An FPGA Implementation of Multicore, Multithreaded PowerPC Processors with Memory Subsystem Using Bluespec

by

Alessandro Yamhure

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2008

Author_____
Department of Electrical Engineering and Computer Science
May 21, 2008

Certified by_____
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

An FPGA Implementation of Multicore, Multithreaded

PowerPC Processors with Memory Subsystem Using Bluespec

by

Alessandro Yamhure

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2008

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis describes the design, construction and integration of a multicore processor and a memory subsystem. The work is part of a joint project with IBM Watson research. The processors have multithreading capacities and implement the PowerPC ISA. The memory system consists of a parameterizable hierarchy of caches and random access memory units. All implementation was done in Bluespec System Verilog hardware synthesis language for placement on FPGA. Furthermore, this document analyzes the main tradeoffs and major challenges involved in building and integrating such a system.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

First, I would like to thank my thesis advisor Professor Arvind for offering guidance and encouragement throughout the duration of my research. I would like to acknowledge the hard work of my collaborators on this project Asif Khan and Murali Vijayaraghavan. Finally, I would like to thank Jessica Tseng, Kattamuri Ekanadham and Pratap Pattnaik of the IBM Thomas J. Watson Research Center for their firm belief in this project.

Contents

Figures

Chapter 1

Introduction

This thesis describes my design and implementation of a cache coherence memory subsystem and explains how I incorporated it into a multi-core environment using the Bluespec hardware synthesis language. Although within our research organization processors and cache systems have each been independently implemented in Bluespec, the integration of the two components to form a self-sufficient multi-core computing system has not been attempted.

The work presented here is part of a larger project led by Professor Arvind of CSAIL in collaboration with IBM research laboratory in Watson, NY. On the MIT side, I collaborated closely on this venture with PhD candidates Asif Khan and Murali Vijayaraghavan, who are also members of CSAIL. The aim of the project is to design, implement and test an in-order PowerPC processor with multi-threading capacities and a full-blown memory subsystem. The final product will be suitable for placement on FPGA and capable of booting the Linux Operating System.

In light of the growing popularity of the PowerOpen Environment (POE), the motivation behind the project is to supply the Open Courseware community with the source code for programming a simple PowerPC processor on FPGA. POE is

an open standard (created in 1991) for running a Unix based operating system on the PowerPC computer architecture.

Furthermore, it is of research interest to test and probe the capacities and limitations of Bluespec as a hardware synthesis language and, in a broader sense, as a digital systems design tool. This work will explore the Bluespec tools on a microarchitectural level as well as on a higher system integration level, thus offering insight on possible future improvements of the language.

1.1 Thesis Organization

Chapter 2 outlines the PowerPC design, describing its instruction set architecture and our chosen microarchitecture. Chapter 3 explains our Bluespec implementation of the processor pipeline and its interface to the memory system. Chapter 4 describes the memory subsystem, including the hierarchic design and the cache coherence protocol.

Chapter 2

PowerPC Design

The first part of this chapter outlines the instruction set that our design must support. The second half describes the pipeline microarchitecture of the processor.

2.1 PowerPC Architecture Overview [2]

PowerPC is a RISC microprocessor architecture that allows for superscalar performance. Instructions are of fixed length with consistent formats, permitting a simple instruction decoding mechanism. Load and store instructions provide all of the accesses to memory. The architecture includes a set of general purpose registers (GPRs) for fixed-point computation, including the computation of memory addresses. It also provides a separate set of floating-point registers (FPRs) for floating-point computation. In all computations the same register set is used as both the source for the operands and the destination of the results. Most instructions perform one simple operation.

The POWER architecture is unique among existing RISC architectures in that it is functionally partitioned; the functions of program flow control, fixed-point computation, and floating-point computation are separated. This partitioning is

ideal for the implementation of superscalar designs, in which multiple functional units execute independent instructions in parallel. POWER architectures also aim to be sufficiently simple to allow for very short cycle time, resulting in the fastest possible clock rate.

In addition, the POWER architecture includes restructured forms of most load and store instructions; these modernized forms perform the memory access and place the updated address in the base address register. Thus, there is no need for a separate address computation after each access. These updates were based on a couple of observations; first of all, loads and stores account for roughly 25% of the instructions executed by the majority of programs. Secondly, many applications manipulate arrays, for which the pattern of memory accesses is often regular (for example, every "nth" element).

Furthermore, PowerPC ISA includes a floating-point multiply-add instruction. This expansion of the ISA was based on the observation that many floating-point computational algorithms add a value to the product of two other values.

The PowerPC's application-level registers are organized into three classes [3]:

- *32 general-purpose registers* – These act as the source and destination of all integer operations and are the source for address operands in all load and store operations.

- *32 Floating point registers* - These are the source and destination of all floating-point operations. They contain either 32-bit or 64-bit signed and unsigned integer values or single and double precision floating-point values.

