger and slower going from the processor to the main memory and beyond to magnetic hard drives, which is the largest and slowest form of memory in the system.

When a processor writes to a memory block, the cache is designed to follow one of two policies: write-through or write-back. The *write-through* policy specifies that on a write, contents are written into the cache and to lower-level memory (either another cache level or physical memory). In the *write-back* policy, the contents are written to cache, and are only written back to lower levels when that cache line is replaced. While simpler to implement, the write-through policy tends to cause main memory transactions that may have been avoided by a write-back policy. Conversely, using a write-back policy effectively hides writes from main memory and the rest of the system until cache line replacement.

The rest of this section deals with memory coherency and consistency, two concepts that are important for correct operation of multiprocessor systems. The effect of caches on memory coherency is also addressed.

2.2.1 Memory Coherency and Caches

A memory system is considered to be *coherent* if a read to an arbitrary memory address returns the most recently written value. This definition, however, encompasses two aspects of memory system behaviour: *coherency* and *consistency*. Writing correct shared-memory programs require careful consideration of both aspects.

A coherent memory system exhibits three properties: preservation of program order, a coherent view of memory, and write serialization [22]. The preservation of program order simply means that if a processor reads a memory location after writing to it, the written value is returned. Coherent memory means that if a processor writes to a memory location that is followed by a read by a different processor, then the written value is returned if the two accesses are sufficiently separated and no other writes occur in between them. Write serialization means that if two writes to the same memory address by two different processors occur, all processors in the system see the writes occurring in the same order.

Cache memory leads to the problem of maintaining coherency in multiprocessor systems. The problem is that the view of memory by different processors, through their caches, may be different. That is, copies of shared data may reside in multiple caches, and when any processor modifies the cached data, all other caches that contain that data will have the old, incorrect value (affecting the second property of coherent memory systems). These other caches must be informed of the change for proper operation of the program. [21]

Table 2.1 illustrates the cache-coherence problem. Suppose there are two processors with write-through caches in a system. When processor A reads memory location X, it is stored in processor A's cache. The same occurs when processor B reads memory location X. If processor B subsequently writes a different value to memory location X, then processor A's cache will contain a stale value for that location. If processor A reads location X again after processor B's write, it will retrieve stale data from the cache. [22]

Time	**Event**	**Cache A**	**Cache B**	**Memory**
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU B writes X	1	0	0

Table 2.1: The Cache Coherence Problem in Multiprocessor Systems

There are two basic protocol classes for enforcing cache coherency: snooping and directory-based [22]. *Snooping* protocols involve having processor caches monitor (snoop) the shared memory bus for writes by other processors. If the processor's cache contains the data being written, the protocol can either invalidate its cache line (forcing a read to memory on the next access) or update its contents. Example protocols include Write Once, Synapse N+1, Berkeley, Illinois, and Firefly [16] [18]. In a *directory*-based protocol, a central directory tracks the sharing status of blocks of physical memory. When a processor writes to a memory block, it secures exclusive-write access to that block. Messages are passed in order to ensure that no stale memory blocks exist in processor caches. Example directory protocols include the $D\mathrm{ir}_1\mathrm{NB}$ and $D\mathrm{ir}_0\mathrm{B}$ schemes [2]. A careful analysis (see Section 4.1) of the Nios processor and Avalon bus in an SMP configuration will show that neither of these two methods are feasible without making invasive changes to either the processor or the bus structure. A hybrid cache coherency protocol is developed instead.

2.2.2 Memory Consistency

Memory consistency refers to the rules that a particular computer system follows with respect to the ordering of memory accesses (reads and writes). A memory consistency model provides a formal specification to the programmer of how the memory system behaves. The model places restrictions on the values that can be returned by a read during shared-memory program execution, and that behaviour restricts what hardware and software optimizations may be used. Defining a memory consistency model is critical to ensuring correct operation of parallel shared-memory programs. The model that applies to the final SMPOPC Nios system cannot be described until the design and behaviour has been finalized, however a number of models are briefly described here.

Since programs are executed sequentially, one would expect that a read would return the value of the most recent preceding write. This is strict consistency, and is exhibited by uniprocessors through preservation of program order (*i.e.*, the order of execution as described by the program). Multiprocessor systems with no cache and shared access to a memory bus also provide strict consistency.

The sequential consistency model is a relaxed version of the strict model, wherein all memory accesses are serialized (they execute one at a time, or atomically), and that operations from a single processor appear to execute in program order [26]. This model is simple and behaves as programmers expect from computers. This model, unfortunately, disallows many optimizations in multiprocessors systems that are available in uniprocessor systems [1]. As a result, a number of more relaxed models exist, many of which are used by real systems.

While some optimizations pose a challenge to the sequential model, adding a data cache presents a new set of similar challenges. In particular, two issues present themselves: detecting when a write is complete, to preserve program order between a write and following operations; also, invalidating other caches in the system on a write is inherently non-atomic, making it harder to make writes appear atomic. The first issue is solved by implementing a mechanism to acknowledge the receipt of invalidation or update messages by target caches. Once all caches have acknowledged the write, the processor issuing the write is notified and may continue execution. The non-atomicity issue can be addressed by forcing write serialization when writing to the same location, and by disallowing the read of a written value until all caches have acknowledged the receipt of the invalidation or update message. [1]

Beyond the sequential consistency model lie other, more relaxed models. These models relax specific program orderings, such as read after write (RAW) ordering, write after write (WAW) ordering, or any access after a read (RWAR) ordering. Typically, models relaxing the later orderings also relax the earlier orderings as well. Through order relaxation, two specific abilities can be enabled: read others' write early, and read own write early, wherein a processor can read another processor's or its own write (respectively) prior to full acknowledgement of the write by all caches.

Relaxing RAW ordering defines when the writing processor is able to read the new value after a write, with respect to same location serialization, and with respect to when the value is visible to other processors. The common Intel x86 architecture relaxes both constraints, such that a read can return the value of a write prior to being serialized or made visible to other processors. Relaxing WAW ordering allows processors to pipeline or overlap writes to different memory locations. Relaxing all program orders allow any memory operation to be reordered with the following memory operation if they both access different memory locations.

Regardless of which orderings are relaxed, models provide safety net mechanisms that allow the programmer to enforce program order when used. These often entail explicit serialization, synchronization or fence instructions, or specific sequences of instructions that enforce program order.

2.3 Summary

This chapter was an introduction to the area of multiprocessing, highlighting centralized shared-memory symmetric multiprocessing systems. Cache memory was explained, and the problem of cache coherency in multiprocessor systems was illustrated. Finally, memory consistency, an important aspect for ensuring the correctness of parallel programs, was presented. In the next chapter, the Altera Nios softcore processor and Avalon bus interface are examined in the context of SMP systems.

Chapter 3

The Nios Processor and Avalon Bus

The Altera Nios processor and Avalon bus module are of central importance when analyzing SMP Nios systems in the context of cache coherency. Both processor and bus are described here, with a focus on the properties that are relevant to symmetric multiprocessing. A general description of the Nios features and capabilities is provided, followed by an elaboration on cache memory and interrupt processing. Details of the Avalon bus are presented in the remainder of the chapter.

In reference to the Nios, the terms core, softcore, processor, microprocessor, and central processing unit (CPU) are interchangeable. Interrupts and exceptions are also synonyms.

3.1 Nios Embedded Softcore Processor

The Nios embedded softcore processor is designed specifically for SOPCs. It is customizable for a wide range of applications, and is optimized for Altera PLDs. The 32-bit Nios, when combined with external flash program storage and large external main memory is a powerful SOPC. Examples of the flexibility of the Nios are provided throughout this section.

Nios v3.0 features a single-issue five-stage pipeline reduced instruction set computer (RISC) architecture. Figure 3.1 shows a block diagram of the Nios core. The pipeline implementation is transparent to software.

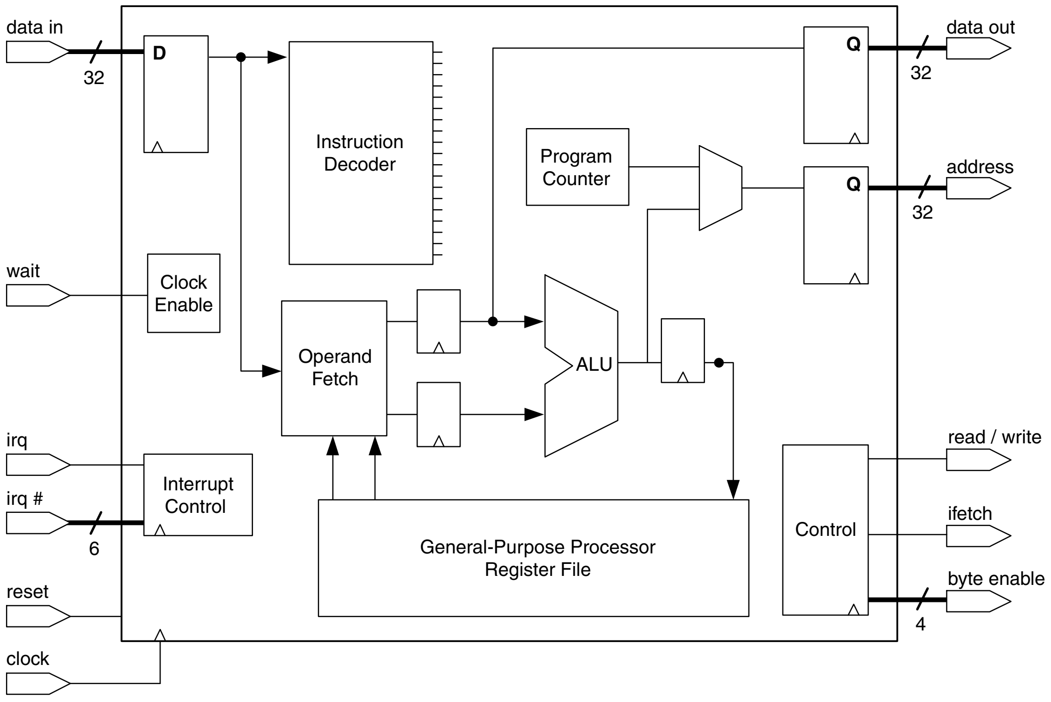

Figure 3.1: Nios Core Block Diagram [11]

The Nios is available in both 16-bit and 32-bit variants. The word size of each variant applies to the data bus size, arithmetic logic unit (ALU) width, internal register width, and address bus size. Both variants have a simple and complete instruction set that utilizes 16-bit instruction words to reduce code size and bandwidth requirements.

The Nios instruction set architecture (ISA) is tailored to be generated from the popular C and C++ high-level programming languages. The ISA includes a standard set of arithmetic and logic operations. Bit operations, byte extraction, data movement, control flow and conditional execution are also supported. The processor is little-endian and supports the following addressing modes: 5- or 16-bit immediate, full or partial width register-indirect, and full or partial width register-indirect with offset. The multiply instruction can be configured to be fully implemented in hardware, partially in hardware, or fully in software, depending on the needs of the system designer.

A large windowed register file is implemented within the Nios core. The window makes thirty-two registers available at a time, and slides with a granularity of sixteen registers. These registers are divided into four classes: eight registers each for the globals (%g), locals (%l), incoming parameters (%i), and outgoing parameters (%o). The system designer is able to select a register file size of 128, 256, or 512 registers (providing eight, sixteen, or thirty-two register windows, respectively), depending on anticipated need. The designer can optionally use the MFLAT compiler option, where only thirty-two registers are available, with no windowing. Software is then obliged to save register values to memory, increasing the average context switch time. The worse case context switch time (*i.e.*, saving all registers to memory), however, is constant and significantly less than the run-time for the default register window overflow or underflow interrupt service routine (ISR).

The Nios processor features a modified-Harvard memory architecture with separate data and instruction-memory bus masters. Both control ports are implemented as Avalon bus masters. The instruction bus-master is a read-only, 16-bit wide (the instruction word size), latency-aware Avalon bus-master. It is used to fetch instructions to be executed by the Nios. The latency awareness gives the Nios the ability to perform read operations to latent memory devices[1]. This minimizes the impact of latent memory while increasing the operating frequency of the system as a whole. The system designer is also able to store program instructions in high-latency, non-volatile memories such as flash memory. The instruction master issues new read requests

[1]Latent memories have long access times compared to the system clock period.

prior to the completion of the previous read, using branch-not-taken branch prediction to provide zero-latency speculative fetch addresses. A penalty is only assessed when the branch is taken (mispredicted). The Nios ISA also specifies a single branch delay slot.

A Nios data master is sized according to the processor word size (16- or 32-bits). It performs data reads and writes to memory, but also fetches interrupt vectors (see Section 3.3) from the interrupt vector table during exception handling. In the context of data, the master is not latency-aware since it is not useful to predict data addresses or continue execution before access is complete [10]. The result is that accessing latent memories incurs wait states; assuming no arbitration conflicts, single cycle accesses may only be achieved when using zero-wait-state memory.

The Altera Nios provides the system designer with a number of feature-performance-size trade-off customizations to better meet the requirements of the system. The system-development software supports a set of four general preset configurations (standard features / average LE usage, minimal features / minimal LE usage, full features / maximum LE usage, and standard debug / average LE usage). These general preset configurations select a set number of specific customizations, such as register file size and multiplier implementation mentioned above. Other options available for customization include:

- The option to make the `WVALID` control register writeable for window pointer overflow and underflow control (some operating systems require this feature). This option increases the size of the CPU by approximately fifteen LEs.

- A pipeline implementation using more LEs (reducing stalls) or fewer LEs (increasing stalls). This option implements a forwarding path from the output of the ALU to an input of the ALU, eliminating stalls for certain data hazards. Approximately thirty-two LEs are used, and there may be a reduction in system operating frequency.

- An instruction decoder implementation using LEs or on-chip memory.
- Support for rotate through carry (RLC/RRC) instructions. The provided software development kit (SDK) compiler does not use these instructions. They are provided for user-written assembly, and they require twelve to twenty-one LEs to implement.
- Support for interrupts and software traps. This is on by default, and generates interrupt control signals and supporting hardware in the Nios core. This should only be disabled when trying achieve the smallest Nios implementation possible. Safely disabling this option means that the designer knows that the software will not cause register window exceptions, will not execute `TRAP` instructions, and the system will not have any hardware interrupt sources.
- Support for optional C/C++ libraries and subroutines:
  - Catch spurious interrupts - a default interrupt handler is installed. Increases code size and memory usage slightly.
  - Call C++ constructors - used to initialize statically allocated C++ classes.
  - Window pointer manager - to handle register window underflows. Can reduce code size if the designer knows the software function call depth will not exceed the number of register windows.
  - Fast multiply - for purely software multiply implementations. Increases code size of multiply subroutine.
  - Small printf() - reduces code size (from 40 kB for a full implementation to 1 kB) when floating-point support is not required. Integers, characters, and strings are supported in the minimal implementation.

- An on-chip hardware debug module, which allows system designers to use hardware breakpoints and tracing with additional software and/or hardware.

One feature of softcore processors that provides unprecedented extensibility over their hardcore counterparts is custom instructions. That is, the Nios allows system designers to incorporate custom logic directly into the processor's ALU (as shown in Figure 3.2). This allows a designer to accelerate time-critical software algorithms by implementing complex computational tasks as single-cycle combinational or multi-cycle sequential operations. A designer may reduce a complex and lengthy sequence of RISC instructions into a single custom instruction implemented in hardware. The provided SDK includes facilities (C macros) for accessing custom instruction hardware via special assembly stub instructions (`USR0 - USR4`). Further details regarding the Nios CPU can be found in [10] and [11].

Of particular relevance to system development is (i) the Nios can take advantage of on-chip memory for cache, (ii) its support of vectored exceptions including interrupts generated by external hardware, and (iii) its interface to the Avalon bus. It is important to note that the Nios was not designed with cache coherency facilities for use in an SMP architecture when using on-chip memory for cache.

3.2 Nios Cache Memory

A Nios core can be configured with optional single-cycle L1 instruction and data caches. The designer may specify each cache to be from 1 kB to 16 kB in size (size must be a power of two). Each cache is direct-mapped, such that the low bits of the memory address are used as an index to the cache, as shown in Figure 3.3. Direct-mapped caches are simpler to implement and result in a smaller hardware circuit, but have a smaller hit rate than fully-associative or set-associative caches [22].

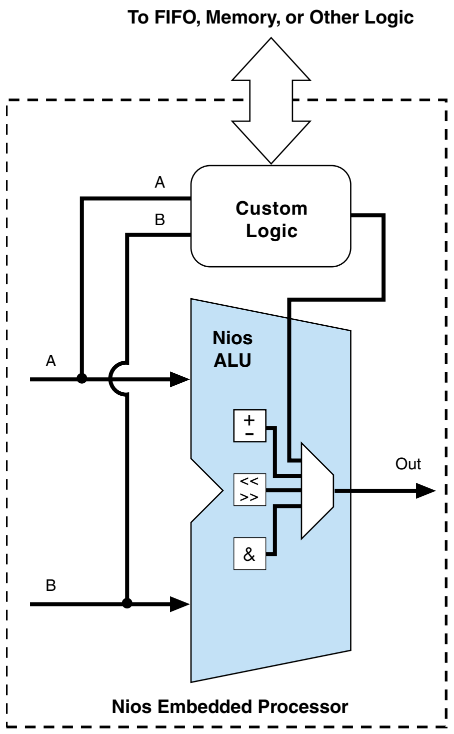

Figure 3.2: Custom Logic and the Nios ALU [4]

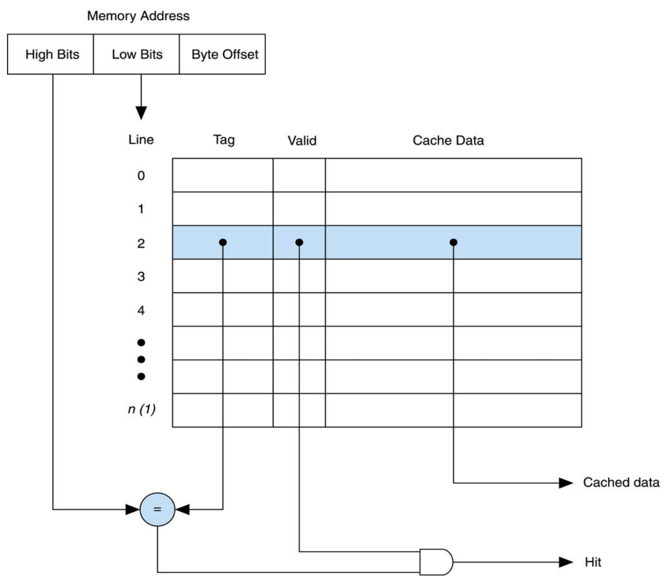

Figure 3.3: Direct-Mapped Cache [11]

The Nios data cache uses a write-through policy (meaning a full write request is made to memory, in addition to the cache). The instruction cache does not support writes, since the instruction master does not either. Furthermore, the data cache can be automatically bypassed when performing a load instruction by preceding it with the prefix instruction `PFXIO`. This is particularly useful when accessing Nios peripherals, as I/O operations should not be cached.

A Nios system requires instruction and data cache initialization and enabling before they can be used. Initialization is achieved by invalidating every cache line. The Nios provides for this facility via the write-only `ICACHE` and `DCACHE` control registers. These line-invalidate registers invalidate the cache line corresponding to the memory address that is written to them. The instruction and data caches each have an enable bit in the `STATUS` control register which must be set, allowing for run-time cache enabling and disabling. A cache must be disabled prior to using its line-invalidate register. Since the Nios cache does not have built-in automatic cache coherency facilities, these line-invalidate registers are critical for informing a cache that another processor has written data to cached memory.

Instruction and data caches are implemented using on-chip memory and a small amount of support logic. Only relatively modern FPGAs contain the required memory resources to support cache. Cache may only be used with 32-bit Nios processors, and only when targetting Altera Cyclone, Cyclone II, Stratix, Stratix GX or Stratix II FPGAs. The only relevant features of the cache in the context of coherency are the ability to invalidate individual cache lines and the use of a write-through policy.

3.3 Nios Interrupt Processing

A Nios CPU supports up to sixty-four vector exceptions, including external hardware interrupts, internal exceptions, and software `TRAP` instructions. There is a global interrupt enable bit in the `STATUS` control register, as well as a 6-bit interrupt priority mask. Each vector number is its own priority, with 0 being the highest priority and 63 being the lowest. The Nios provides precise exception handling; that is, the interrupted program is restored to a state as if the exception had not occurred.

Internal exceptions represent register window underflow or overflow, which occurs when too many `SAVE` or `RESTORE` instructions are executed, respectively. Direct software exceptions call exception handlers via the `TRAP` instruction. An immediate value encoded with the instruction represents the exception number. Software exceptions are processed regardless of whether interrupts are enabled or not, and regardless of the current interrupt priority.

External hardware interrupts are raised by driving a 6-bit interrupt number onto the Avalon bus `irq_number` signal and asserting the `irq` signal. The Avalon bus (see Section 3.4) uses automatically generated connection logic that allows peripherals to simply assert a single `irq` signal, which is decoded into the proper interrupt number and presented to the Nios. If interrupts are enabled and the requested interrupt has a higher priority than the priority mask, then the exception is handled. Interrupt priority 0, which is assigned to the hardware debug module, is always handled, regardless of current priority or whether interrupts are enabled or not. External interrupt sources should assert the `irq` signal until acknowledged by software (usually via a register write). `irq` signal de-assertion prior to the beginning of interrupt processing results in an ignored interrupt. In the case of multiple Nios masters, a slave peripheral's interrupt is raised on all processors that can master that peripheral (*i.e.*, all processors connected to its slave port).

Figure 3.4 shows the Nios exception handling process. Once an interrupt request is received, the current state (context) of the system is saved. This includes the following actions:

- Saving the `STATUS` register to the `ISTATUS` register.
- Opening a new register window (automatic and very low latency register saving).
- Disabling global interrupts in the `STATUS` register.
- Setting the interrupt priority mask in the `STATUS` register according to the current interrupt.
- Saving the program counter (PC) of the interrupted program to register %o7 (the last "output" register of the current register window).
- Retrieving the address of the interrupt's ISR from the interrupt vector table.

The interrupt vector table consists of sixty-four 4-byte entries (256 bytes total). Each entry represents the starting address of the interrupt service routine (ISR, or sometimes exception handler) for that interrupt number. The interrupt vector table may reside in random access memory (RAM) or read-only memory (ROM), and its base address (`VECBASE`) is configurable. An interrupt's entry is calculated by multiplying the interrupt number by four to determine its offset,

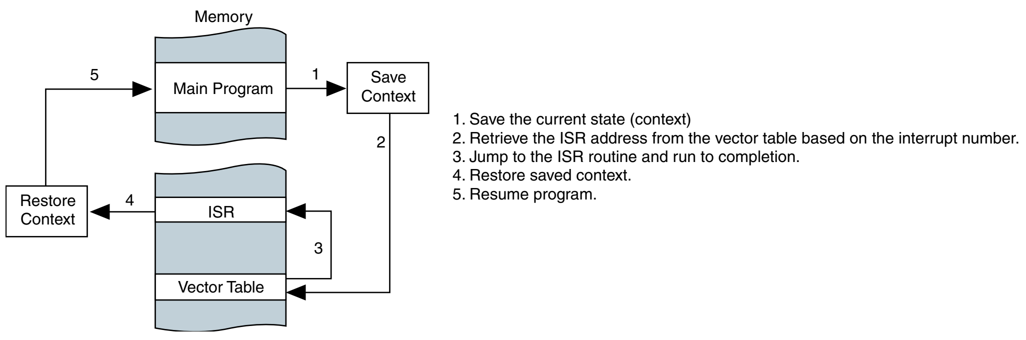
Memory
Main Program
Save Context
Restore Context
ISR
Vector Table
1
2
3
4
5
1. Save the current state (context)
2. Retrieve the ISR address from the vector table based on the interrupt number.
3. Jump to the ISR routine and run to completion.
4. Restore saved context.
5. Resume program.

Figure 3.4: Nios Exception Handling Process [9]

then adding the vector table base address. For example, interrupt #3 is located at memory address $\texttt{VECBASE} + 3 \times 4 = \texttt{VECBASE} + 12$. Note that interrupt 0 (the hardware debug module) is handled differently, and thus entry 0 in the interrupt vector table is unused. Table 3.1 defines the vector table, where the first sixteen vectors are defined by Altera; the remaining forty-eight are user-defined interrupt vectors (for software `TRAP` instructions or assigned to hardware modules at system build time).

Vector Number	**Vector Offset (Hex)**	**Assignment**
0	000	Hardware debug module
1	004	Register window underflow
2	008	Register window overflow
3 - 5	00c - 014	GNUPro debugger
6 - 15	018 - 03c	Reserved for future use
16 - 63	040 - 0fc	Available vectors

Table 3.1: Exception Vector Assignments

The address returned from the interrupt vector table is loaded into the PC, and the ISR is executed. The last instruction of the ISR is `TRET`, which indicates that the ISR is complete. This causes the saved context to be restored and the interrupted program resumes execution.

The Nios supports nested exceptions, which allow higher priority exceptions to interrupt lower priority exceptions. The same exception handling process occurs in this case, except that the interrupted program is itself an exception. Nested exceptions are enabled by re-enabling global interrupts within an ISR (recall that they are automatically disabled by the exception handling hardware).

At this point, it is important to distinguish between two different types of ISRs that can be implemented in a Nios system. They are categorized into simple and complex exception handlers. A simple ISR has the following properties:

- It does not re-enable interrupts.
- It does not use `SAVE`, `RESTORE`, or `TRAP` instructions (either directly or by calling subroutines that execute them).
- It does not alter the contents of registers %g0..%g7, or %i0..%i7. An ISR is always free to use the %l0..%l7 and %o0..%o7 registers.

The first three properties ensure that the register window will not change, and therefore no window overflows or underflows are possible. If they were possible, interrupts would need to be re-enabled such that the overflow or underflow ISR may execute. The fourth condition exists so that these registers will not be altered once the ISR is complete, as the interrupted code has direct access to the %g and %i register series. This saves the routine from having to save and restore any of those registers.

A complex exception handler violates one or more of the conditions listed above. Such an ISR is necessary to allow nested interrupts or the execution of more complex code (such as subroutines that `SAVE`, `RESTORE`, or `TRAP`). In addition to the context saving automatically performed by the hardware, a complex ISR must also ensure the following:

- The contents of `ISTATUS` must be preserved before re-enabling interrupts (which automatically overwrite its contents with `STATUS`).
- The current window pointer must be checked to ensure that re-enabling interrupts will not cause a register window underflow (or it must take appropriate action to prevent an underflow).
- The ISR must re-enable interrupts (after satisfying the first two conditions) before executing a `SAVE` or `RESTORE` instruction (directly or indirectly). This allows register window overflow and underflow handlers to execute, if necessary.
- Prior to completion of the ISR, the contents of the `ISTATUS`, current window pointer, and any used registers in the %g or %i series must be restored.

The Nios SDK provides generic facilities to easily write ISRs as normal C or C++ routines, as opposed to Nios assembly. These facilities include two routines, `nr_installuserisr` and `nr_installuserisr2`, which both install a user ISR to a specific interrupt number. Knowledge of the Nios interrupt vector table and its use is not required to use these routines. The routines allow the programmer to access normal facilities such as easily calling other functions.

The first routine passes an integer `context` argument, while the second additionally passes the interrupt number and the interrupted PC. The second installer is useful for using the same ISR for multiple interrupt sources. Installing an ISR in this fashion automatically makes it a complex exception handler, as these routines wrap the ISR in a funnel assembly routine, which essentially performs a full function call, as well as enabling interrupts prior to executing the ISR. It is this funnel code that allows the ISR to be written like a normal routine. Unfortunately, the funnel code also introduces complexity and latency when entering and exiting an ISR, thereby increasing the run-time between the exception event and returning to normal execution. These latencies are acceptable for many situations, such as a UART ISR. Altera's simulation results for such an ISR are reproduced in Table 3.2. Unfortunately, this is unacceptable for latency-critical ISRs.

Item	**Time (μs)**	**CPU Cycles**
ISR entry latency	2.79	93
Running the ISR	3.21	107
ISR exit latency	1.92	64
Total	**7.92**	**264**

Table 3.2: UART ISR Latency for 33 MHz Clock [9]

The funnel code is composed of thirty-five assembly instructions prior to execution of the ISR, and twenty-six assembly instructions upon returning from the ISR. This includes sixteen and fourteen data memory accesses prior to and after ISR execution, respectively. ISRs that use the provided installation routines are known as "user" ISRs.

In contrast to a user ISR, a "system" ISR does not use the funnel routine to setup register windows and save register contents. When an exception occurs, the processor jumps directly to the assembly routine, thus eliminating entry and exit latency, and shortening the overall execution time spent servicing the interrupt. Hence, the cache coherency ISR was written as a system ISR. Instead of utilizing the provided installation routines, a generic system ISR installer was written. This code is listed in Appendix A.

3.4 Avalon Bus

The Avalon bus is a bus architecture that was designed to serve as the interconnection network for a SOPC. To this end, the Avalon bus is a simple interface that specifies the signals between master and slave ports, as well as the timing of the protocol. Besides simplicity, the Avalon bus was designed to also use minimal logic resources within a PLD and to have synchronous operation to avoid complex timing analysis issues [8]. When generating an SOPC system using the Avalon bus, all interconnection logic is automatically generated by Altera's SOPC Builder tool. Configuration is performed using the easy-to-use SOPC Builder graphical user interface.

A traditional shared bus implementation uses a single tri-state bus in which master-slave pairs are arbitrated. Any devices connected to the bus that are not participating in the current transaction must not drive any values on the bus, using tri-state drivers in high-impedance mode. This works well in traditional SMP systems because master and slave devices are physically separate, located on self-contained PCBs or across backplanes. Designs use a shared set of bus lines to conserve board space and the number of available I/O pins. Timing issues are also simplified. A single bus becomes the bandwidth bottleneck, as only one transaction may occur on the bus at a time. While most PLDs provide tri-state drivers for off-chip communication, only some PLDs provide internal resources to support a limited internal three-state bus. As a result, it is more common to use multiplexers to implement an arbitrated bus, as multiplexers are supported by all PLDs.

The Avalon bus is a "switch fabric" used by Altera's SOPC Builder to interconnect processors and other devices in a Nios embedded processor system [8], and is not actually a bus in the traditional sense. Specifically, the Avalon bus is a point-to-point implementation of a "shared" bus with support for simultaneous multiple bus masters [6]. In other words, there is a dedicated connection from each potential bus-master to each of the slave devices that it can master. Although each processor and device appears to connect to a real bus, there are no shared lines in the system. This structure is illustrated in Figure 3.5 for an *N*-processor system.

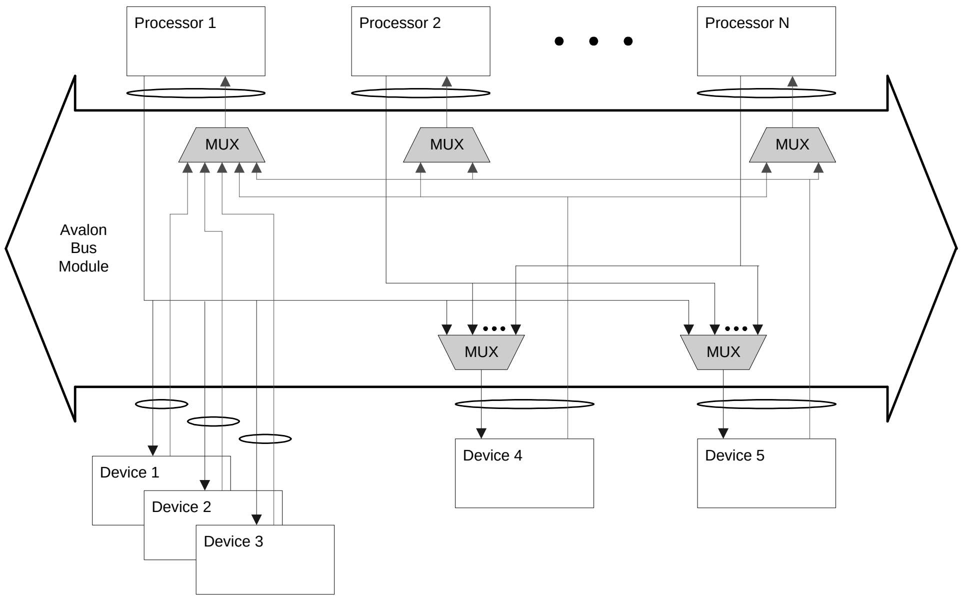

Figure 3.5: Basic Structure of an Avalon Bus Module

Consequently, the multi-master architecture increases system bandwidth by eliminating the bottleneck of a single bus. System masters contend for individual slaves, not for the bus itself. This technique is called slave-side arbitration, and it makes the protocol flexible enough for high bandwidth peripherals. Slave-side arbitration means that any number of transactions may occur simultaneously, as long as there is no contention for the same slave. If more than one master requests the same slave, each master is granted access in turn, either in the default round-robin fashion or using a configurable priority scheme. This arbitration is encapsulated within the Avalon bus module, and is hidden from the system designer (though the arbitration rules are configurable through SOPC Builder). Once access to a slave has been granted, Avalon bus multiplexers feeds the appropriate signals to the slave. Figure 3.6 shows the use of multiplexers in an example system of two masters and two slaves.

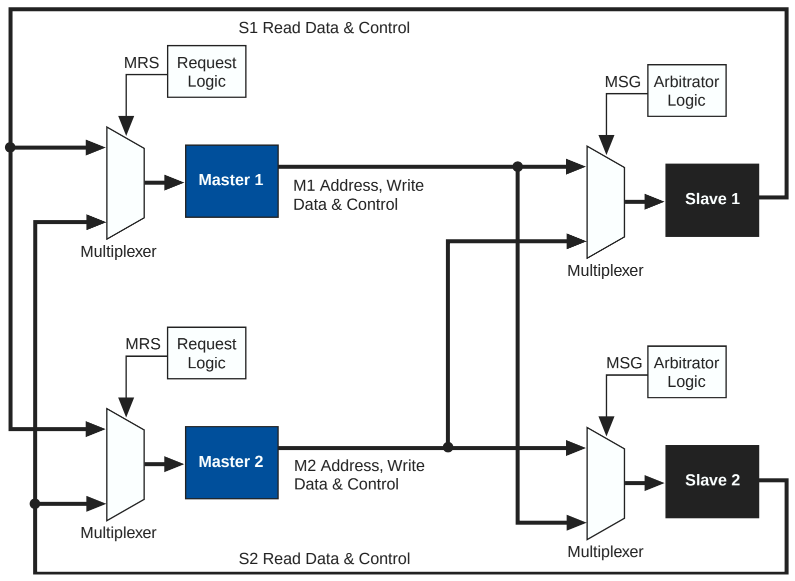

Figure 3.6: Avalon Multiplexers Routing Signals

As can be seen, the Avalon bus also specifies separate address, data, and control lines. This provides an easy interface to on-chip user logic, avoiding the need to decode data and address bus cycles. Additionally, the Avalon bus uses dynamic bus sizing. In other words, the address and data busses to each slave peripheral are only as large as they need to be. For example, a slave with only four accessible registers would have an address width of two. Dynamic bus sizing means that the Avalon bus module also automatically handles data transfers between devices of different data widths. Additionally, the Avalon bus module automatically handles wait-state generation, latent transfers, and interrupt generation (as mentioned in Section 3.3).

Transactions on the Avalon bus may occur in byte, half-word, or word sizes (eight, sixteen, or thirty-two bits, respectively). A transaction may begin immediately after another transaction, with no clock cycles wasted, regardless of the master-slave pair. The protocol also defines bus transactions for latency-aware peripherals, streaming peripherals, and multiple bus masters. Each of these advanced transfer modes allow multiple units of data to be transferred during a single bus transaction (reducing overhead when moving large amounts of data).

The Nios uses memory-mapped I/O to access memory and peripherals on the Avalon bus (the Nios processor, associated slave peripherals and Avalon bus are collectively referred to as the *system module*). The Nios uses the full 4 GB (32-bit) address space, presenting an address that the Avalon bus module decodes into a slave select signal and an offset.

3.5 Summary

The preceding descriptions of the salient features of the Nios processor and Avalon bus interface, coupled with the presentation of SMP systems in Chapter 2, allows for a careful analysis of the issues facing the proper operation of an SMP Nios system. In the next chapter, these issues are detailed and analyzed to develop a prototype SMP Nios system.

Chapter 4

Prototype Cache Coherency Module

Before an SMP Nios system can be implemented, the issues facing proper operation must be raised and addressed. The greatest challenge to implementing a high-performance SMPOPC system is enforcing cache coherency: typical softcore processors available for constructing such a system are not designed with cache coherency in mind. Specifically, the bus architecture typically used in PLDs (such as the Avalon bus) effectively makes snooping impossible. Thus, other features are required to achieve cache coherency. In this chapter, the problems facing proper cache coherency enforcement are discussed, and a general system architecture addressing these problems is outlined. Furthermore, an initial prototype cache coherency module (CCM) is developed as a proof-of-concept that cache coherency can be maintained in an SMP Nios system.