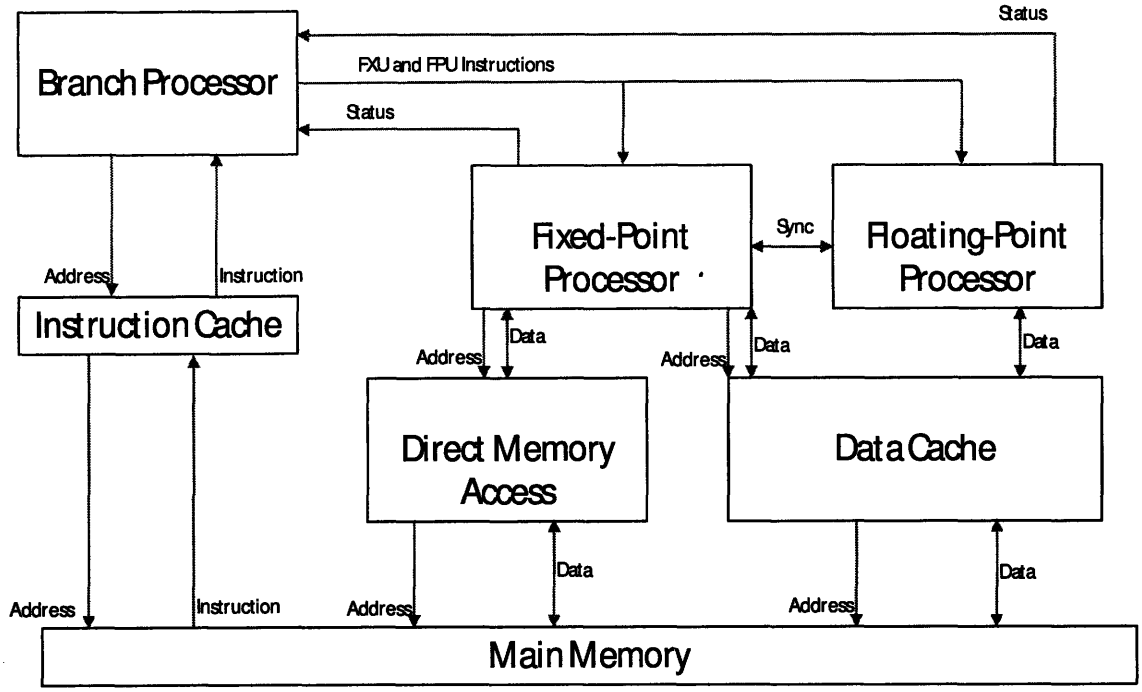

Figure 2-1: POWER and PowerPC Architecture

- *Special- purpose registers* – These give control and status of resources within the processor core. There are six SPR's in PowerPC:
    - *Instruction address register-* contains the address of the current instruction.
    - *Link register-* contains the address of the instruction to return after servicing a function call.

- *Fixed point exception register-* contains carry and overflow information from integer arithmetic operations.
- *Count register-* contains a loop counter that is decremented by certain branch operations.
- *Condition register-* contains eight 4-bit fields which reflect the result of certain operations and provides a mechanism for testing and branching.
- *Processor version register-* a 32-bit read only register that identifies the version and revision level of the processor.

2.2 PowerPC Instruction Set Architecture

The term *instruction set architecture* refers to the instruction set that is actually visible to the programmer. The ISA serves as the boundary between the software and hardware. According to Hennessy and Patterson, there are seven dimensions to an ISA [2]:

1. *Class of ISA* - Almost all present day ISA's (including the PowerPC) are general-purpose register architectures, where the operands are either registers or memory locations. There are two popular versions within this class; *register-memory* ISAs which can access memory as part of many instructions and *load-store* ISAs (such as PowerPC) which can access memory only with load or store instructions.

2. *Memory addressing* – Almost all computers (including PowerPC) use byte addressing to access memory operands. The PowerPC also requires that objects be word aligned.
3. *Addressing modes* – These are modes used to specify the address of a memory object. The PowerPC addressing modes are register, immediate (constant), displacement (where an offset is added to a register to form the memory address), two registers and no register (absolute).
4. *Types and sizes of operands* – The PowerPC ISA supports operands that are byte, halfword, word or doubleword in size.
5. *Operations [3]* – The general categories of operation within PowerPC ISA are branch, condition register, storage access, integer arithmetic, integer comparison, integer logic, integer rotate/shift, floating-point arithmetic, floating-point comparison, floating-point conversion, FPSCR management, cache control, processor management.
6. *Control flow instructions* – Virtually all ISAs (including PowerPC) support conditional branches, unconditional jumps, procedure calls and returns. PowerPC uses PC-relative addressing, where the branch address is described by an offset to the current program counter.
7. *Encoding an ISA* – The two choices for ISA encoding are *fixed length* and *variable length*. The PowerPC implements fixed length (32-bit) instructions.

The PowerPC ISA consists of three parts:

- User Instruction Set Architecture, which includes the base instruction set and related facilities available to the application programmer
- Virtual Environment Architecture, which includes the instruction set related to the storage model and Time Base as seen by the application programmer
- Operating Environment Architecture, which includes the instruction set not available to the application programmer, affecting storage control, interrupts and timing facilities.

User ISA [4]

Instructions that the processor can execute fall into three classes:

- branch instructions
- fixed-point instructions
- floating-point instructions

The operands for fixed-point instructions are byte, halfword, word or doubleword in length. Floating-point instructions operate on single-precision and double-precision floating-point operands. The instructions used by the PowerPC architecture are four bytes long and word-aligned. It allows for byte, halfword, word, and doubleword operand loads and stores between memory and a set of 32 General Purpose Registers (GPRs). It also provides for word and doubleword

operand loads and stores between memory and a set of 32 Floating-Point Registers (FPRs). Signed integers are represented in two's complement form. There are no instructions within the User ISA that directly modify storage. To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 1 shows a logical representation of instruction processing.