The goal of the CCM is to enforce cache coherency with a minimum of alterations to existing vendor-provided IP. This requires a careful examination of the Nios and the Avalon bus module, to understand which features will facilitate, and which features will hinder, cache coherency. It is also advantageous to make the process of instantiating a cache coherent SMPOPC as seamless and transparent to the user as possible, with little to no deviation from existing system generation processes. This prototype serves as a proof-of-concept that the system can be easily modified to enforce cache coherency.

4.1 SMP Issues in Programmable Logic

Symmetric multiprocessing on a programmable chip involves the implementation of multiple softcore processors on a single programmable logic device. Modern programmable logic devices provide sufficient resources (LEs and on-chip memory) to implement complex systems of 32-bit softcore processors with cache support. Development tools such as SOPC Builder provide direct support for implementing multiple softcore processors on a programmable chip. However, development tools do not yet provide a way to automatically implement a functioning SMP system.

SMPOPC systems are architecturally identical to their discrete SMP counterparts. This includes having identical processors, each with equal access to memory and I/O subsystems. In an SMPOPC system, these requirements are fulfilled using the system-development tool to instantiate processors with identical features. These processors must be specified to each have a connection with equal arbitration priority to each I/O peripheral and memory device. Even when fulfilling these requirements, two issues currently prevent full working of SMPOPC systems: (i) there is no way to uniquely identify the processors in a system, and (ii) enforcing cache coherency. Cache coherency is the most significant barrier to symmetric multiprocessing on a programmable chip. Custom hardware and software development is necessary to ensure cache coherency.

4.1.1 Uniquely Identifying Processors

Some way to uniquely identify processors is needed, as a way to temporarily select a bootstrap processor to execute global initialization on startup, and to allow operating systems to assign processes and threads to specific processors. One aspect of global initialization is setting up the shared interrupt vector table. Local per-processor initialization includes enabling interrupts, setting the interrupt priority mask, clearing and enabling caches, *etc*. In traditional SMP systems, a motherboard often identifies each processor according to the physical socket in which it resides. In a PLD, however, physical sockets do not exist.

While the Nios processor does have a `CPU ID` control register, this read-only register returns a code that is unique only to the particular version of the Nios. Therefore, each Nios of the same version returns the same ID. Several solutions exist: (i) changing the value of the `CPU ID` control register; (ii) adding a control register to the Nios; (iii) implementing a small ROM for each processor, containing a unique processor ID (PID); and (iv) implementing a custom instruction in each Nios to return a unique value.

The first two solutions do not fall within the goal of being non-invasive to the Nios. The fourth solution is needlessly complicated for a simple problem that would consume one of only four available custom instruction opcodes. Therefore, the ROM was chosen, since in addition to being simple and non-invasive, it takes advantage of the Avalon bus architecture to make each ROM accessible to only its corresponding processor. This exclusivity also allows all the ROMs to be assigned the same address, thus conserving address space.

4.1.2 Comments on Cache Coherency with an Avalon Bus

The use of the Avalon bus (and other similar PLD bus architectures) effectively prevents the use of bus snooping protocols to implement cache coherency, since the bus is not physically shared. A non-trivial amount of hardware re-development would be necessary to build a device capable of monitoring every set of primary bus connection points. These primary bus connection points are denoted by ovals in Figure 3.5. Hence, cache coherency is a very relevant problem to solve in this context. Rather than modify the tool used for system generation or modify the structure of the Avalon Bus, another solution was sought, as described below.

4.2 Architecture

The first architectural design decision is whether to implement a snooping or a directory protocol. A directory protocol could be used, but it is not as effective as a snooping protocol for small-scale systems, as message passing either requires a dedicated bus (high hardware cost), or consumes additional bandwidth on the already-congested system bus. Either implementation requires invasive changes to each Nios processor so that its cache can send, receive and understand the directory protocol messages. Such a protocol would also incur a large hardware cost in the form of the central directory.

Alternatively, a snooping protocol could be used. At the architectural level, there are a number of places that snooping hardware can be placed. The Nios processor implements a pair of instruction and data caches with a write-through policy [11]. Traditionally, cache coherency is enforced by creating a hardware module for each cache that monitors the processor's memory bus. This, unfortunately, is not possible due to the point-to-point nature of the Avalon bus (see Section 4.1.2). Thus, a snooping architecture cannot be used.

An alternative is to add a slave peripheral to the system module to inform processors of a memory write. Implementing cache coherency through a slave peripheral allows system developers to simply instantiate a CCM using the standard system generation GUI. It is also easy to implement, as the Avalon bus is an interface specification with well-defined signals. This is, in reality, a *hybrid snooping protocol*, that snoops the bus but uses a central "directory" to enforce coherence. The slave peripheral can be given access to the relevant signals on various Avalon bus interfaces. These interfaces can be standard interfaces to peripherals, such as on-chip RAM or a memory controller, or special interfaces such as a tri-state bridge, which is used to communicate with off-chip SRAM and flash memories.

Figure 4.1 shows the CCM in relation to a typical *N*-way SMP Nios system. The CCM must be able to detect writes (typically by monitoring write enable signals), as well as read the address bus. This allows the module to notify the processors of an address that has been written to, so that the appropriate cache line can be invalidated.

The reason why the cache line must be invalidated, as opposed to updated (see Section 2.2) is that the Nios has the native ability to invalidate particular cache lines, but not to update them. The invalidation is performed by writing the appropriate address to specific control registers implemented in each Nios processor. The invalidate policy was selected in the interest of minimizing invasive changes to the system. The cache coherency protocol used is depicted by Figure 4.2.

The implementation of cache clearing through processor control registers requires that software play a role in maintaining coherency. Due to the importance of maintaining coherency, the software component was written in the form of a high-priority interrupt service routine. This is a perfect match for the ability of a slave peripheral to raise interrupts. Thus, enforcing cache coherency is a marriage of hardware and software.

4.3 Hardware Cache Coherency Module

The cache coherency module is responsible for detecting when a memory write has occurred, and notifying processors of such an event. The VHDL code for the CCM hardware is listed in Appendix B. Figure 4.3 shows the corresponding schematic diagram.

The Nios processor must have the ability to enable and disable the CCM. This is required as there are situations where the CCM must not raise an interrupt (one situation is before initialization is finished, where the caches are enabled and interrupt vector table is set). A single bit `CONTROL` register is used to disable operation (highlighted by oval 3 in Figure 4.3). The `CONTROL`

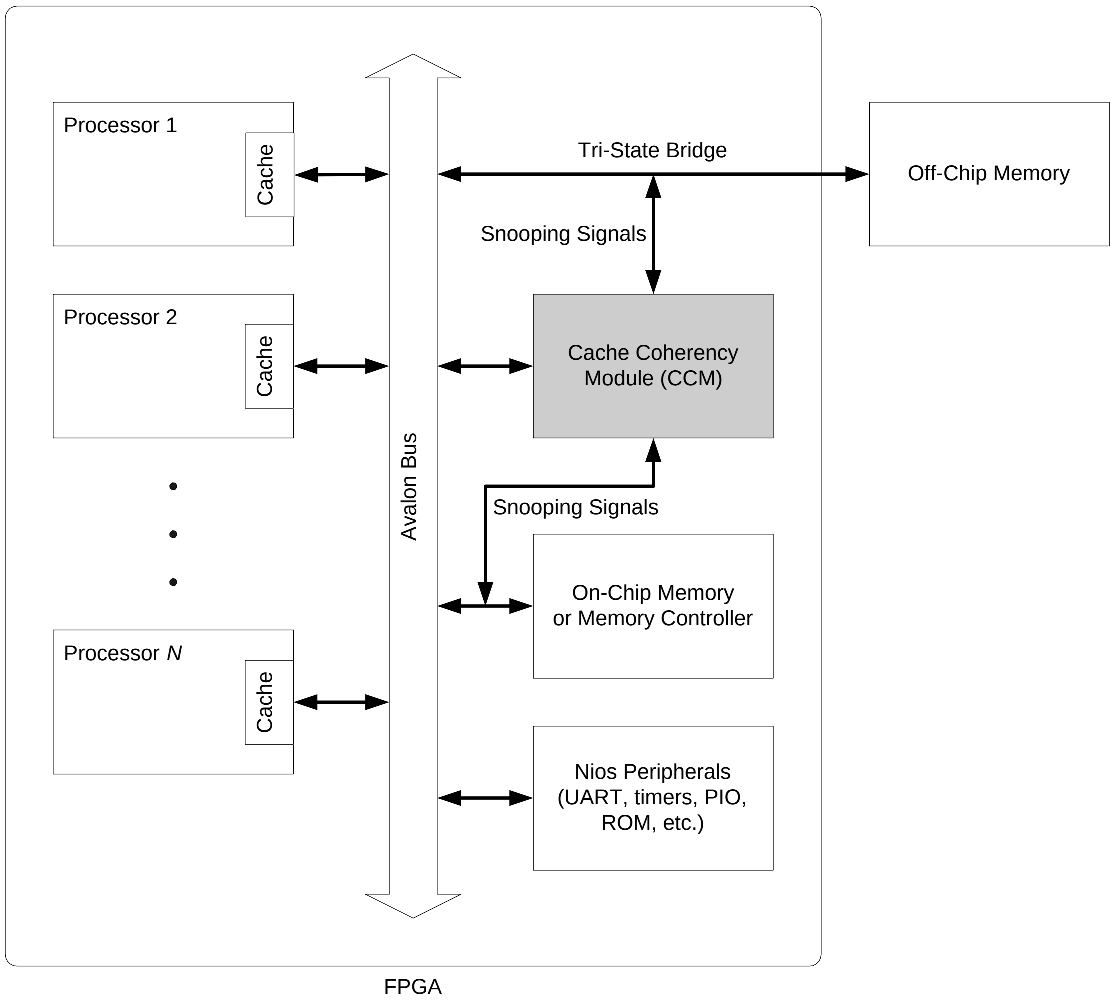

Figure 4.1: System Architecture with a Cache Coherency Module

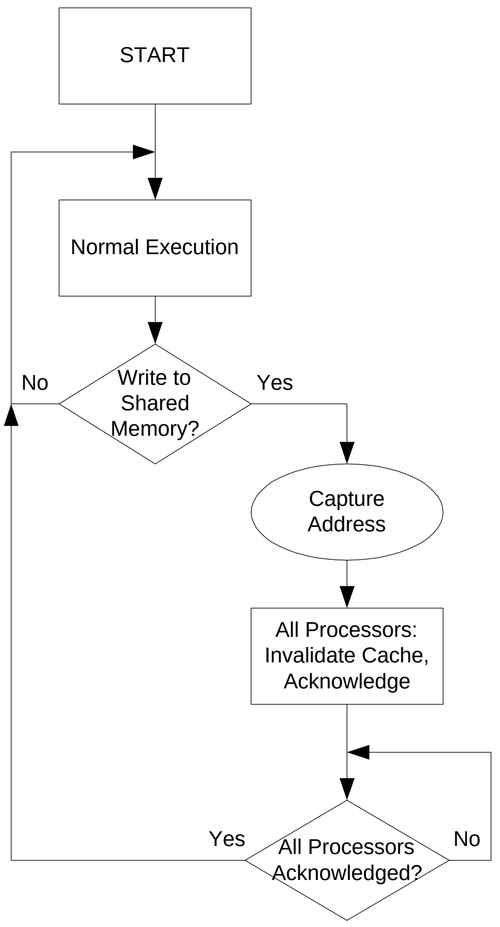
START
Normal Execution
Write to Shared Memory?
No
Yes
Capture Address
All Processors: Invalidate Cache, Acknowledge
All Processors Acknowledged?
Yes
No

Figure 4.2: Cache Coherency Protocol

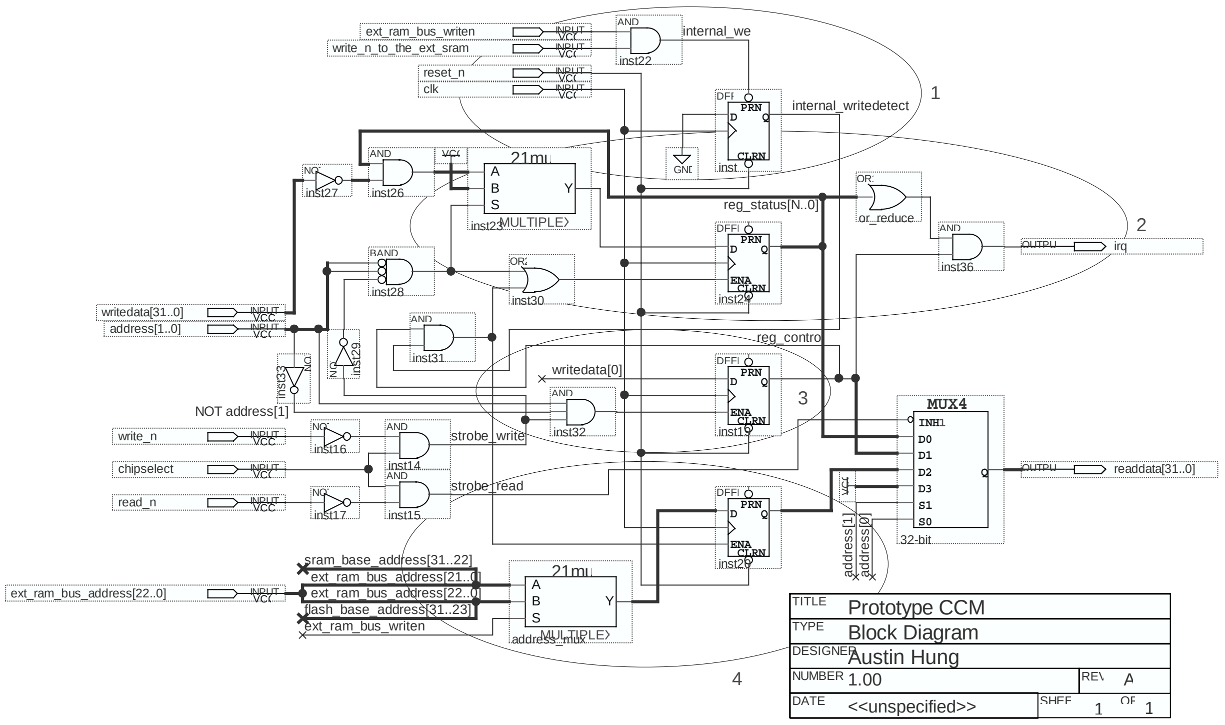

Figure 4.3: Prototype CCM Schematic

register uses the system clock and reset signals, and is reset to logic ‘0’ (disabled). It is assigned to register offset 0x01, and any processor may write any value to it.

Snooping the bus effectively requires the ability to detect a write transaction and capture the corresponding memory address. The CCM accomplishes this task by snooping the write enable and address lines on the bus of every memory device (recall that there is no single bus for all devices in the system), and asserting a write detect signal (oval 1). If the CCM is enabled, the write detect signal also causes the STATUS register (oval 2) to be set (one bit per processor in the system). There are two types of devices to snoop: asynchronous off-chip memories and Avalon bus slave interfaces.

For asynchronous off-chip memories using the Avalon tri-state bridge, the CCM implements a write detect mechanism by asynchronously setting a D flip-flop when a write enable signal is asserted. This flip-flop uses the system clock and reset signals. The input is tied to logic ‘0’, such that the following rising clock edge resets the write detect. The setting of the flip-flop is asynchronous because not all memory devices have a synchronous write enable signal. The tri-state bus, for example, asserts the SRAM write enable in between rising edges of the clock. A synchronous set would be unable to detect a write.

Conversely, for memories (on- or off-chip) that are accessed via an Avalon bus slave port (*i.e.*, on-chip RAM and off-chip synchronous dynamic RAM - SDRAM), the Avalon bus provides a well defined synchronous interface to snoop. In this case, the write detect is simply a wire that follows the synchronous Avalon write enable signal, saving a flip-flop.

The address lines are registered every clock cycle, using the write detect as a clock enable. Using this timing guarantees that the proper address is registered for most memories, whether synchronous or asynchronous, and regardless of technology. This is because the write enable signal typically triggers the write (asynchronously, or is read on a clock edge), and thus the address must already be on the bus to comply with setup timings. One needs only refer to the device timings of various memories to verify this behaviour. For example, AMD's AM29LV065D [14] asynchronous flash memory, IDT's IDT71V416 [24] asynchronous SRAM memory, or Micron's MT48LC4M32B2 [28] synchronous SDRAM memory.

Thus, the `ADDRESS` register (oval 4) only records an address on the rising edge following a detected write enable signal. The address lines, however, only provide the offset for the particular memory device, which does not correspond to the address used by the processor and cache. The address is converted to a useable form by being logically ORed with the memory device's base address prior to being registered in `ADDRESS`. The SOPC Builder tool requires that all peripheral address spaces be aligned with the slave's address range [7]. As a result, the low bits that the slave's offset might occupy are guaranteed to be 0, which allows the full address to be constructed by a logical OR operation. Once an address is captured in the `ADDRESS` register, it is available for processors to read so that the corresponding cache line may be properly invalidated.

One requirement of a Nios system is that each processor must acknowledge any interrupt that it is servicing. The Nios processor acknowledges interrupts in an ISR, guaranteeing that the ISR is being serviced. The CCM must track which processors have had the opportunity to acknowledge the interrupt (and have therefore cleared their cache), and de-assert `irq` signal only when all processors have completed the invalidate operation. The `STATUS` register serves just such a purpose.

The N-bit `STATUS` register is a one-hot encoding of the processor ID (*i.e.*, bit 0 corresponds to processor 0, bit 1 corresponds to processor 1, *etc*.). N is the number of processors present in the system. The register is logical OR reduced to produce an interrupt request signal (`irq`), notifying all processors that a write has occurred. The `irq` signal is also masked by the `CONTROL` register. A processor acknowledges the interrupt by writing the one-hot encoding of its PID, thereby resetting the corresponding bit in `STATUS`. When all processors have acknowledged the interrupt, the OR reduction causes the `irq` signal to be de-asserted. A read of the `STATUS` register indicates which processors have not yet acknowledged the interrupt. With this acknowledgement scheme, the hardware cost is incremental (*i.e.*, only one additional flip-flop per processor). This makes the system trivially scalable up to thirty-two processors, the data width of the CCM's Avalon bus interface.

The CCM includes miscellaneous hardware for reading and writing peripheral registers via the Avalon bus interface. Table 4.1 describes these registers and their offsets.