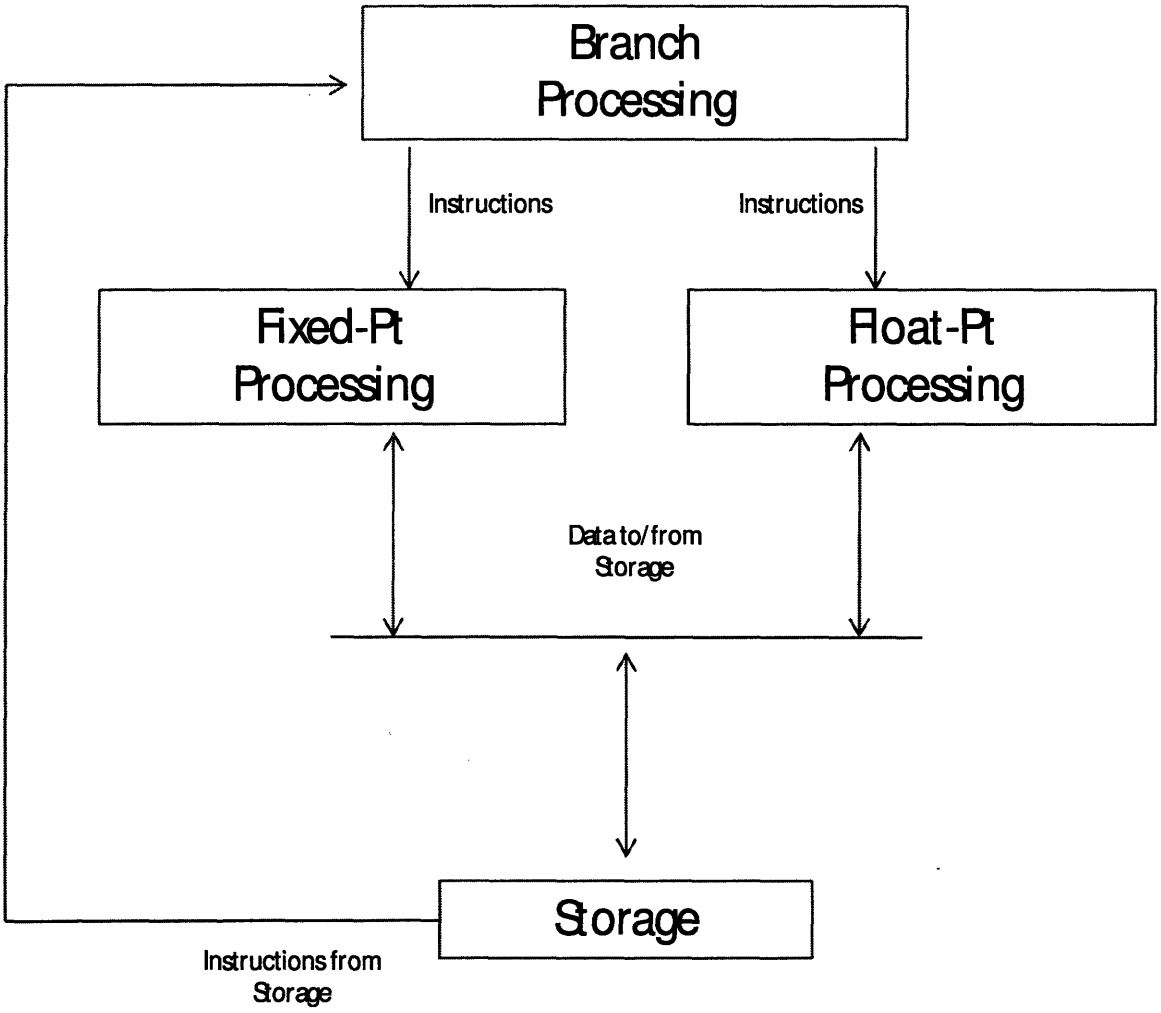

Figure 2-2: Logical Processing Model

Virtual Environment Architecture [5]

In the User ISA, discussed above, storage is defined as a linear array of bytes indexed from 0 to $2^{64} - 1$. Each byte is indexed by its "address" and contains a value. This information, along with the User ISA, is enough to allow the programming of applications that do not require any special features of the system environment. PowerPC's virtual environment expands this simple storage model to include caches, virtual storage and shared storage. When used in combination with the Operating Environment Architecture (introduced below), the Virtual Environment Architecture allows for complete and direct control of this expanded storage model.

The PowerPC system implements a virtual storage model for applications. Applications exist within a "virtual" address space inside a storage model that is created through a combination of hardware and software. This "virtual" address space is larger than both the effective and real address spaces. The 2^{64} bytes of storage that each program can access is termed the "effective" address space. The hardware translates this effective address into a "real" address before using it to access storage. Each such effective address lies in a "virtual page", which is mapped to a "real page" (4 KB virtual page) or to a contiguous sequence of real pages (large virtual page) before data or instructions in the virtual page are accessed.

Operating Environment Architecture [6]

The processor or processor unit contains the sequencing and processing controls for instruction fetch, instruction execution and interrupt action. There are three types of instructions that the processing unit can execute:

- instructions executed in the branch processor
- instructions executed in the fixed-point processor
- instructions executed in the floating-point processor

Almost all instructions that are executed by these three processing units are "nonprivileged" and belong to the User Instruction Set. Some additional "nonprivileged" instructions for cache management are part of the Virtual Environment Architecture. Instructions related to the Operating Environment Architecture are "privileged" and are capable of control of the processor's resources and control of the storage hierarchy. In this context, by "privileged instructions" I am referring to those instructions that are associated with the supervisor state while "nonprivileged instructions" are those associated with the problem state.

2.3 Chosen Microarchitecture

Many of the design choices and system specifications were outlined by the research team at the IBM laboratory. At MIT, we were often given considerable

freedom on implementation decisions as long as the overall product fit the needs specified by IBM and supported the ISA described in the previous chapter.

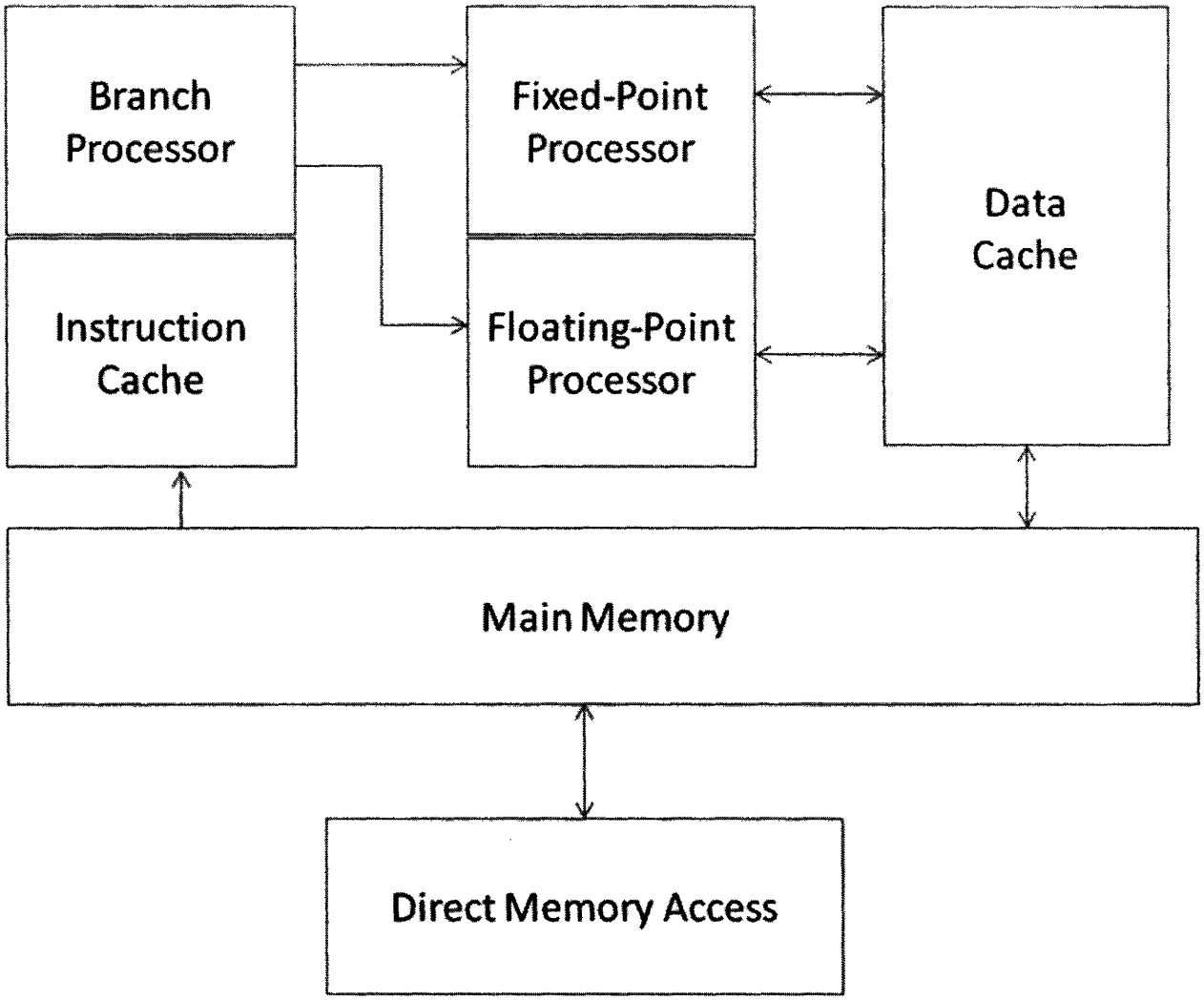

Figure 2-3: Logical View of the PowerPC Processor architecture

2.3.1 Pipeline Abstraction

The main pipeline structure was designed by Ekanadham Kattamuri and his team at IBM Watson Research Center, as a generic abstraction that contains and controls computational stages regardless of the nature and number of stages.

This is a key feature in the processor pipeline which allows stages to be added and removed easily. As depicted in figure X, it consists of an instruction unit (Iunit) and an execution unit (Xunit). Both are implementations of the aforementioned generic abstraction. They share address translation and memory interface.

The Iunit is responsible for bringing one instruction block into the machine, which involves address translation, instruction fetching, instruction decoding and scheduling on the following Xunit.