Register	**Offset (Hex)**	**Width (Bits)**	**R/W**
`STATUS`	00	N	Read-write
`CONTROL`	01	1	Read-write
`ADDRESS`	02	32	Read-only

Table 4.1: Prototype CCM Registers

It should be noted that systems with greater than thirty-two processors can be supported with minor changes to the CCM, at the cost of additional hardware. Each processor can instead write its binary encoded PID to the `STATUS` register, which is in turn decoded to set the corresponding bit in an internal N-bit register. The internal register may be of arbitrary length, since it is not constrained by the width of the data bus.

4.4 Interrupt Service Routine

Figure 4.4 shows the assembly code for handling an ISR for the prototype CCM. The constants `na_ccm`, `np_ccmaddress`, and `na_ccmstatus` represent the base address of the CCM and the register offsets for `ADDRESS` and `STATUS`, respectively. These constants have been defined in include files.

```
nr_ccmisr: pfx %hi(0x20)  ; MOVIA %l0,na_ccm
        movi %l0,0x0
        pfx %hi(0x80)
        movhi %l0,0x10
        pfxio %hi(0x0)    ; address = na_ccm->np_ccmaddress;
        ldp %l6,[%l0,np_ccmaddress]
        rdctl %l5         ; nm_caches_disable();
        movhi %l5,0x0
        wrctl %l5
        nop
        pfx %hi(0xa0)     ; nm_icache_invalidate_line(address);
        wrctl %l6
        pfx %hi(0xe0)     ; nm_dcache_invalidate_line(address);
        wrctl %l6
        rdctl %l5         ; nm_caches_enable();
        movhi %l5,0x3
        wrctl %l5
        nop
        pfx %hi(0x0)      : %l1 = _cpuid
        movi %l1,0x0
        pfx %hi(0xa0)
        movhi %l1,0x00
        movi %l5,0x1      ; %l5 = 1
        ldp %l7,[%l1,0x0] ; %l7 = *_cpuid
        ext16d %l7,%l1
        lsl %l5,%l7       ; na_ccm->np_ccmstatus = 1 << (*_cpuid);
        stp [%l0,np_ccmstatus],%l5
nr_ccmisr_loop: pfxio %hi(0x0)
        ldp %l5,[%l0,np_ccmstatus]
        skprz %l5
        br nr_ccmisr_loop
        nop
        tret %o7
```

Figure 4.4: Prototype CCM ISR

The ISR begins by retrieving the memory address to be cleared from the CCM `ADDRESS` register (lines 1–6). It then disables both caches (lines 7–10) and invalidates the appropriate cache line (lines 11–14), and then re-enables the caches (lines 15–18). Finally, it acknowledges the interrupt by writing the one-hot encoding of its PID to the CCM `STATUS` register (lines 19–27), and spin locks (lines 28–32) until the `STATUS` register is fully cleared (which occurs when all processors in the system have acknowledged the interrupt). Finally, the ISR returns to the interrupted program.

The ISR for the prototype CCM is thirty-three instructions long. This includes three accesses to the CCM's Avalon slave port and one to the PID ROM. Accessing the PID ROM does not cause any contention on the bus, since each processor has exclusive access to the memory over an unshared bus.

The issue of self-modifying code now needs to be addressed. Self-modifying code consists of a set of instructions that is modified by the software itself to achieve some task. The consequence of allowing self-modifying code is that the separate instruction cache may now contain incoherent instructions. As a result, the written memory may be either instructions or data, so both caches must be invalidated (since there is no way to determine whether the written value represents an instruction or data). Prohibiting self-modifying code reduces the number of instructions in the ISR by two (lines 11 and 12), and allows the instruction cache to remain enabled (which results in performance improvements if any ISR code happens to be in the instruction cache).

As mentioned in Section 3.3, the Nios provides interrupt vectors 16 to 63 for the user, making vector 16 the highest priority interrupt number available for the CCM to use, and that is the number assigned to it. This is acceptable for the following reasons:

- The hardware debug module and internal hardware exceptions should have higher priority over clearing the cache, thus interrupt vectors 0–2 may take precedence over the CCM.
- It is acceptable to allow a debugger to have higher priority to capture all system behaviour,

including any CCM ISRs, thus interrupt vectors 3–5 may take precedence over the CCM.

- The rest of the interrupts (6–15) are currently unused.

Thus, it is important that the system designer ensure that the CCM be given interrupt number 16 when configuring the system.

4.5 Software Requirements

The software requirements for maintaining cache coherency are quite simple. In general, they involve creating an initialization barrier point after per-processor initialization for all APs (processors not designated as the BSP). This causes the APs to wait until global initialization is complete. This is similar to the Intel SMP method of holding APs in reset until the BSP has completed initialization [25]. In this case, global initialization is comprised mainly of installing the ISR and enabling the CCM.

If the default `_start` per-processor initialization routine [12] is used, then the BSP must also have a barrier to ensure that all APs have reached the initialization barrier point. This is because the per-processor `_start` routine performs some global initialization, such as clearing the interrupt vector table. If the BSP installs the ISR (in the global initialization routine) before all APs complete the `_start` routine, an AP may clear the interrupt vector table, leading to the spurious interrupt handler being called on a CCM exception, rather than the ISR. This second barrier point can be implemented by a long busy-wait loop (the length of which scales with the number of processors in the system). Alternatively, the programmer may elect to provide a custom `_start` routine with all its global initialization tasks moved to the global initialization routine. This foregoes the need of a bootstrap barrier. In the tests used to validate the system, the provided `_start` routine is used. For production systems, however, it is recommended that a more efficient custom routine be provided.

Finally, each Nios has frame and stack registers pointing to private per-processor stack memory. Since stacks are private, these pointers must be managed in such a way that processors do not accidentally overwrite any private data in other processors' stacks. This management is the responsibility of the application or operating system.

4.6 Design Flaws

While this CCM prototype proves that it is possible to detect a write and invalidate cache lines (see Chapter 6 for details), there are a number of flaws and inefficiencies related to this configuration. One drawback is that all processors clear the specified cache line, whether it contains the actual address or not (*i.e.*, the Nios does not check the cache line tag before resetting the valid bit). This is a limitation of the Nios processor and the way it clears the cache.

Another limitation is that since there is a single shared interrupt line to all processors, all processors execute the ISR. This includes the writing processor. Additionally, the writing processor will clear its cache, even though it contains the current and correct value. Thus, the next read to that location causes a cache miss, increasing latency. This could be rectified by determining which processor made the write and not setting its `STATUS` bit. Assuming the one-hot `STATUS` register scheme is used, this would allow the ISR to check the appropriate `STATUS` bit and skip the cache clearing stage if not set for that particular processor. This capability can be enabled by snooping the internal, unpublished Avalon bus signals, however this may not be a good idea as Altera does not guarantee any particular implementation for the internals of the Avalon bus module.

The system memory bus becomes a bottleneck when an interrupt is raised as all the processors in the system attempt to fetch the ISR code simultaneously, since the Avalon bus does not provide facilities for serving multiple masters requesting the same address (broadcasting the

ISR). This leads to a long latency that scales with the number of processors since all processors must acknowledge the interrupt before any can proceed. This can be mitigated by storing the ISR code in multiple shared on-chip ROMs (enabling multiple simultaneous access to the ISR code). The latency can be reduced by a factor of up to N by duplicating the ROMs, up to one for each processor (since at that point the ROM is no longer shared).

The single shared CCM Avalon slave port may also be a bottleneck, as each processor must access the CCM a minimum of three times per interrupt. This bottleneck may be mitigated or eliminated by creating more identical slave ports that are shared among a smaller number of masters (or even creating one port per Nios).

Finally, the major design flaw in the prototype CCM design is that it does not handle multiple stores that are close together. Even though the CCM raises an interrupt as soon as it detects a write, it does not consider multiple store instructions in a pipeline (instructions currently in the pipeline are "in-flight"). An interrupt in the Nios system allows all in-flight instructions to complete before executing the ISR, therefore a number of stores may execute prior to the interrupt being serviced. With the prototype CCM, only one address is stored to the `ADDRESS` register per interrupt. Thus, if multiple stores occur, only the most recent store is cleared from the processor cache, which leads to cache coherency problems with the prior stores. A worst-case test was written (see Appendix E) to expose the problem and quantify how many consecutive stores could be executed before the interrupt routine was executed. Given a five-stage pipelined design, the Nios should not complete more than five instructions before beginning instruction fetches for the ISR. The resulting waveforms from the test are shown in Figure 4.5.

The waveform is the output of Altera's SignalTap II embedded logic analyzer [13]. The test platform is a dual Nios system, with SRAM serving as program and data memory. Monitored signals include the off-chip tri-state bridge (address lines - `ext_addr`, SRAM chipselect - `SRAM_ce_n`, and SRAM write enable - `SRAM_we_n`), internal CCM signals (`internal_we` and

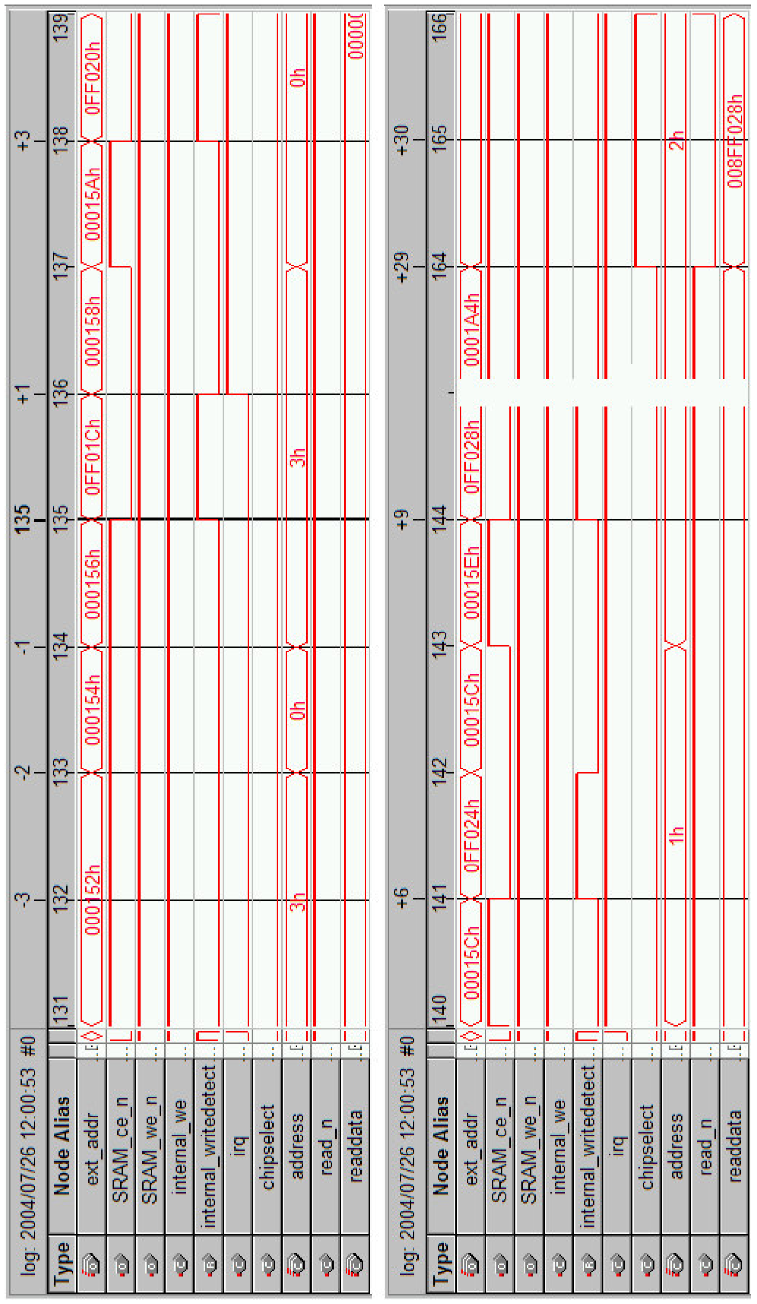

Figure 4.5: Multiple Store Instructions with the Prototype CCM

`internal_writedetect`), and CCM Avalon slave port signals (`irq`, `chipselect`, `address`, `read_n`, and `readdata`). The logic analyzer is triggered by the rising edge of the `irq` signal, which indicates that the CCM has detected a write.

The upper waveform shows the first two of four writes to memory. Time is measured in clock cycles. At times 132, 133, 134, and 136, store instructions are read from the instruction cache. A write to SRAM address 0x0FF01C occurs at time 135, as indicated by the CCM's `internal_writedetect` signal. At time 136, an interrupt is raised and the address on the bus is captured. Further writes occur to SRAM addresses 0x0FF020, 0x0FF024, and 0x0FF028 at times 138, 141, and 144, respectively. Recall that the Avalon bus provides decoding services, and thus only offsets reach the selected peripheral or memory.

The lower waveform continues from the upper waveform, and shows the last two writes and the Nios processors fetching from the CCM the address to be invalidated. At times 164 and 165, each of the two processors read the CCM `ADDRESS` register (at address 0x02), which returns 0x008FF028 (the full address of the written location). The rest of the ISR is executed, and at time 238 (not shown), `irq` is de-asserted. At the end of the ISR, only a single address, 0x008FF028 (offset 0x0FF028) has been invalidated. The first three addresses in the series have not been invalidated, and the system has no record of those writes having occurred.

Though the CCM suffers from this critical problem, the prototype proves that cache coherency can be enforced in an SMP Nios system. Luckily, the problem can be resolved if all the writes can be captured, instead of just the last one. That is the driving reason behind the second-generation CCM.

4.7 Summary

This chapter has raised and addressed specific issues related to implementing an SMP Nios system, including unique processor identification and cache coherency. A proposed architecture for implementing a hybrid snooping protocol is described, and a prototype cache coherency hardware-software solution is presented. This prototype was tested to prove that cache coherency could be maintained, but failed to handled the case of multiple in-flight writes. Additionally, a number of performance limitations were identified. In the next chapter, a second-generation CCM design is presented to address the critical flaw and improve performance.

Chapter 5

Second-Generation Cache Coherency Module

To address the major design flaw of the initial CCM prototype, a new module was designed. While the system architecture shown in Figure 4.1 remains unchanged, the CCM internals are vastly different. Consequently, the interrupt service routine was refined appropriately, while the software requirements remain unchanged (since the basic operation of cache coherency enforcement remained static). In addition to addressing the major design flaw, a number of other improvements were made to reduce cache clearing overhead.

5.1 Hardware Cache Coherency Module 2

Supporting a series of writes requires that each address in the series must be captured. As a result, a single 32-bit `ADDRESS` register is no longer sufficient, and memory must be used instead. A first-in-first-out (FIFO) memory with independent read and write ports is an ideal candidate for storing multiple addresses. A write to the FIFO can be triggered by write detect circuitry similar to that used in the prototype, while a read from the FIFO can be initiated by the Avalon bus slave port circuitry. A FIFO can only write a single value at a time, while multiple writes may occur due to the system instantiating a single Avalon bus for each memory device or bridge. Thus, a FIFO is required for each interface.

In addition to fixing the critical flaw, other hardware changes could facilitate a reduction in overhead when dealing with a CCM interrupt. For example, the Avalon bus slave port can be reproduced to remove it as a source of contention as all processors rush to retrieve invalidation addresses. New hardware circuits are needed to adapt the CCM to these changes.

The VHDL code for the second-generation CCM hardware is listed in Appendix C. Figure 5.1 shows the block diagram of the new CCM for a two-processor system with off-chip flash and SRAM. Due to changes in operation and the replication of hardware, the overall design has been highly modularized. The CCM is now composed of three main modules: Avalon bus slave port modules (the two large blocks in the upper left of Figure 5.1), a central register module (the block on the right), and bus snooping modules (the two bottom blocks).

Very little hardware exists outside the modules. The hardware that is outside the modules serves a supporting role, typically for conversion and selection functions. First, each bus snooping module has an input that is the address that is being written. The memory device address bus value is logically ORed with the memory device's base address to calculate the real address. Also, each snooping module has its own write detect register, which functions similar to the write detect circuitry in the prototype CCM.

The `all_read` signal is the logical AND of all `address_read` signals from the slave interface modules. A logic '1' on the `all_read` signal indicates that all processors have invalidated the current address. When this signal is asserted and the CCM has been enabled, this asserts the `new_address` signal, which instructs the CCM to register the next address (if one is available) into the `ADDRESS` register. If more than one snooping module has available addresses, then the

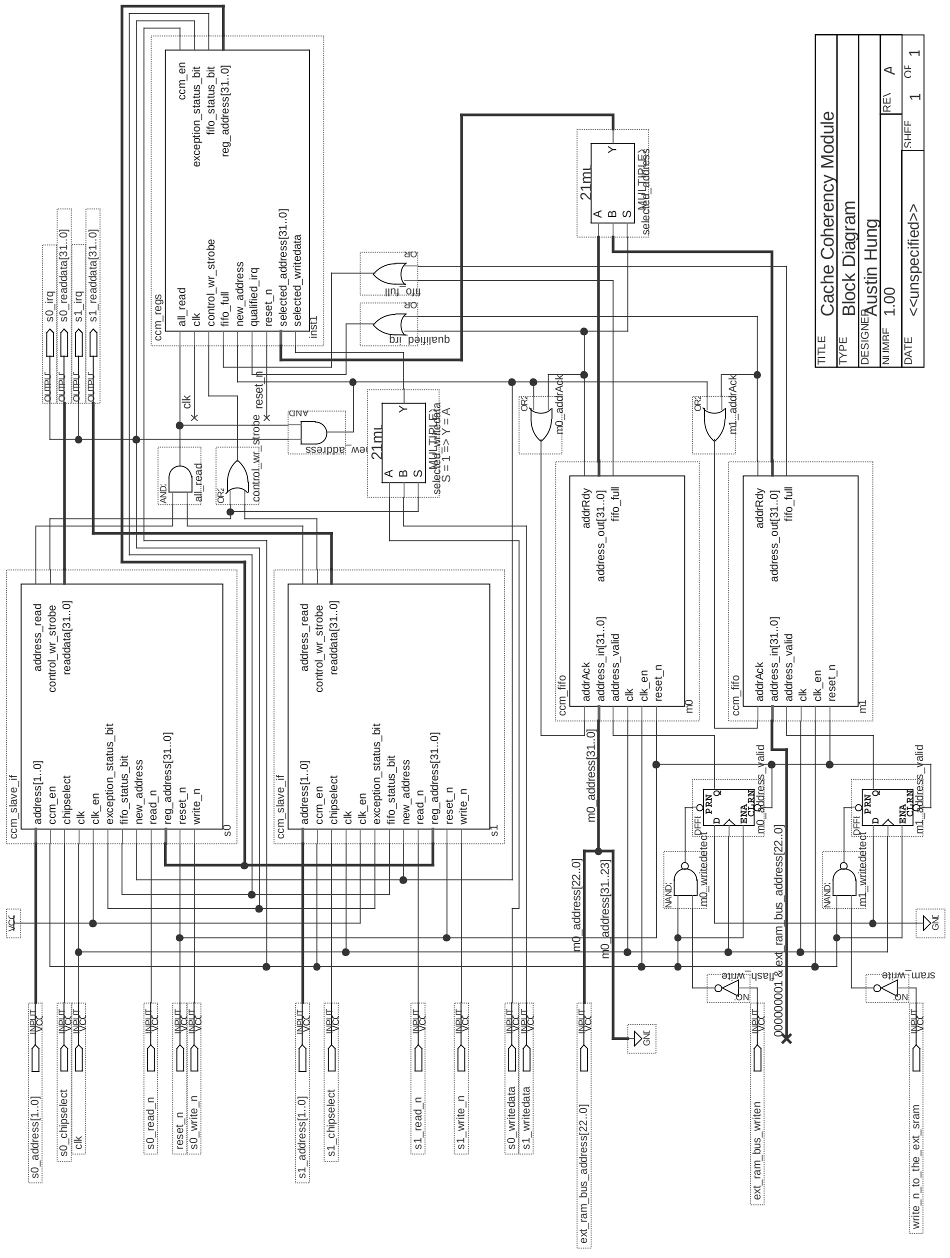

Figure 5.1: Second-Generation CCM Schematic

selection logic chooses the address from the lowest snooping module number[1]. This selection method is simple. Since all addresses get serviced during the same interrupt, priority is unimportant. If `new_address` is asserted and an address is available (as indicated by the `addrRdy` signal) then the `addrAck` signal is asserted for the chosen snooping module, which causes the address to be placed into the `ADDRESS` register and the `address_read` signal to be de-asserted. Each snooping module indicates that it has available addresses by asserting its `addrRdy` signal. The `qualified_irq` signal is a logical OR of all `addrRdy` signals, indicating that one or more snooping modules contains at least one address that has not yet been invalidated.