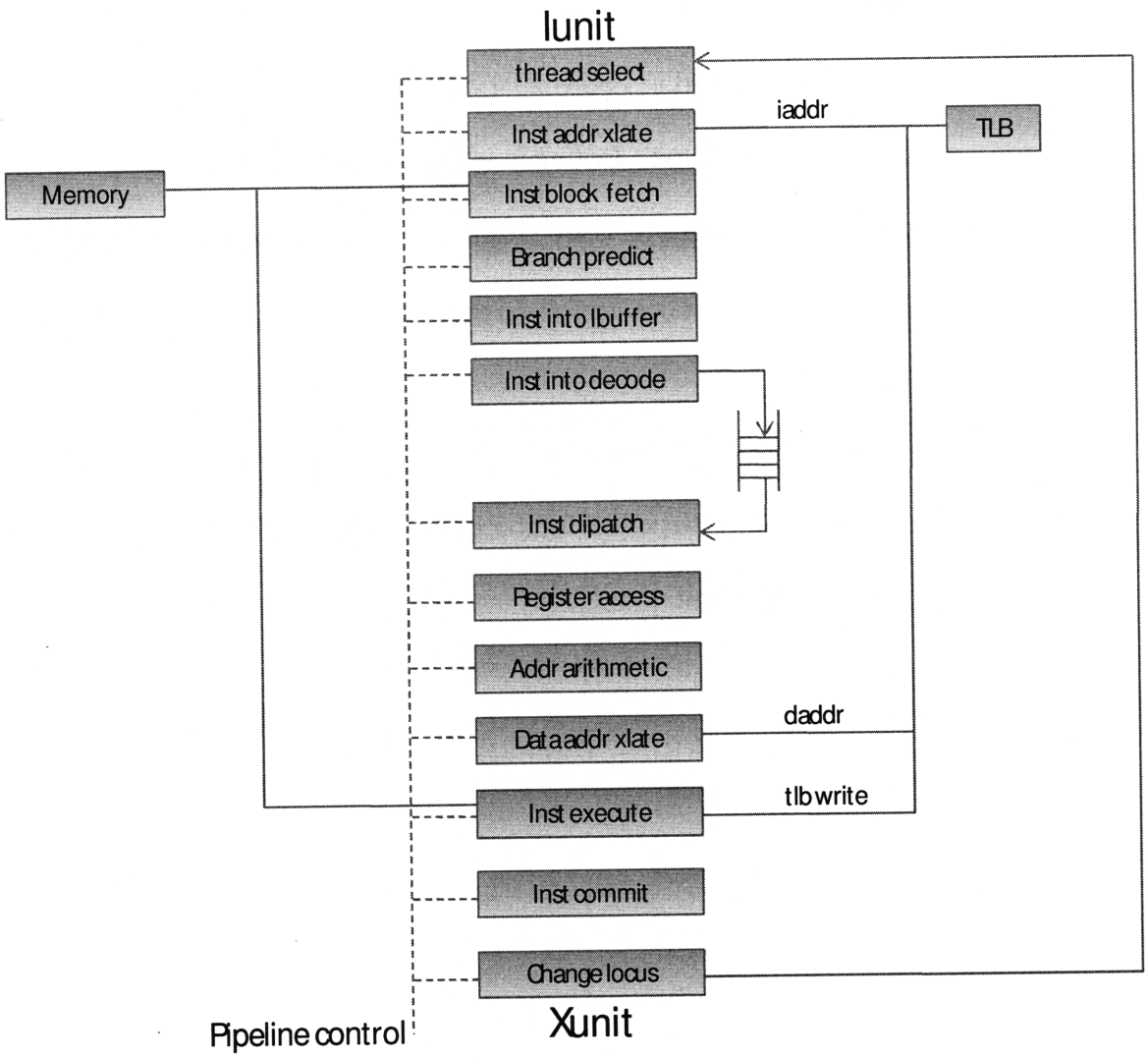

Figure 2-4 [7]: Processor Pipeline

The Xunit is responsible for executing one instruction of a thread, which involves reading the register file, effective address calculation, accessing the memory, executing arithmetic and logical operations and writing to the register file. The act of committing the results of instruction must be performed by a single pipeline stage ("inst commit") in order to guarantee atomicity.

A separate control system exists for thread scheduling. Each pipeline stage contains state relevant to each thread which contribute to the scheduling of the next thread. Additionally, separate FIFOs and register files exist for each thread.

2.3.2 Threading

A major challenge in the design of the pipeline was the support of multithreading. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. Thread-level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Each stage in our pipeline represents an operation to be performed on an incoming packet that produces an output packet. Additionally, the operation performed by a stage may alter the status of the threads. To capture this and allow for efficient multithreading, our pipeline uses the notion of a thread's *status*.

A thread's status can take one of the following values[7]:

- **Active**: indicates that a packet for that thread should be normally processed.
- **Suspend**: indicates that any packet for that thread must be abandoned. In order to re-execute that same instruction, the thread will have to reactivate later.

- **Reset**: abandons all processing for a particular thread and additionally discards any partial state associated with the thread.
- **Restart**: indicates that a thread has been restarted with a new instruction at another address.

At the beginning of a cycle, each stage is provided with an input packet and a status vector describing the status of each thread as seen by that stage. At the end of a cycle, the stage returns an output packet and possibly alters the status of the threads. A central *controller* collects the results of all the stages, pushes output packets to the next stage and propagates the statuses of threads up the pipeline.

2.3.3 Memory Interface

The interface between the processor pipeline and the memory subsystem is designed to tolerate arbitrary latency for memory accesses. The processor pipeline includes a stage known as the *load/store unit* which acts as the bridge between the two systems. The processor and memory system act differently and operate at different speeds. The load/store unit is responsible for enabling communication between these two worlds.

The load/store unit remembers the latest memory request of each thread. It submits them to the memory in a round-robin fashion. When a store request is accepted from a thread, no other memory requests are accepted from that thread until completion of that store request. Ensuing memory requests are suspended

and retried until they are accepted. Furthermore, a store request is not issued to the memory system until all preceding memory requests from that thread have been completed.

When a load request is accepted, the load instruction is suspended and retried while the load is being processed by the memory system. The load/store unit buffers data received from the memory system and delivers the data to the appropriate threads on their next retry of the load.

While the cache coherence protocol is charged with the responsibility of ensuring coherence between cores, the load/store unit is charged with the responsibility of guaranteeing sequential consistency and coherence between threads. This demands that all stores be seen in the same order by all threads. Thus the load/store unit is blocking with respect to store requests and non-blocking with respect to load requests; no memory request from the same thread can overtake a store request, and vice versa.

Chapter 3

Bluespec Implementation

Once the design of each module was agreed upon, construction and testing of the component was done in *Bluespec.* After giving some background on the Bluespec hardware synthesis language, this chapter describes the implementation of the processor pipeline modules that I was heavily involved with.

3.1 Bluespec Background

Blusepec is a strongly-typed hardware synthesis language which makes use of the "Term Rewriting System" (TRS)[1] to describe computation as a series of atomic state changes. This intermediate TRS description can then be translated through a compiler into Verilog RTL. The Verilog description can, in turn, be translated into a netlist ready for place-and-route on an FPGA.

3.1.1 Bluespec Syntax

In Bluespec, a module represents a circuit. A standard module consists of three parts:

- State (e.g. registers, memory etc.)
- Rules that modify the state

- Interfaces which provide a mechanism for outside interaction with the module.

All state is specified explicitly within a module. Behavior of a module is expressed in terms of atomic actions on the state. Every atomic action is conditional on a *predicate*. Together, actions and their associated predicates form a grouping called, in Bluespec, a "*rule*". A *rule* will *fire* if the predicate is true, resulting in actions on the state.

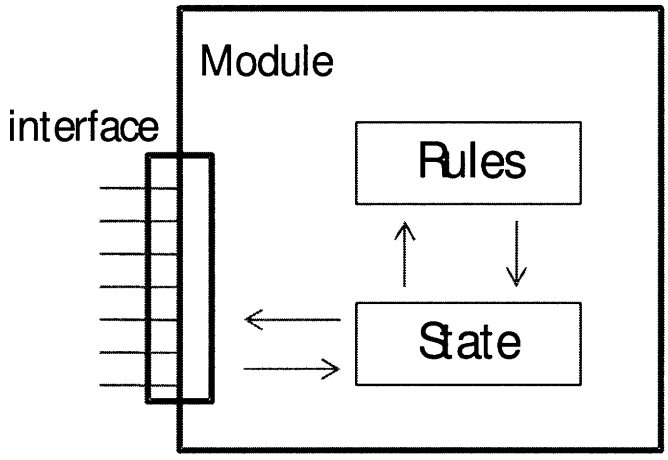

Figure 3-1: Block Diagram of a Standard Module

Finally, a module interacts with the outside world, and vice versa, through its interface. An interface consists of a set of methods. Calling a method is conditioned on a predicate, known as a *guard*. There are three types of methods in Bluespec:

- A *read* method, which simply returns a value resulting from a combinational lookup.
- An *action* method, which affects state.

- An *actionvalue* method, which is a combination of the two types of methods above – it acts on state within the module and returns a value.

3.1.2 The Bluespec Compiler

The Bluespec compiler translates Bluespec descriptions either into Verilog RTL or a cycle-accurate system-C simulation. As shown in figure 2-2, this is done by first evaluating the high-level description into a list of rules and a description of all state involved. This, in turn, is translated into a time-conscious hardware description. A key step in the compilation process involves determining what rules should *fire* on a given cycle as a function of the state involved as well as what order rules should *fire* in a given cycle. This task is collectively referred to as *"scheduling"*.

Scheduling

In order to come up with a global schedule, the Bluespec compiler makes the conservative assumption that an action will use any method that it could ever use. Given that assumption, it scans all pairs of atomic actions in search of possible conflicts. If two rule predicates are disjoint, then the atomic actions can never happen in the same cycle and there is no conflict. Otherwise, priority between the two rules is determined by analyzing each action component pair. If an action must occur before another action, the sequential restriction is reflected

in the scheduling of the rules by making the more urgent rule occur before the other.

A true conflict between atomic actions exists when two (or more) action component pairs impose *contradictory* scheduling restrictions on the rules. In this case, the atomic actions never fire in the same cycle and the compiler schedules the rules as conflicting.

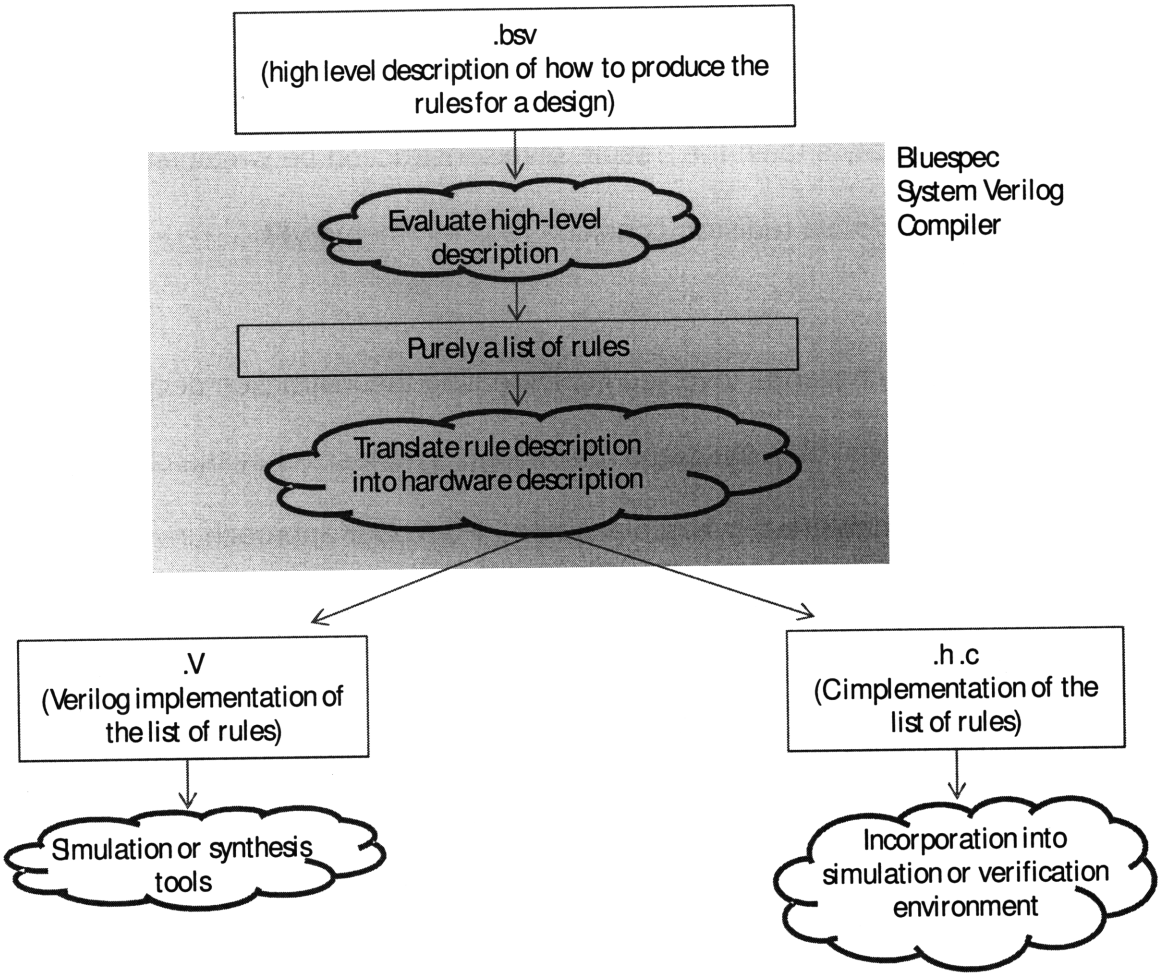

Figure 3-2: Diagram of Compiler Flow

3.2 Instruction Decode

The instruction decode stage was designed and implemented by Asif Khan, Murali Vijayaraghavan and I. In our implementation, the purpose of the "instruction decode" stage within the pipelined processor is to break down each instruction (that comes from the PowerPC ISA) into three components which answer the following three questions:

- *What* exactly needs to be done by this instruction?
- *Where* exactly does the data required for this instruction come from (general purpose registers, condition register)?
- *Where* exactly should the "result" of this instruction be written/stored (general purpose register, condition register, memory)?