Selection logic also exists for writing to the `CONTROL` register. Given simultaneous writes to this register (which is possible due to multiple slave interfaces), the logic simply chooses to write the value from the lowest slave interface port number[2]. This scheme is used for its simplicity, and also because writes to the `CONTROL` register should be rare, with simultaneous conflicting writes practically impossible.

5.1.1 Slave Interface Module

The internals of the Avalon bus slave port module (hereby referred to as the slave interface) are shown in Figure 5.2. This module serves as the interface between the Avalon bus and the internal CCM registers. There is one slave interface for each Nios processor in the system, eliminating bus contention when accessing the CCM. The slave interface has three functions:

1. to place the requested data on the bus,
2. to write to the CCM enable bit, and
3. to indicate when the master has read an address for invalidation.

[1]The snooping module numbers are assigned arbitrarily.
[2]Slave interface port numbers are also assigned arbitrarily.

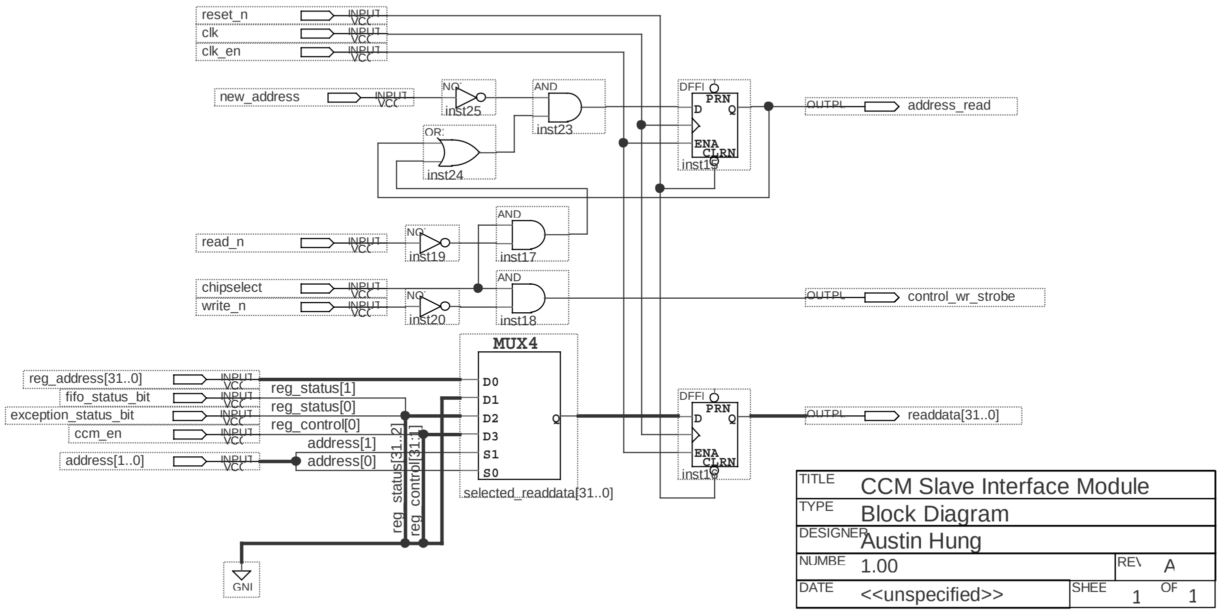

Figure 5.2: Avalon Bus Slave Port Module

The slave interface module includes hardware for interfacing to the Avalon bus, similar to that of the prototype CCM. The internal `CONTROL` and `STATUS` registers, however, have been moved to the centralized register module to avoid unnecessary duplication. A read from the registers uses the central register as a source. A `control_wr_strobe` signal indicates a write to the `CONTROL` register and simple priority logic in the CCM writes to the central register.

Instantiating a slave interface for every processor not only eliminates a potential bottleneck, but also allows a redesign of the interrupt acknowledgement. Recall that the prototype CCM used one-hot encoded N-bit `STATUS` register, which was OR reduced to produce an interrupt request. Having one port per processor, however, allows each port to track the acknowledgement state of each processor individually, without a central register. Moreover, the interrupt acknowledgement can be done automatically and passively by detecting `ADDRESS` register reads. This new passive acknowledgement scheme reduces the ISR length by nine instructions (including the PID access). Also, this scheme is even more scalable than the prototype scheme, as no change is required to support more than thirty-two processors, and each subsequent slave interface is incremental in hardware costs. The additional hardware required is small because port replication eliminates the Avalon slave arbiters since each bus is now point-to-point, instead of having N:1 multiplexers and the associated arbitration logic.

An `address_read` register serves as a partial acknowledgement signal. In reality, it acknowledges the address (that is, the current address has been read by the processor, and the next address is needed from an address FIFO). On a system reset, this register is set to logic ‘1’, indicating that an address is requested. If the `new_address` signal is asserted, indicating an address is available, the register is reset until such a time as the processor reads the `ADDRESS` register. An interrupt continues until all addresses have been acknowledged by all processors.

5.1.2 Registers Module

The new CCM design saves hardware by using a central register architecture, rather than replicating registers in each of the slave interfaces. This module, shown in Figure 5.3 simply contains four registers: the 32-bit `ADDRESS` register, which holds the current address to invalidate; the `fifo_status_bit` and `exception_status_bit` signals, which together form the `STATUS` register; and the `ccm_en` bit, which represents the `CONTROL` register.

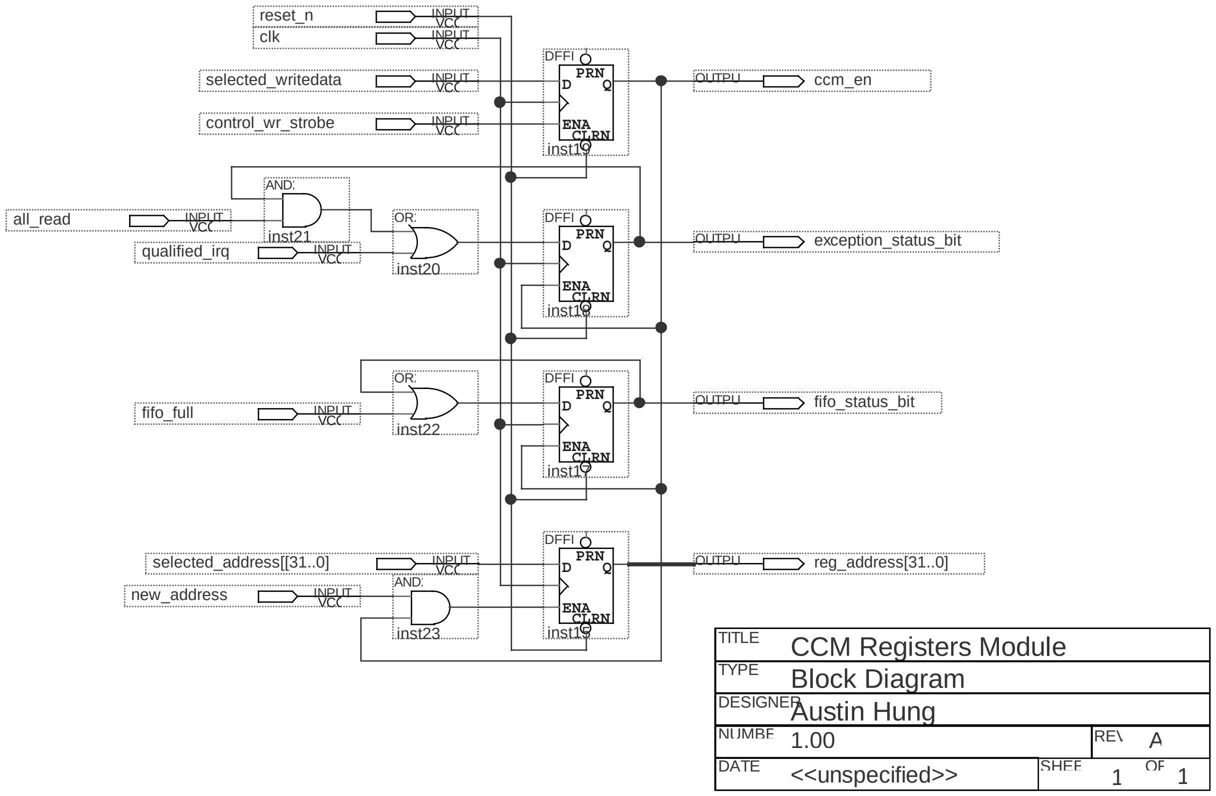

Figure 5.3: CCM Internal Registers Module

The `ADDRESS` register is written to by address selection logic in the parent module, and is read by all the slave interface modules. A new address is written into this register when all processors have acknowledged the current address.

The `fifo_status_bit` is an OR reduction of all the error signals from bus snooping modules, indicating that at least one FIFO has reached or exceeded capacity. The `exception_status_bit` indicates the interrupt status of the CCM, and directly drives the `irq` signals of each slave interface. If the CCM is enabled, then this bit is set when a bus snooping module detects a write, and is reset only when all addresses captured have been acknowledged (*i.e.*, all FIFOs are empty).

Finally, the `ccm_en` bit is set when a processor enables the CCM (as described in 5.1.1. It is cleared on system reset and when the processor disabled the CCM.

5.1.3 Bus Snooping Module

The bus snooping modules are responsible for capturing the addresses of writes to memory devices. One module is instantiated for each internal memory device (on-chip RAM, for example), and for each off-chip memory interface (SRAM or flash on the tri-state bus, for example). The module internals are shown in Figure 5.4.

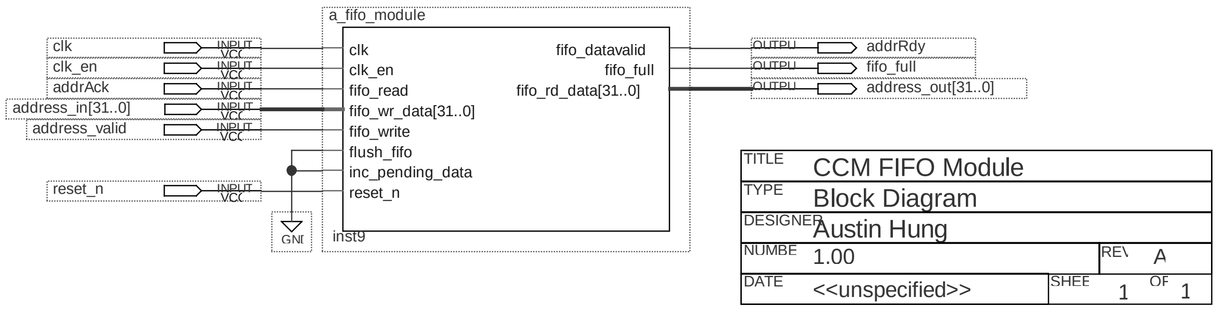

Figure 5.4: Bus Snooping FIFO Module

A bus snooping module is composed primarily of an instantiated Altera 32-bit wide FIFO module, which stores the captured addresses. The bus snooping module essentially renames signals into a more semantically convenient form. Upon a detected write (the `address_valid` signal is asserted), the FIFO captures the address on the bus. Since the FIFO is no longer empty, the FIFO's `datavalid` signal is asserted, indicating that an address exists to be invalidated (this signal is renamed to `addrRdy`). When all processors have acknowledged the current address, then the logic selects a FIFO from which to draw the new address, causing a read (via the `addrAck` signal) to occur on the selected FIFO. When all FIFOs are empty, all `addrRdy` signals are logic '0', indicating no more addresses remain and the interrupt has been handled.

This leads to the question of what depth FIFO should be used. Given the worst-case scenario of up to four pipelined writes per processor (see Section 4.6), a FIFO depth of $4 \times N$ is required to guarantee that no writes will be lost. In practice, since the instruction mix is typically only 15% writes to memory, a system designer could reduce the depth of the FIFOs reasonably safely to save embedded memory bits in the FPGA. If a smaller FIFO capacity is not sufficiently large, the `fifo_full` signal could be used to indicate that a write may have been lost, thus prompting all processors in the system to invalidate every line in the appropriate cache(s).

5.2 Interrupt Service Routine 2

Due to the changes in the CCM architecture, the ISR must change as well. Figure 5.5 shows the new ISR code listing. The first major change is to accommodate the new passive acknowledgement scheme. Since an acknowledge occurs automatically upon reading the `ADDRESS` register, it is possible to avoid accessing the PID, decoding it and writing to `STATUS` register. As a result, a savings of nine instructions (lines 26 through 34 of the old ISR) is realized.

In the prototype ISR, the algorithm disabled the relevant caches, retrieved from the CCM a single address to be invalidated, invalidated the appropriate cache line(s), re-enabled the cache(s) and waited for all processors to acknowledge the interrupt. The new CCM, however, can provide

```
nr_ccmisr: rdctl %l5           ; nm_caches_disable();
        movhi %l5,0x0
        wrctl %l5
        nop
        pfx %hi(0x20)          ; MOVIA %l0,na_ccm
        movi %l0,0x0
        pfx %hi(0x80)
        movhi %l0,0x10
nr_ccmisr_loop: pfxio %hi(0x0) ; address = na_ccm->np_ccmaddress;
        ldp %l6,[%l0,np_ccmaddress]
        pfx %hi(0xa0)          ; nm_icache_invalidate_line(address);
        wrctl %l6
        pfx %hi(0xe0)          ; nm_dcache_invalidate_line(address);
        wrctl %l6
        pfxio %hi(0x0)         ; check CCM STATUS register, bit 0
        ldp %l5,[%l1,np_ccmstatus]
        skp0 %l5,0x0
        br nr_ccmisr_loop
        nop
        rdctl %l5              ; nm_caches_enable();
        movhi %l5,0x3
        wrctl %l5
        nop
        tret %o7
```

Figure 5.5: Second-Generation CCM ISR

multiple addresses to be invalidated during a single interrupt. As a result, the new ISR must be restructured accordingly. The new ISR now disables and re-enables the relevant cache(s) outside the loop. Inside the loop, the ISR continually retrieves addresses and invalidates them until all addresses have been acknowledged by all processors. The assembly instructions required to execute these tasks have not changed from the original ISR, they have simply been re-ordered. The new ISR is only twenty-four instructions long (twenty-two if self-modifying code is disallowed).

5.3 Summary

This chapter has presented a second-generation cache coherency module that, in addition to enforcing cache coherency, also addresses the fatal flaw of the prototype module. This second-generation module was tested to prove that that it can handle the case of multiple writes. In the next chapter, the validation of each CCM design is described, and the impact on the overall system of including each CCM design is examined.

Chapter 6

Results and Analysis

In this chapter, the computing platform used to develop the CCM module is described. The results of the tests used to validate operation of the CCM modules are presented, as is the impact of the CCM on hardware resources and performance.

6.1 Development Platform

Altera provides a number of different development kits to aid users in the design of embedded systems, in particular SOPC systems featuring the Altera Nios softcore processor. The cache coherent *N*-way SMP systems were developed developed on a Nios Development Kit, Stratix Professional Edition, pictured in Figure 6.1.