The major design challenge involved implementing the instruction decode function in such a way that the combination logic synthesized by the compiler would be of reasonable size when placed on FPGA. Our approach involved "flattening" the decision tree by categorizing instructions as described below.

Within each 32-bit instruction, bits 0:5 always specify the opcode. Many instructions also have an extended opcode. The unique combination of opcode and extended opcode effectively communicates *what* needs to be done by the instruction. We chose to categorize the vast instruction set into 14 broad types of instructions. Instructions within the same category use the same ALU component and interface with the same special registers, thus making it easier to dispatch

the operation to the appropriate destination in the subsequent pipeline stages. The 14 instruction categories in the decode stage are:

i. **Branch:** possibly changes the sequence of instruction execution (often conditioned on a predicate).

ii. **System Call:** provides the means by which a program can call upon the system to perform a service.

iii. **Condition Register Move:** specified fields of the condition register are set by a move to the condition register from a general purpose register or from another field within the condition register.

iv. **Condition Register Set:** condition register fields are set as the implicit result of a fixed-point or floating-point operation.

v. **Arithmetic:** adds, multiplies, divides etc.

vi. **Compare:** equal to zero, greater than zero etc.

vii. **Trap:** provided to test for a specified set of conditions; if any of the conditions tested by a trap instruction are met, the system trap handler is invoked. Otherwise, instruction execution continues normally.

viii. **Logical:** Ands, Ors etc.

ix. **Rotate:** rotates a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

x. **Shift:** simple bit shift.

xi. **Move:** copy data from one floating-point register to another.

xii. **Load:** a byte, halfword, word or doubleword in storage addressed by an effective address (calculated in a subsequent pipeline stage) is loaded into a general purpose register.

xiii. **Store:** the contents of a general purpose register are stored into a byte, halfword, word or doubleword memory location at an effective address (calculated in a subsequent pipeline stage).

xiv. **Other:** does not fit into any of the above categories.

Thus, the "instruction decode" stage classifies every instruction into one of the above categories. The output of the pipelined stage contains information about which category the instruction belongs to. What distinguish several instructions within the same category (among other things) are the source and destination of data involved. Knowing that we would be interfacing with a general purpose register file equipped with two read-ports and one write-port, we implemented a system that deciphers which of these ports will be used for the present instruction along with which exact register each port will need to communicate with.

Thus we end up with three fields in our decoded output:

- **Rd:** which could be a read or a write register
- **Rs1:** which could be a read or a write register
- **Rs2:** which is always a write register

Depending on the instruction, any number of these registers could be used (including none or all) in an operation. Therefore we labeled each field with read and/or write valid Boolean bits.

In addition to the 32 general purpose registers, the data sources and/or destinations for the operation could be "special" registers. Therefore, the result of the decode stage contains Boolean fields that indicate whether any of the five special register (see 2.2 for background on special registers) is read and/or written or neither.

Lastly, in the case that an instruction cannot be decoded by our module, an error Boolean is set. Also, since the least significant bit of the 32-bit instruction is often crucial in distinguishing different operations within the same category, we extracted this bit into an output field. The final result of the instruction decode process takes the form:

```
typedef struct {
    InstType instType;

    RegName rd;
    Bool rdReadValid;
    Bool rdWriteValid;
    RegName rs1;
    Bool rs1ReadValid;
    Bool rs1WriteValid;
    RegName rs2;
    Bool rs2Valid;

    Bool crRead;
    Bool lrRead;
    Bool ctrRead;
    Bool xerRead;
    Bool fpscrRead;
    Bool crWrite;
    Bool lrWrite;
    Bool ctrWrite;
    Bool xerWrite;
    Bool fpscrWrite;
```

```
    Bool lsb;

    Bool instError;
} DecodedInst deriving (Bits, Eq);
```

3.3 Arithmetic Logic Unit

For most components of the ALU, the primitive functional units of the verilog library were used. This section describes the optimized components that we designed for use as functional units in the *execute* stage of the pipelined processor.

Divider

We chose to design and incorporate our own divider because the primitive divider operates on 32-bit operands. However, we implemented an expanded version of the division algorithm used by the Verilog library, known as the radix-2 restoring division algorithm described below [2].

To compute *a/b,* put *a* in register A, *b* in register B and 0 in register P. Then perform 64 divide steps, where each divide step consists of four parts:

1. Shift the register (P, A) one bit left.
2. Subtract the content of register B from register P, placing the result in P.
3. If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

4. If the result of step 2 is negative, restore the old value of P by adding the contents of B to P.

At the end of the 64 step division process, register A will contain the quotient and register P will contain the remainder. A numerical example can be found in the relevant bibliography.