This development board features a Stratix EP1S40 FPGA, with a capacity of up to 41,250 logic elements (LEs) and 3,423,744 bits of on-chip memory. In addition to the Stratix device, the development board includes the following features:

- a 50 MHz system clock (via socketed oscillator) or external clock input,

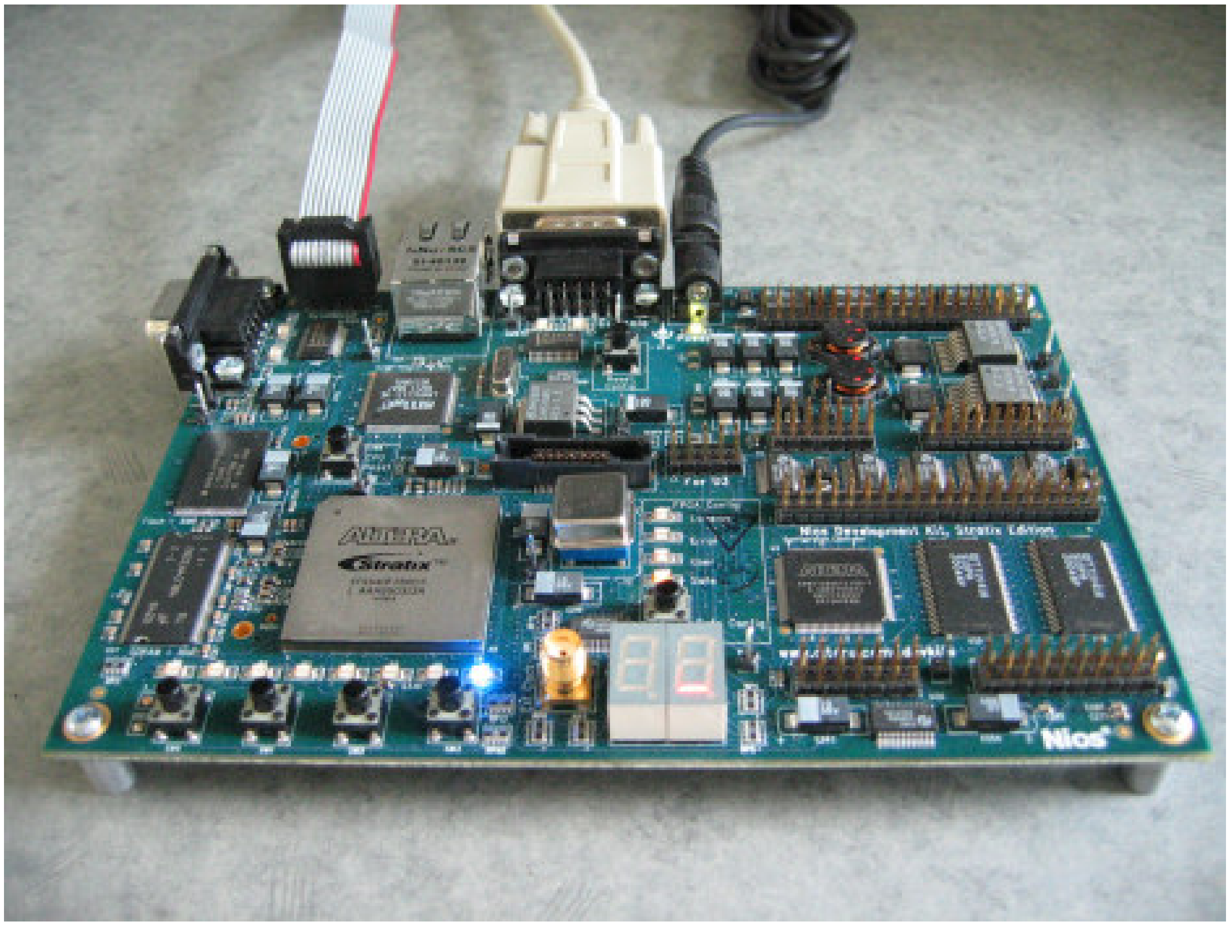

Figure 6.1: Altera Nios Development Kit, Stratix Professional Edition

- SRAM (1 MB in two banks of 512 kB, 16-bit wide, 2×IDT71V416 chips),
- SDRAM (16 MB, 32-bit wide, 1×MT48LC4M32 chip),
- flash (8 MB, 1×AM29LV065D chip),
- a Type I CompactFlash connector,
- a 10/100 Mbit/s Ethernet PHY/MAC controller and RJ-45 connector,
- two serial connectors (RS-232 DB9 port),
- two 5 V-tolerant expansion/prototype headers and a 16×2 LCD module,
- two Joint Test Action Group (JTAG) connectors,
- mictor connector for debugging,
- four user-defined push-button switches,
- eight user-defined LEDs, and
- a dual 7-segment LED display.

Test systems were compiled using Altera's Quartus II v3.0SP2 design software and the SOPC Builder system development tool, v3.02 Build 245. Nios embedded processor design toolkit v3.1 was used to generate the softcore(s) in each of the systems.

6.2 Shared-Memory Test

To verify that the system was in fact enforcing cache coherency, a simple program utilizing shared memory was written using the GCC development toolset provided by Altera. One processor was arbitrarily designated as a BSP for the purposes of the test. The program begins by performing the default start-up initialization (the `_start` routine), which includes enabling interrupts and initializing the cache. All processors but the BSP are forced to synchronize on a single locking variable (initialized to 0) in shared memory. Any global initialization can then be executed, such as initializing the interrupt vector table and enabling the cache coherency module. At the end of the global initialization, the BSP sets the locking variable to 1. When set, all processors may continue execution of the program. This implements the initialization barrier mentioned in Section 4.5. Since the APs spin on the lock, the contents of the lock are stored in each data cache. If cache coherency is not enforced, then the processors never proceed past the barrier. Conversely, if the system is fully functional, then the processors are released from the lock when the BSP writes a 1 and the CCM invalidates the shared-memory address. This test is referred to as the “shared-memory” test. A code listing for this test can be found in Appendix D. Note that this test does not expose the critical flaw, as there are no cases of multiple writes in the pipeline.

A pass for the shared-memory test requires that when the CCM is enabled, all processors should continue past the synchronization point, as indicated by console messages containing the ID number of each processor. When the CCM is disabled, only the BSP prints its message, since it has the correct lock contents in its cache, while the APs are stuck in the lock. A fail indicates that the write by the BSP is not propagated to the APs. It is expected that all uniprocessor systems should pass this test, as cache coherency is not an issue in such systems. Any multiprocessor systems that are not equipped with either CCMs are expected to fail this test, proving that cache coherency is a problem. Both the prototype CCM systems (denoted by CCM-1) and the second-generation CCM systems (denoted by CCM-2) should pass, showing that the CCM modules behave as expected.

6.3 Multi-Write Test

A second test, hereby referred to as the “multi-write” test, exposes the critical flaw from the original CCM design, and verifies that the flaw has been resolved. The test is similar to the shared-memory test, but blocks of assembly code have been added to ensure that multiple writes are present in the processor pipeline. Since other processors load these shared memory values into cache prior to the writes, the system is considered fully functional only if all the new values are seen by all processors. The code listing for this test can be found in Appendix E.

For the multi-write test, a pass requires that when the CCM is enabled, all processors read the appropriate values for the memory addresses that are written to by consecutive store instructions. A read of an old value for any memory location by any processor is considered a failure. It is expected that all uniprocessor systems and CCM-2 equipped systems pass this test, showing that the critical multi-write flaw has been addressed by the CCM-2 design. Multiprocessor CCM-1 systems are expected to fail this test due to the flaw.

6.4 Results

The shared-memory test program was executed twice on each system, once with the CCM enabled, and once with it disabled (to ensure that the expected difference in results were due to the operation of the CCM). The systems tested were 1-, 2-, 4-, and 8-processor SMPOPC systems in three classes: without a CCM, with the prototype CCM (CCM-1), and with the second-generation CCM (CCM-2).

Table 6.1 shows the results of the testing. As can be seen the results match those expected, indicating that the developed CCM-2 module is able to fully enforce cache coherency in a Nios SMPOPC system.

Tables 6.2, 6.3 and 6.4 outline the usage statistics for the EP1S40 Stratix FPGA when config-

Number of	**Shared-Memory Test**			**Multi-Write Test**		
Nios Cores	**No CCM**	**CCM-1**	**CCM-2**	**No CCM**	**CCM-1**	**CCM-2**
1	Pass	Pass	Pass	Pass	Pass	Pass
2	Fail	Pass	Pass	Fail	Fail	Pass
4	Fail	Pass	Pass	Fail	Fail	Pass
8	Fail	Pass	Pass	Fail	Fail	Pass

Table 6.1: Test Program Results

ured with the baseline Nios system, in 1-, 2-, 4-, and 8- processor configurations without a CCM, with a CCM-1, and with a CCM-2, respectively. Each Nios was instantiated in its default configuration, with the exception of the addition of 1 kB instruction and data caches. The baseline system consisted of a standard set of slave peripherals in the system module, including an 8-bit universal asynchronous receiver-transmitter (UART), an Avalon tri-state bridge to communicate with SRAM and flash, the CCM (if appropriate) and one 2×8 PID ROM for each processor.

Number of	**Logic Elements**		**On-chip Memory**		**Frequency**
Nios Cores	**Usage**	**%**	**Usage**	**%**	**(MHz)**
1	2,791	6%	46,224	1%	102.24
2	5,563	13%	92,448	2%	86.85
4	11,278	27%	184,896	5%	68.42
8	23,541	57%	369,792	10%	61.49

Table 6.2: No CCM SMPOPC System Device Usage

The EP1S40 FPGA can be populated with much more than eight Nios processors, or a large amount of additional logic (custom instructions or other hardware). Only a maximum of 10%

Number of	**Logic Elements**		**On-chip Memory**		**Frequency**
Nios Cores	**Usage**	**%**	**Usage**	**%**	**(MHz)**
1	2,841	6%	46,224	1%	99.45
2	5,710	13%	92,448	2%	85.35
4	11,613	28%	184,896	5%	69.89
8	24,260	58%	369,792	10%	60.85

Table 6.3: Prototype SMPOPC System Device Usage

Number of	**Logic Elements**		**On-chip Memory**		**Frequency**
Nios Cores	**Usage**	**%**	**Usage**	**%**	**(MHz)**
1	2,861	6%	46,480	1%	100.16
2	5,790	14%	92,960	2%	83.44
4	11,708	28%	185,920	5%	69.66
8	24,302	58%	371,840	10%	61.74

Table 6.4: Second-Generation CCM SMPOPC System Device Usage

of the on-chip memory bits was used, thus there are plenty of resources for increasing the cache sizes up to 4 kB or 8 kB. The resources of the largest FPGA device in the four Altera FPGA families capable of implementing Nios caches are listed in Table 6.5. It is clear that even the low-cost Cyclone family is capable of being configured with a four-way Nios system and that the largest FPGA Altera offers may be able to support up to thirty-two Nios processors in an SMP configuration.

Family	**Device**	**Logic Elements**	**Memory (bits)**
Stratix II	EP2S180	179,400	9,383,040
Stratix I	EP1S80	79,040	7,427,520
Cyclone II	EP2C70	68,416	1,152,000
Cyclone	EP1C20	20,060	294,912

Table 6.5: Maximum Capacities of Altera FPGA Families

6.5 Analysis

It is important to determine what hardware resources are required to enforce cache coherency through the addition of a CCM. This is important because a hardware intensive solution makes the SMPOPC architecture less worthwhile, as the CCM consumes LEs and memory resources at the expense of other functional logic. This additional logic may also contribute to the critical path, thus slowing down the overall system. The following comparisons between the device usage of the prototype and second-generation cache coherency modules against the baseline system shows that implementing a CCM has only a small incremental cost in hardware resources.

CCM-1 required an increasing number of LEs per system processor as the number of processors increased, largely due to the increasing complexity of the single Avalon slave port arbitrator.

The CCM cost starts at fifty LEs for one processor, increasing to approximately ninety LEs per processor for an eight-way system, for a total of 719 additional LEs (1.7% of the 1S40's total LEs). No embedded memory bits were used, and the frequency varied against the baseline system from -2.72% to +2.14%, well within the noise margin of the synthesis and place and route CAD tools.

CCM-2 also required an increasing number of LEs per system processor as the number of processors increased. The CCM cost starts at seventy LEs for one processor, and reaches a peak at 113.5 LEs per processors in a two-way system, and finally shrinks to 95.125 LEs per processor in the eight-way system. The second-generation CCM also requires $4 \times 32 = 128$ bits per memory interface per processor for a full depth FIFO (this may be reduced by the designer if it is determined it is safe to do so). Again, the frequency varies against the baseline system from -3.93% to +1.81%.

The number of embedded memory bits used did not exceed 11% of the total number of bits available in the 1S40. It is clear that on-chip memory can be used for other things, such as storing important and frequently used portions of program memory (such as the CCM ISR or the interrupt vector table), or implementing a level-2 cache to reduce memory access latency.

It should be noted that, while the frequency of the system almost decreases exponentially as the number of processors in the system is doubled, the CCM does not affect the frequency in any significant way. The frequency of the cache coherency enabled systems varied by a magnitude of less than 4% (usually 2% or less) when compared to the baseline system. Figure 6.2 illustrates the critical paths for baseline (no CCM) 1-, 2-, 4-, and 8-processor systems, as identified by the default timing analysis performed by the Quartus II design software. The logical critical path is shown in light gray, while the segments of the critical path are shown in white.

The critical path in the uniprocessor system is composed of eight segments, originating from the processor to an address line of the external tri-state bus. The critical paths of the multipro-

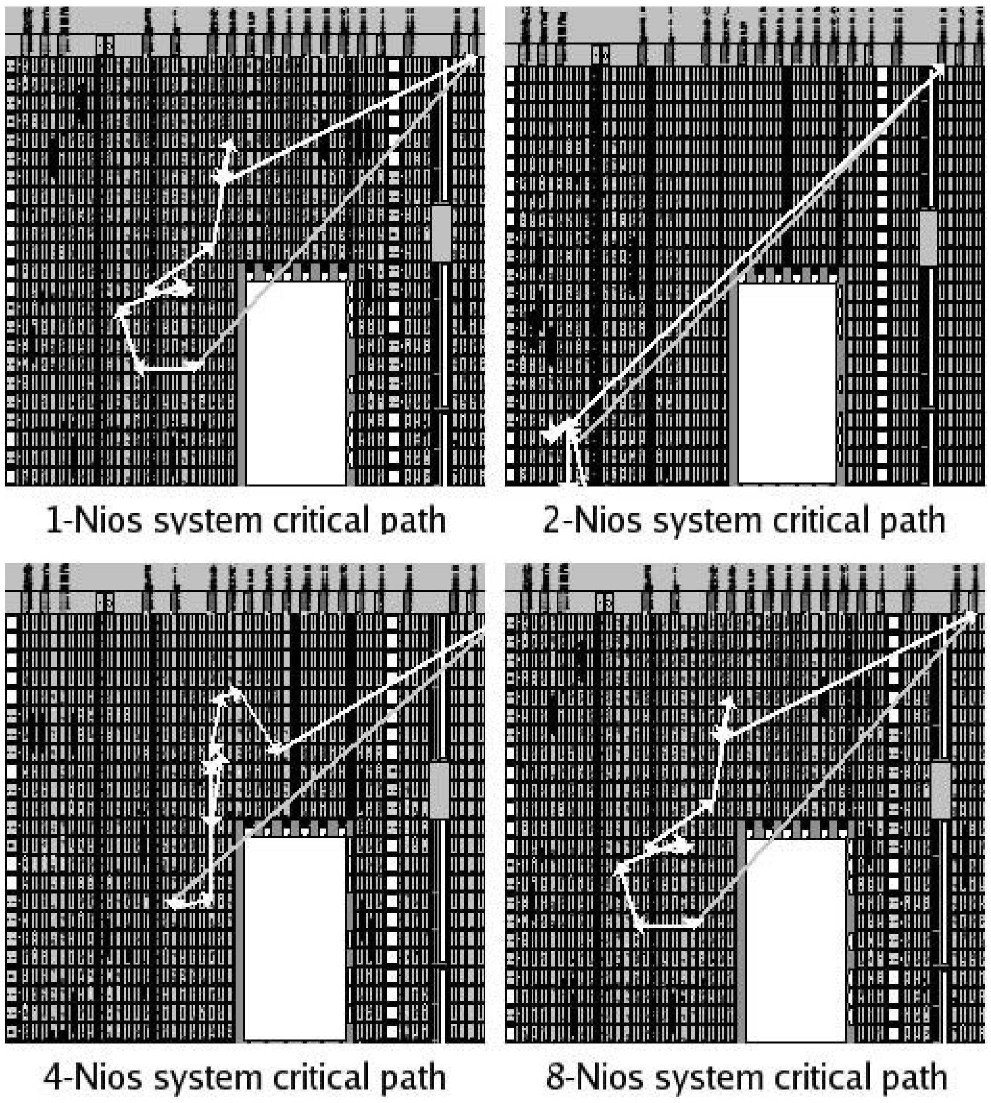
1-Nios system critical path
2-Nios system critical path
4-Nios system critical path
8-Nios system critical path

Figure 6.2: Critical Paths for Baseline Systems

cessor systems are very similar to each other, and are composed of nine, twelve, and thirteen segments, respectively, originating from the tri-state bus arbitrator to an address line of the external tri-state bus. In the multiprocessor systems, the critical path is composed of more segments due to the addition of more processors, which add to the arbitration logic and multiplexers of the Avalon bus. Timing analysis of systems with a CCM shows that the CCM is not a part of the critical path, and thus any decrease in system clock frequency can not be attributed to the CCM.

The performance numbers were obtained using the default configurations and optimizations for the Nios and Quartus II design software. The degradation of clock frequency as the number of processors increases is a consequence of the place and route tools. If performance is an issue, it may be improved by utilizing additional Quartus options and optimizations that are disabled by default. If even greater performance is desired, then the designer may spend some effort in hand-placing portions of the system within the FPGA.

Finally, the appropriate memory consistency model must be determined. Consider that the Nios, a simple scalar pipelined processor, statically schedules instructions. As a result, though up to five instructions may be simultaneously in-flight, only one instruction is executed at a time, and instructions are not re-ordered by hardware. Moreover, the Nios completes a memory access before the next instruction executes. Finally, a Nios optionally supports a data cache with a write-through policy. Therefore a single Nios system fully supports a strict memory consistency model.

An instantiated SMP Nios system, conversely, does not support the strict model. Consider instead how the characteristics of an SMP Nios system with enforced cache coherency affect the program order and write atomicity requirements explained in Section 2.2.2. The presence of cache allows a processor to write a value, followed by a read to that new value, prior to the write completing (*i.e.*, prior to all other processors invalidating their cache). This relaxation is called read-own-write-early.

With respect to program order, an SMP Nios system can be said to follow program order, given that instructions are statically scheduled. Though the Nios does not execute the instruction following a write until the memory transaction is complete, this does not include any cache invalidations that may be required. To this end, the ISR provides a real acknowledgement for writes, but in the meantime, processors continue to execute in-flight instructions at the time the CCM raises an interrupt. This can lead to the following situation where a processor B has already cached memory location Y, but can not read Y before memory location X has a certain value. If processor A writes to memory location Y, followed by a write to X, then processor B may see the write to X and proceed to read Y. Since location Y is cached and the read instruction is in-flight, processor B will read Y before it can be invalidated, causing the old, incorrect value to be returned.

Avoiding the problem involves forcing processor A to wait until processor B has invalidated Y prior to writing X. Unfortunately, this can not be accomplished with the Nios architecture and a CCM module. Instead, the problem can be solved one of two ways. The first is by placing synchronizing writes three or more instructions after the data write (causing the synchronizing write to be located either as the last instruction prior to ISR execution, or after ISR execution, guaranteeing that the synchronizing read will not succeed prior to the old cache data being invalidated). The second is by forcing such data reads to bypass the cache entirely by using the GCC `volatile` keyword or `PFXIO` assembly instruction.

An SMP Nios system also makes writes appear atomic. Write atomicity has two requirements: that all processors see writes to the same location in the same order (write serialization), and that a written value cannot be read by another processor unless all processors read the same value. The first condition is enforced by the nature of cache coherency, since all processors are forced to execute the same ISR at the same time, and thus all processors have a uniform view of memory writes. The second condition is also satisfied by the ISR, since the ISR is essentially uninterruptible, thus no read can occur prior to all processors invalidating their cache.

Since writes appear atomic, if write serialization can be enforced, then the SMP Nios system follows the sequential consistency model. Furthermore, such a system supports the read own write early relaxation. Alternatively, since the solution to the program order problem is not necessarily a desirable solution, a relaxed WAW ordering can be used instead of a sequential model. The only difference between using the sequential model and the relaxed WAW model is the semantics of memory behaviour. A programmer can choose to use either of the two models. The choice and understanding of the chosen model affects the semantics, and therefore correctness, of a parallel shared-memory program.

6.6 Summary

This chapter has described the platform on which cache coherent Nios SMPOPC systems were developed. The results of validation testing were presented, and the impact of the two CCM designs were assessed. The resulting cache coherent system was analyzed to develop a memory consistency model. The next chapter presents the conclusions drawn from this thesis and discusses possible future work.

Chapter 7

Conclusions and Future Work

This thesis presents the development and validation of a generic, *N*-way SMP Nios system with enforced cache coherency. Creating a working SMP system using Altera's SOPC Builder system development tool is easy, and requires only the addition of processor IDs, and cache coherency enforcement if caches are used. The solution to the cache coherency problem requires few additional resources, has minimal affect on overall system performance, is unique due to the limited Nios uniprocessor model and the point-to-point nature of the Avalon bus, and is scalable and affordable. Furthermore, this solution is readily adaptable to similar systems with write-through cache, cache line invalidation or update facilities, and a well-specified bus interface.

The prototype cache coherency module suffered from a fatal flaw for cases of additional writes occurring after the original exception event due to processor pipelining. The second-generation CCM was able to eliminate this flaw by implementing a memory to capture all in-flight write addresses.

A drawback to the prototype CCM as presented is a processor blocking time that scales with the number of processors in the system, as they all contend to access the CCM. The solution to this drawback, as implemented in the second-generation CCM, involves building a slave port for each processor to allow all processors to simultaneously access the CCM during the ISR. This is analogous to adding read ports to the CCM. This design makes the system far more scalable (does not run into the 32-bit `STATUS` register limitation, as each port tracks the status of its master processor) with little to no additional hardware, as point-to-point access to the Avalon slave ports are arbitration free.

A similar situation faces both the prototype and second-generation CCMs, as all processors rush to read the ISR when an interrupt is raised. This is somewhat mitigated if the ISR happens to be in the Nios instruction cache and self-modifying code has been disallowed (so the instruction cache remains enabled during the ISR itself). The problem could potentially be eliminated by storing the ISR code in an on-chip ROM, which would reduce access latency. A single shared ROM still has the problem of contention, however, so the ROM may be duplicated so that only a few processors share a ROM, or even so that each processor has exclusive access to its own ROM (similar to the PID ROM). Another advantage of having ISR ROMs is that the instruction cache may be disabled during ISR execution, saving the ISR instructions from replacing program instructions in the cache. This is effective because it reduces the miss rate on the instruction cache once the program resume execution, and because the instruction cache does not reduce latency when compared to accessing an on-chip ROM.

Future work focuses on increasing system efficiency in a number of ways. For example, a level-2 cache can be built into the module, taking advantage of fast on-chip memory to further reduce access times and contention for the memory bus. Also, for single writes, the module could determine which processor wrote the address, and thus allow that processor to continue execution without interruption (or have a foreshortened ISR). Similarly, the CCM could track the current addresses that are contained in each processor's cache, avoiding unnecessary cache clearing in the case that a processor has not cached the address currently being invalidated.

Finally, the CCM could be made to include hardware synchronization primitives as functionality separate from cache coherency enforcement. A simple test-and-set or fetch-and-increment primitive could easily be implemented, as processor accesses to the CCM bypass any caches (via the `PFXIO` instruction), and the CCM represents a single point of control for the shared variable. After writing the proper libraries to take advantage of this feature, this would allow processors to perform atomic, single-instruction synchronization functions (a read to the CCM could automatically set or increment a synchronization register). Alternatively, other hardware synchronization methods [29] could be incorporated into the CCM.

Appendix A

System ISR Installer

This assembly code installs a system ISR (the routine directly pointed to in the interrupt vector table). A system ISR cannot easily call other routines, and traps are disabled. Contrasted this with user ISRs, which are normal routines with normal facilities. The only difference between a user ISR and a normal routine is that it can only be interrupted by a higher priority interrupt. The reason a system ISR is used is to avoid Altera's default user ISR funnel code, which performs additional unnecessary tasks, such as decrementing the register window and saving registers to the stack. As a result, the funnel code leads to relatively high-latency interrupts. Since the CCM ISR routine must be very fast, and does not require normal routine facilities, the user funnel can be avoided. Instead, the CCM ISR uses whatever resources are automatically provided to all interrupts by the hardware itself: free access to register %o0-%o5 and %l0-%l7. Since the CCM ISR does not require that many registers, and `SAVE` and `RESTORE` instructions are not used, the register window will not underflow or overflow.

```
        .include "excalibur.s"
        .text
        .global nr_installsystemisr
        .global nr_ccmisr

;--------------------------------------
; void nr_installsystemisr(int trapNumber, nios_isrhandlerproc3 *trapProc);
;
;  Description: Install a trap routine
;        Input: %o0 = trap number
;               %o1 = trap handler routine
;       Output: none
; Side Effects: %g0 & %g1 altered
;    CWP Depth: 0
;

.ifdef __nios32__
        .equ nmul,2
.else
        .equ nmul,1
.endif

.equ _ISRManagerDebugging_,0

nr_installsystemisr:
        MOV     %g0,%o0                ; %g0 = Trap Number
        LSLI    %g0,nmul               ; %g0 = offset into vector table

        MOVIA   %g1,nasys_vector_table  ; %g1 -> vector table
        ADD     %g1,%g0                ; %g1 -> entry for this trap in table

        ST      [%g1],%o1              ; install the handler routine last

        JMP     %o7                    ; return
        NOP                            ; delay slot
```

Appendix B

Prototype CCM VHDL

This VHDL code represents the prototype CCM module for a dual-processor Nios system with external asynchronous SRAM and flash on the tri-state bus. The only difference between the CCM module for different number of processors is the `NUM_NIOS` generic parameter in line 8, which is set to the number of Nios processors in the system.

```
 1  library altera_vhdl_support;
 2  use altera_vhdl_support.altera_vhdl_support_lib.all;
 3
 4  library ieee;
 5  use ieee.std_logic_1164.all;
 6
 7  ENTITY ccm IS
 8     GENERIC(NUM_NIOS      : integer:=   2);
 9     PORT(   -- Avalon slave port
10        signal address     : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
11        signal chipselect  : IN STD_LOGIC;
12        signal clk         : IN STD_LOGIC;
13        signal read_n      : IN STD_LOGIC;
14        signal reset_n     : IN STD_LOGIC;
15        signal write_n     : IN STD_LOGIC;
16        signal writedata   : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
17        signal irq         : OUT STD_LOGIC;
18        signal readdata    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
19
20        -- external tri-state bus signals
21        signal ext_ram_bus_address     : IN STD_LOGIC_VECTOR(22 DOWNTO 0);
```

```
      signal ext_ram_bus_writen      : IN STD_LOGIC;
      signal write_n_to_the_ext_sram : IN STD_LOGIC
   );
END ccm;


ARCHITECTURE europa OF ccm IS
   SIGNAL internal_we  : STD_LOGIC;
   SIGNAL internal_writedetect : STD_LOGIC;
   SIGNAL strobe_read  : STD_LOGIC;
   SIGNAL strobe_write : STD_LOGIC;
   -- interrupt driver, address 0x00
   SIGNAL reg_status   : STD_LOGIC_VECTOR(NUM_NIOS-1 DOWNTO 0);
   -- interrupt enable/disable, address 0x01
   SIGNAL reg_control  : STD_LOGIC;
   -- byte address store, address 0x02
   SIGNAL reg_address  : STD_LOGIC_VECTOR(31 DOWNTO 0);
   SIGNAL read_mux_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
   SIGNAL address_mux  : STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN

   -- write detect signal for crossing into a synchronous domain
   process (clk, reset_n, internal_we) begin
      if reset_n = ’0’ then
         internal_writedetect <= ’0’;
      elsif internal_we = ’0’ then
         internal_writedetect <= ’1’;
      elsif clk’event and clk = ’1’ then
         internal_writedetect <= ’0’;
      end if;
   end process;

   strobe_read <= chipselect AND NOT read_n;
   strobe_write <= chipselect AND NOT write_n;
   -- STATUS REGISTER: interrupt status bit
   process (clk, reset_n) begin
      if reset_n = ’0’ then
         reg_status <= (OTHERS => ’0’);
      elsif clk’event and clk = ’1’ then
         if std_logic’((strobe_write AND
               to_std_logic(address = "00"))) = ’1’ then
            reg_status <= reg_status AND
               NOT writedata(NUM_NIOS-1 DOWNTO 0);
         elsif std_logic’(internal_writedetect) = ’1’ AND
               reg_control = ’1’ then
            reg_status <= (OTHERS => '1');
         end if;
      end if;
   end process;

   -- CONTROL REGISTER: bit 0 is the interrupt enable bit
   process (clk, reset_n) begin
      if reset_n = '0' then
         reg_control <= '0';
      elsif clk'event and clk = '1' then
         if std_logic'(strobe_write AND
               to_std_logic(address = "01")) = '1' then
            reg_control <= writedata(0);
         end if;
      end if;
   end process;

   -- ADDRESS REGISTER
   process (reset_n, clk) begin
      if reset_n = '0' then
         reg_address <= x"00000000";
      elsif clk'event AND clk = '1' then
         if internal_writedetect = '1' AND reg_control = '1' then
            reg_address <= address_mux;
         end if;
      end if;
   end process;

   -- Combinational register reads (read_wait_states = "0")
   read_mux_out <= A_EXT(reg_status, 32) WHEN address = "00" else
               A_REP(reg_control, 32) WHEN address = "01" else
               reg_address WHEN address = "10" else
               x"FFFFFFFF";
   readdata <= read_mux_out when strobe_read = '1' else
               x"00000000";

   -- Combinational address mux
   address_mux <= A_WE_StdLogicVector(
      (std_logic'(write_n_to_the_ext_sram) = '1'),
      "00000000" & (("0" & ext_ram_bus_address)
         OR "100000000000000000000000"),
      "000000000" & ((ext_ram_bus_address
         OR "00000000000000000000000")) );

   internal_we <= ext_ram_bus_writen AND write_n_to_the_ext_sram;
```

```
    irq <= or_reduce(reg_status) AND reg_control;

END europa;
```

Appendix C

Second-Generation CCM VHDL

This VHDL code represents the second-generation CCM module for a dual-processor Nios system with external asynchronous SRAM and flash on a tri-state bus. A `ccm_slave_if` component is instantiated for each Nios processor in the system, and a `ccm_fifo` component (and associated top-level signals and registers) is instantiated for each memory device to be supported. This code has been formatted to better fit these pages, and Altera's autogenerated VHDL for the FIFO megafunction IP block has been removed (though the component interface remains).

```
library altera_vhdl_support;
use altera_vhdl_support.altera_vhdl_support_lib.all;

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ccm_slave_if is
        port (
              -- inputs:
                 signal address : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal ccm_en : IN STD_LOGIC;
                 signal chipselect : IN STD_LOGIC;
                 signal clk : IN STD_LOGIC;
                 signal clk_en : IN STD_LOGIC;
                 signal exception_status_bit : IN STD_LOGIC;
                 signal fifo_status_bit : IN STD_LOGIC;
                 signal new_address : IN STD_LOGIC;
                 signal read_n : IN STD_LOGIC;
                 signal reg_address : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                 signal reset_n : IN STD_LOGIC;
                 signal write_n : IN STD_LOGIC;
              -- outputs:
                 signal address_read : OUT STD_LOGIC;
                 signal control_wr_strobe : OUT STD_LOGIC;
                 signal readdata : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
              );
end entity ccm_slave_if;

architecture europa of ccm_slave_if is
                signal address_rd_strobe :  STD_LOGIC;
                signal reg_control :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal reg_status :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal selected_read_data :  STD_LOGIC_VECTOR (31 DOWNTO 0);
begin

  process (clk, reset_n)
  begin
    if reset_n = ’0’ then
      readdata <= "00000000000000000000000000000000";
    elsif clk’event and clk = ’1’ then
      if std_logic’(clk_en) = ’1’ then
        readdata <= selected_read_data;
      end if;
    end if;
  end process;

  address_rd_strobe <= (chipselect AND NOT read_n) AND
    to_std_logic(((address = "00")));
  control_wr_strobe <= (chipselect AND NOT write_n) AND
    to_std_logic(((address = "11")));
  reg_status <= "000000000000000000000000000000" &
    (Std_Logic_Vector’(A_ToStdLogicVector(fifo_status_bit) &
    A_ToStdLogicVector(exception_status_bit)));
  reg_control <= "0000000000000000000000000000000" &
    (A_TOSTDLOGICVECTOR(ccm_en));
  selected_read_data <=
    (((A_REP(to_std_logic(((address = "00"))), 32) AND reg_address)) OR
    ((A_REP(to_std_logic(((address = "10"))), 32) AND reg_status))) OR
    ((A_REP(to_std_logic(((address = "11"))), 32) AND reg_control));

  process (clk, reset_n)
  begin
    if reset_n = '0' then
      address_read <= '1';
    elsif clk'event and clk = '1' then
      if std_logic'(clk_en) = '1' then
        if std_logic'(new_address) = '1' then
          address_read <= '0';
        elsif std_logic'(address_rd_strobe) = '1' then
          address_read <= '1';
        end if;
      end if;
    end if;
  end process;

end europa;


library altera_vhdl_support;
use altera_vhdl_support.altera_vhdl_support_lib.all;

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ccm_regs is
        port (
              -- inputs:
                 signal all_read : IN STD_LOGIC;
                 signal clk : IN STD_LOGIC;
                 signal control_wr_strobe : IN STD_LOGIC;
                 signal fifo_full : IN STD_LOGIC;
                 signal new_address : IN STD_LOGIC;
                 signal qualified_irq : IN STD_LOGIC;
                 signal reset_n : IN STD_LOGIC;
                 signal selected_address : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                 signal selected_writedata : IN STD_LOGIC;
              -- outputs:
                 signal ccm_en : OUT STD_LOGIC;
                 signal exception_status_bit : OUT STD_LOGIC;
                 signal fifo_status_bit : OUT STD_LOGIC;
                 signal reg_address : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
              );
end entity ccm_regs;

```

```
architecture europa of ccm_regs is
                signal internal_ccm_en :  STD_LOGIC;
                signal internal_fifo_status_bit :  STD_LOGIC;
begin

  process (clk, reset_n)
  begin
    if reset_n = '0' then
      reg_address <= "00000000000000000000000000000000";
    elsif clk'event and clk = '1' then
      if std_logic'((new_address AND internal_ccm_en)) = '1' then
        reg_address <= selected_address;
      end if;
    end if;
  end process;

  process (clk, reset_n)
  begin
    if reset_n = '0' then
      internal_fifo_status_bit <= '0';
    elsif clk'event and clk = '1' then
      if std_logic'((NOT internal_fifo_status_bit AND
          internal_ccm_en)) = '1' then
        internal_fifo_status_bit <= fifo_full;
      end if;
    end if;
  end process;

  process (clk, reset_n)
  begin
    if reset_n = '0' then
      exception_status_bit <= '0';
    elsif clk'event and clk = '1' then
      if std_logic'(internal_ccm_en) = '1' then
        if std_logic'(qualified_irq) = '1' then
          exception_status_bit <= '1';
        elsif std_logic'(all_read) = '1' then
          exception_status_bit <= '0';
        end if;
      end if;
    end if;
  end process;

  process (clk, reset_n)
  begin
```

```
    if reset_n = '0' then
      internal_ccm_en <= '0';
    elsif clk'event and clk = '1' then
      if std_logic'(control_wr_strobe) = '1' then
        internal_ccm_en <= selected_writedata;
      end if;
    end if;
  end process;

  fifo_status_bit <= internal_fifo_status_bit;
  ccm_en <= internal_ccm_en;

end europa;


library altera_vhdl_support;
use altera_vhdl_support.altera_vhdl_support_lib.all;

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ccm_fifo is
        port (
              -- inputs:
                 signal addrAck : IN STD_LOGIC;
                 signal address_in : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                 signal address_valid : IN STD_LOGIC;
                 signal clk : IN STD_LOGIC;
                 signal clk_en : IN STD_LOGIC;
                 signal reset_n : IN STD_LOGIC;
              -- outputs:
                 signal addrRdy : OUT STD_LOGIC;
                 signal address_out : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
                 signal fifo_full : OUT STD_LOGIC
              );
end entity ccm_fifo;

architecture europa of ccm_fifo is
component a_fifo_module is
           port (
                 -- inputs:
                    signal clk : IN STD_LOGIC;
                    signal clk_en : IN STD_LOGIC;
                    signal fifo_read : IN STD_LOGIC;
                    signal fifo_wr_data : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                    signal fifo_write : IN STD_LOGIC;
                    signal flush_fifo : IN STD_LOGIC;
                    signal inc_pending_data : IN STD_LOGIC;
                    signal reset_n : IN STD_LOGIC;
                 -- outputs:
                    signal fifo_datavalid : OUT STD_LOGIC;
                    signal fifo_full : OUT STD_LOGIC;
                    signal fifo_rd_data : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
                 );
end component a_fifo_module;

                signal internal_addrRdy :  STD_LOGIC;
                signal internal_address_out :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal internal_fifo_full :  STD_LOGIC;
begin

  the_a_fifo_module : a_fifo_module
    port map(
      fifo_rd_data => internal_address_out,
      fifo_datavalid => internal_addrRdy,
      fifo_full => internal_fifo_full,
      fifo_wr_data => address_in,
      clk_en => clk_en,
      inc_pending_data => '0',
      fifo_write => address_valid,
      clk => clk,
      fifo_read => addrAck,
      reset_n => reset_n,
      flush_fifo => '0'
    );

  addrRdy <= internal_addrRdy;
  fifo_full <= internal_fifo_full;
  address_out <= internal_address_out;

end europa;


library altera_vhdl_support;
use altera_vhdl_support.altera_vhdl_support_lib.all;

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ccm is
        port (
              -- inputs:
                 signal clk : IN STD_LOGIC;
                 signal ext_ram_bus_address : IN STD_LOGIC_VECTOR (22 DOWNTO 0);
                 signal ext_ram_bus_writen : IN STD_LOGIC;
                 signal reset_n : IN STD_LOGIC;
                 signal s0_address : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal s0_chipselect : IN STD_LOGIC;
                 signal s0_read_n : IN STD_LOGIC;
                 signal s0_write_n : IN STD_LOGIC;
                 signal s0_writedata : IN STD_LOGIC;
                 signal s1_address : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal s1_chipselect : IN STD_LOGIC;
                 signal s1_read_n : IN STD_LOGIC;
                 signal s1_write_n : IN STD_LOGIC;
                 signal s1_writedata : IN STD_LOGIC;
                 signal write_n_to_the_ext_sram : IN STD_LOGIC;
              -- outputs:
                 signal s0_irq : OUT STD_LOGIC;
                 signal s0_readdata : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
                 signal s1_irq : OUT STD_LOGIC;
                 signal s1_readdata : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
              );
end entity ccm;

architecture europa of ccm is
component ccm_slave_if is
           port (
                 -- inputs:
                    signal address : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
                    signal ccm_en : IN STD_LOGIC;
                    signal chipselect : IN STD_LOGIC;
                    signal clk : IN STD_LOGIC;
                    signal clk_en : IN STD_LOGIC;
                    signal exception_status_bit : IN STD_LOGIC;
                    signal fifo_status_bit : IN STD_LOGIC;
                    signal new_address : IN STD_LOGIC;
                    signal read_n : IN STD_LOGIC;
                    signal reg_address : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                    signal reset_n : IN STD_LOGIC;
                    signal write_n : IN STD_LOGIC;
                 -- outputs:
                    signal address_read : OUT STD_LOGIC;
                    signal control_wr_strobe : OUT STD_LOGIC;
                    signal readdata : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
                 );
end component ccm_slave_if;

component ccm_regs is
           port (
                 -- inputs:
                    signal all_read : IN STD_LOGIC;
                    signal clk : IN STD_LOGIC;
                    signal control_wr_strobe : IN STD_LOGIC;
                    signal fifo_full : IN STD_LOGIC;
                    signal new_address : IN STD_LOGIC;
                    signal qualified_irq : IN STD_LOGIC;
                    signal reset_n : IN STD_LOGIC;
                    signal selected_address : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                    signal selected_writedata : IN STD_LOGIC;
                 -- outputs:
                    signal ccm_en : OUT STD_LOGIC;
                    signal exception_status_bit : OUT STD_LOGIC;
                    signal fifo_status_bit : OUT STD_LOGIC;
                    signal reg_address : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
                 );
end component ccm_regs;

component ccm_fifo is
           port (
                 -- inputs:
                    signal addrAck : IN STD_LOGIC;
                    signal address_in : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                    signal address_valid : IN STD_LOGIC;
                    signal clk : IN STD_LOGIC;
                    signal clk_en : IN STD_LOGIC;
                    signal reset_n : IN STD_LOGIC;
                 -- outputs:
                    signal addrRdy : OUT STD_LOGIC;
                    signal address_out : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
                    signal fifo_full : OUT STD_LOGIC
                 );
end component ccm_fifo;

                signal all_read :  STD_LOGIC;
                signal ccm_en :  STD_LOGIC;
                signal clk_en :  STD_LOGIC;
                signal control_wr_strobe :  STD_LOGIC;
                signal exception_status_bit :  STD_LOGIC;
                signal fifo_full :  STD_LOGIC;
                signal fifo_status_bit :  STD_LOGIC;
                signal internal_s0_readdata :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal internal_s1_readdata :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal m0_addrAck :  STD_LOGIC;
                signal m0_addrRdy :  STD_LOGIC;
                signal m0_address_in :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal m0_address_out :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal m0_address_valid :  STD_LOGIC;
                signal m0_fifo_full :  STD_LOGIC;
                signal m1_addrAck :  STD_LOGIC;
                signal m1_addrRdy :  STD_LOGIC;
                signal m1_address_in :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal m1_address_out :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal m1_address_valid :  STD_LOGIC;
                signal m1_fifo_full :  STD_LOGIC;
                signal new_address :  STD_LOGIC;
                signal qualified_irq :  STD_LOGIC;
                signal reg_address :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal s0_address_read :  STD_LOGIC;
                signal s0_control_wr_strobe :  STD_LOGIC;
                signal s1_address_read :  STD_LOGIC;
                signal s1_control_wr_strobe :  STD_LOGIC;
                signal selected_address :  STD_LOGIC_VECTOR (31 DOWNTO 0);
                signal selected_writedata :  STD_LOGIC;
begin

  clk_en <= '1';
  s0 : ccm_slave_if
    port map(
      address_read => s0_address_read,
      control_wr_strobe => s0_control_wr_strobe,
      readdata => internal_s0_readdata,
      address => s0_address,
      new_address => new_address,
      clk_en => clk_en,
      chipselect => s0_chipselect,
      read_n => s0_read_n,
      fifo_status_bit => fifo_status_bit,
      write_n => s0_write_n,
      ccm_en => ccm_en,
      clk => clk,
      reset_n => reset_n,
      reg_address => reg_address,
      exception_status_bit => exception_status_bit
    );

  s1 : ccm_slave_if
    port map(
      address_read => s1_address_read,
      control_wr_strobe => s1_control_wr_strobe,
      readdata => internal_s1_readdata,
      address => s1_address,
      new_address => new_address,
      clk_en => clk_en,
      chipselect => s1_chipselect,
      read_n => s1_read_n,
      fifo_status_bit => fifo_status_bit,
      write_n => s1_write_n,
      ccm_en => ccm_en,
      clk => clk,
      reset_n => reset_n,
      reg_address => reg_address,
      exception_status_bit => exception_status_bit
    );

  the_ccm_regs : ccm_regs
    port map(
      fifo_status_bit => fifo_status_bit,
      ccm_en => ccm_en,
      reg_address => reg_address,
      exception_status_bit => exception_status_bit,
      selected_writedata => selected_writedata,
      selected_address => selected_address,
      new_address => new_address,
      control_wr_strobe => control_wr_strobe,
      qualified_irq => qualified_irq,
      fifo_full => fifo_full,
      clk => clk,
      all_read => all_read,
      reset_n => reset_n
    );

  m0_address_in <= "000000000" & ((ext_ram_bus_address OR
    "00000000000000000000000"));
  process (clk, reset_n, ccm_en, ext_ram_bus_writen)
  begin
```

```
    if reset_n = '0' then
      m0_address_valid <= '0';
    elsif ext_ram_bus_writen = '0' AND ccm_en = '1' then
      m0_address_valid <= '1';
    elsif clk'event and clk = '1' then
      if std_logic'(ccm_en) = '1' then
        m0_address_valid <= '0';
      end if;
    end if;
  end process;

  m0 : ccm_fifo
    port map(
      addrRdy => m0_addrRdy,
      fifo_full => m0_fifo_full,
      address_out => m0_address_out,
      address_in => m0_address_in,
      clk_en => ccm_en,
      addrAck => m0_addrAck,
      clk => clk,
      reset_n => reset_n,
      address_valid => m0_address_valid
    );

  m1_address_in <= "00000000" & ((("0" & (ext_ram_bus_address)) OR
    "1000000000000000000000000"));
  process (clk, reset_n, ccm_en, write_n_to_the_ext_sram)
  begin
    if reset_n = '0' then
      m1_address_valid <= '0';
    elsif write_n_to_the_ext_sram = '0' AND ccm_en = '1' then
      m1_address_valid <= '1';
    elsif clk'event and clk = '1' then
      if std_logic'(ccm_en) = '1' then
        m1_address_valid <= '0';
      end if;
    end if;
  end process;

  m1 : ccm_fifo
    port map(
      addrRdy => m1_addrRdy,
      fifo_full => m1_fifo_full,
      address_out => m1_address_out,
      address_in => m1_address_in,
      clk_en => ccm_en,
      addrAck => m1_addrAck,
      clk => clk,
      reset_n => reset_n,
      address_valid => m1_address_valid
    );

  all_read <= s0_address_read AND s1_address_read;
  new_address <= exception_status_bit AND all_read;
  s0_irq <= exception_status_bit;
  s1_irq <= exception_status_bit;
  control_wr_strobe <= s0_control_wr_strobe OR s1_control_wr_strobe;
  fifo_full <= m0_fifo_full OR m1_fifo_full;
  qualified_irq <= m0_addrRdy OR m1_addrRdy;
  m0_addrAck <= m0_addrRdy AND new_address;
  m1_addrAck <= m1_addrRdy AND new_address;
  selected_address <= A_WE_StdLogicVector((std_logic'(((m0_addrRdy))) = '1'),
    m0_address_out, m1_address_out);
  selected_writedata <= A_WE_StdLogic((std_logic'(((s0_control_wr_strobe))) =
    '1'), s0_writedata, s1_writedata);
  s0_readdata <= internal_s0_readdata;
  s1_readdata <= internal_s1_readdata;

end europa;
```

Appendix D

Shared-Memory Test Program

This C code represents a shared-memory program that tests for cache coherency. A single processor is (arbitrarily) designated as the bootstrap processor (BSP). The BSP waits until all other application processors (APs) have loaded the value of the `synch` shared variable into the cache (via read), and begin to busy-wait loop on `synch`. The BSP then proceeds to perform global initialization, which entails installing the CCM ISR and enabling the CCM module. Finally, it writes to `synch` allowing APs to begin execution. If cache coherency is maintained, then the write to `synch` is propagated to all APs and they are able to exit the busy-wait loop and print their PID to the console. If CCM does not enforce coherency (or the CCM module disabled or is not present in the system), then the APs will always be stuck in the loop, and only the BSP will print its PID.

```
#include "nios.h"
#include <stdio.h>

#define BOOT_CPU 0
#ifdef na_ccm_s0
        #define na_ccm     na_ccm_s0
        #define na_ccm_irq na_ccm_s0_irq
#endif

void global_initialize(int cpuid);

const char *_cpuid = (char *)na_cpuid_cpu0;
```

```
int *synch = (int *)0x008FF014;

int main(void) {
        int context = *_cpuid;

        /* pre-load caches */
        (*synch) = 0;

        printf("%d\n", context);
        global_initialize(context);

        /* AP synchronization point */
        while((*synch) == 0) {
                nr_delay(1000); printf("-%d", context);
        }

        nr_delay(100);
        printf("+%d\n", context);

        while(1) {;}

        return 0;
}

/* global_initialize: setup that must be performed by only the BSP */
void global_initialize(int cpuid) {
        if(cpuid == BOOT_CPU) {
#ifdef na_ccm
                /* FIXME: This nr_delay is to "synchronize" the system...
                 * allow all APs in the system to get to the while loop
                 * before proceeding. The delay value scales with the
                 * number of processors in the system. */
                nr_delay(5000);

                /* install CCM ISR */
                nr_installsystemisr(na_ccm_irq, nr_ccmisr);

                /* enable CCM */
                na_ccm->np_ccmcontrol = 1;
#endif

                /* synchronize system */
                (*synch) = 1;
        }
}
```

Appendix E

Multi-Write Test Program

This C code represents a shared-memory program that tests for multi-write cache coherency. This test program is largely based on the shared-memory test program, with the addition of a second write immediately after the write to the synchronizing shared variable `synch`. Alternatively, the program can include fourteen writes, to guarantee the worst-case scenario where all in-flight instructions are writes. The waveform in Figure 4.5 is the result of this worst-case scenario.

The multiple in-flight writes expose the problem with the prototype CCM, as the CCM module will detect the first write, but only capture the most recent consecutive write prior to the execution of the first ISR instruction. As a result, the cache line for the `synch` variable is not invalidated if the CCM does not support capturing multiple writes, and APs will not continue past the busy-wait loop. A CCM that does support multiple writes (as the second-generation CCM design should) will allow APs to break the loop and print their PIDs.

```
#include "nios.h"
#include <stdio.h>

#undef WORST_CASE
#define BOOT_CPU 0
#ifdef na_ccm_s0
        #define na_ccm     na_ccm_s0
        #define na_ccm_irq na_ccm_s0_irq
#endif
```

```

void global_initialize(int cpuid);

const char *_cpuid = (char *)na_cpuid_cpu0;
int *synch = (int *)0x008FF014;


int main(void) {
        int context = *_cpuid;
        int i;

        /* pre-load caches */
        (*synch) = 0;
        *(synch + 1) = 0;

        printf("%d\n", context);
        global_initialize(context);
        global_initialize(context);

        /* AP synchronization point */
        while((*synch) == 0) {
                nr_delay(1000);
                if(*(synch + 1) == 0) printf("-%d", context);
                else printf("*%d", context);
        }

        printf("+%d\n", context);

        while(1) {;}

        return 0;
}


/* global_initialize: setup that must be performed by only the BSP */
void global_initialize(int cpuid) {
        static int i = 0;
        if(cpuid == BOOT_CPU) {
#ifdef na_ccm
                /* FIXME: This nr_delay is to "synchronize" the system...
                 * allow all APs in the system to get to the while loop
                 * before proceeding. The delay value scales with the
                 * number of processors in the system. */
                nr_delay(5000);

                /* install CCM ISR */
                nr_installsystemisr(na_ccm_irq, nr_ccmisr);

                /* enable CCM */
                na_ccm->np_ccmcontrol = i;
#endif

                /* synchronize system */
                (*synch) = i;

#ifndef WORST_CASE
                /* a second consecutive write to memory */
                asm( "stp [%0,0x1],%1 \n", : : "r" (synch), "r" (i) );
#else
                /* worst-case: maximum in-flight writes to memory */
                asm( "stp [%0,0x1],%1 \n stp [%0,0x2],%1 \n \
                      stp [%0,0x3],%1 \n stp [%0,0x4],%1 \n \
                      stp [%0,0x5],%1 \n stp [%0,0x6],%1 \n \
                      stp [%0,0x7],%1 \n stp [%0,0x8],%1 \n \
                      stp [%0,0x9],%1 \n stp [%0,0xA],%1 \n \
                      stp [%0,0xB],%1 \n stp [%0,0xC],%1 \n \
                      stp [%0,0xD],%1 \n stp [%0,0xE],%1 \n" \
                      : \
                      : "r" (synch), "r" (i) );
#endif

                i = 1;
        }

}
```

Appendix F

Related Papers

The following papers were accepted for publishing as a result of the work described in this thesis.

[1] Austin Hung, William Bishop, and Andrew Kennings. Enabling Cache Coherency for *N*-Way SMP Systems on Programmable Chips. In *Proceedings of the 4th International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, June 2004. CSREA Press. To appear.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical Report WRL-TR 95/7, Digital Western Research Laboratory, Palo Alta, California, September 1995.

[2] Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, June 1988.

[3] Altera Corporation. Out with Gate Counts, In with Logic Elements. Press Release, April 24, 2001.

[4] Altera Corporation. Custom Instructions for the Nios Embedded Processor. Application Note AN-188-1.2, Altera Corporation, San Jose, California, September 2002.

[5] Altera Corporation. Excalibur Device Overview. Data Sheet DS-EXCARM-2.0, Altera Corporation, San Jose, California, May 2002.

[6] Altera Corporation. Simultaneous Multi-Mastering with the Avalon Bus. Application Note AN-184-1.1, Altera Corporation, San Jose, California, April 2002.

[7] Altera Corporation, San Jose, California. *SOPC Builder PTF File Reference Manual*, September 2002.

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

[9] Altera Corporation. Implementing Interrupt Service Routines in Nios Systems. Application Note AN-284-1.0, Altera Corporation, San Jose, California, January 2003.

[10] Altera Corporation. Nios 3.0 CPU. Data Sheet DS-NIOSCPU-2.1, Altera Corporation, San Jose, California, March 2003.

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

[12] Altera Corporation, San Jose, California. *Nios Embedded Processor Software Development Reference Manual*, March 2003.

[13] Altera Corporation. Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems. Application Note AN-323-1.0, Altera Corporation, San Jose, California, September 2003.

[14] AMD. Am29LV065D. Data Sheet 25262, Advanced Micro Devices, Inc., Sunnyvale, California, June 2004.

[15] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–485, 1967.

[16] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[17] Mikael Collin, Raimo Haukilahti, Mladen Nikitovic, and Joakim Adomat. SoCrates - A Multiprocessor SoC in 40 days. In *Conference on Design, Automation and Test in Europe*, Munich, Germany, March 2001.

[18] Susan J. Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*. Ph.D. Thesis, University of California, Berkeley, April 1989.

[19] Bradly Fawcett. PLD Capacity and “Gate Counting”. *XCell*, 23, Q4 1996.

[20] M. J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, number 54, pages 1901–1909, December 1966.

[21] Carl Hamacher, Zvonko Vranesic, and Safwat Zaky. *Computer Organization*. McGraw-Hill, Inc., New York, New York, fifth edition, 2002.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[23] Raymond Hoare, Shenchih Tung, and Katrina Werger. An 88-Way Multiprocessor within an FPGA with Customizable Instructions. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium*, page 258b, Sante Fe, New Mexico, April 2004.

[24] IDT. 71V416S/L Data Sheet. Data Sheet DSC-3624/09, Integrated Device Technology, Inc., Santa Clara, California, January 2004.

[25] Intel Corporation, Santa Clara, California. *MultiProcessor Specification - Version 1.4*, 1997.

[26] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[27] ARM Ltd. Website. `http://www.arm.com/`.

[28] Micron Technology. MT48LC4M32B2 Data Sheet. Data Sheet 128MbSDRAMx32_G.p65, Micron Technology, Inc., Boise, Idaho, September 2003.

[29] Bilge E. Saglam and Vincent J. Mooney III. System-On-A-Chip Processor Synchronization Support in Hardware. In *Proceedings of the Design, Automation and Test in Europe Conference 2001*, pages 633–639, Munich, Germany, March 2001.

[30] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, Inc., New York, New York, beta edition, 2003.

[31] Hiroaki Takada, Shinya Honda, Reiji Nishiyama, and Hiroshi Yuyama. Hardware/Software Co-Configuration for Multiprocessor SoPC. In *Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems*, pages 7–8, Hakodate, Japan, May 2003.

[32] X. Wang and S. G. Ziavras. Parallel Direct Solution of Linear Equations on FPGA-Based Machines. In *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium*, pages 113–120, Nice, France, April 2003.

[33] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, 1995.

[34] Xilinx Incorporated. MicroBlaze RISC 32-Bit Soft Processor. Product Brief, Xilinx, Inc., San Jose, California, August 2002.

[35] Xilinx Incorporated. PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices. Application Note XAPP213, Xilinx, Inc., San Jose, California, February 2003.

[36] Xilinx Incorporated. Virtex II Pro Platform FPGAs: Complete Data Sheet. Data Sheet DS083, Xilinx, Inc., San Jose, California, December 2003.

[37] Kazuo Yoshida and Satoru Shingu. Research and Development of the Earth Simulator. In *Proceedings of the 9th ECMWF Workshop on the Use of High Performance Computing in Meteorology*, pages 1–13, 2000.