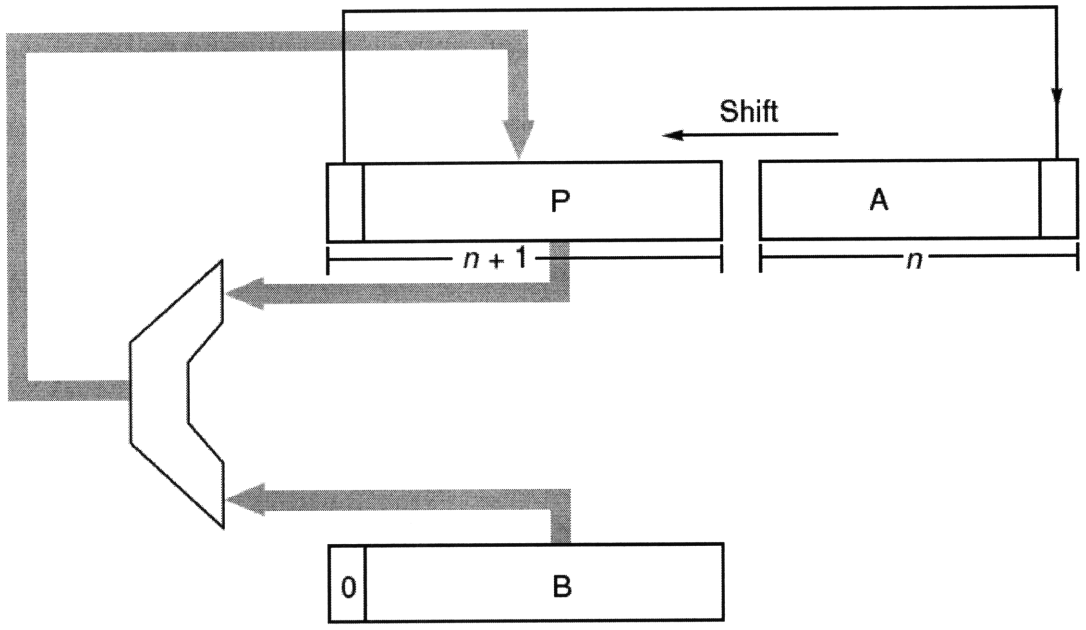

Figure 3-3: Block diagram of divider where *n* is 64 and *B* is 64 bits wide [2].

Chapter 4

Memory Subsystem

This chapter describes the design and implementation of the memory subsystem. It starts by giving background information about the general cache coherence problem that exists in any multi-core environment. It then introduces basic schemes for enforcing cache coherence including our choice of protocol. Following that is an outline of the microarchitecture of the memory. Lastly, I explain the testing methods used to verify the correctness of the scheme.

4.1 Background

In a multi-core environment like ours, each processor possesses its own private cache (see figure 5-1). Often, these caches contain shared data. Each processor views the memory through its individual cache, and thus could end up seeing different values for the same entry. Each cache has exactly one parent and can have zero or more children. Only parents and children can communicate directly.

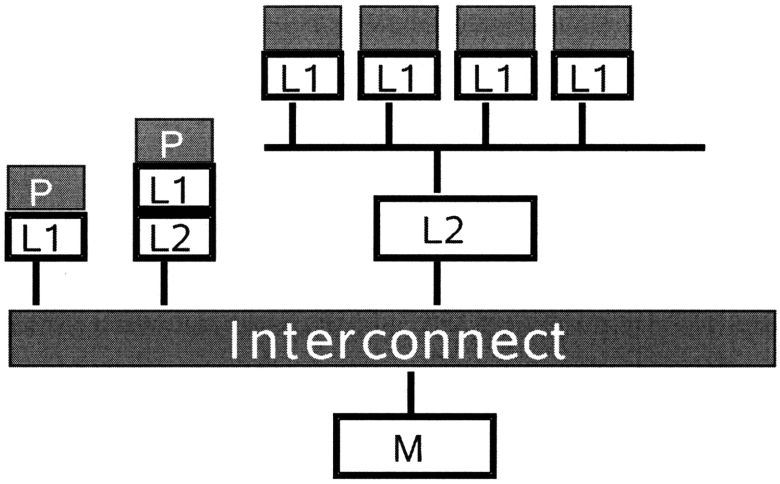

Figure 4-1: A system with multiple caches.

The following simple example (figure 5-2) emphasizes the cache coherence problem.

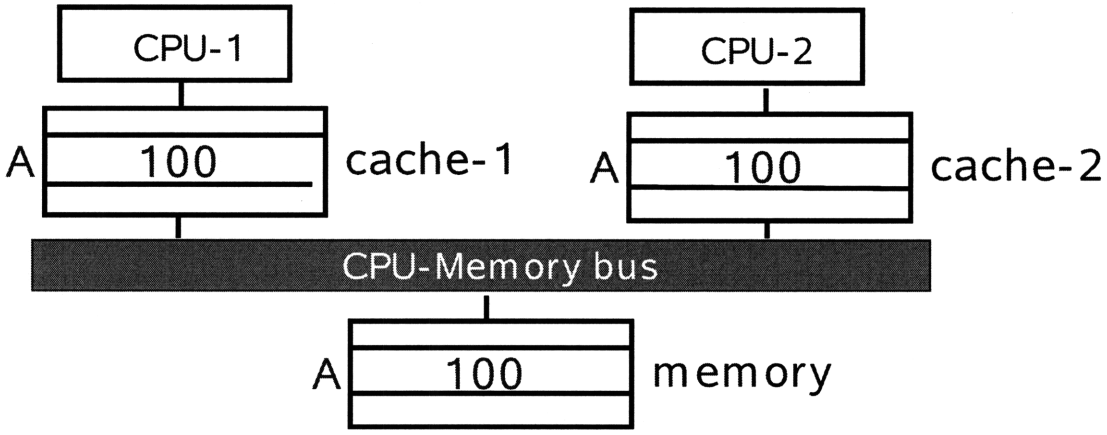

Figure 4-2: Cache Coherence Problem

If one of the two CPUs updates the value of A to 500, the other cache (and possibly the memory depending on the write-back protocol) contains a stale value. This shows how two different processors can have two different values for

the same location. This difficulty is generally referred to as the cache coherence problem. According to Hennessy and Patterson[1], a memory system is coherent if:

> A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
>
> A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
>
> Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The techniques used to maintain coherence in multi-processor environments are called cache coherence protocols. Although many such techniques exist, all fundamentally ensure coherence by tracking the state of any sharing of a data block. There are two main categories of protocols in use. They primarily differ in *where* the sharing status of a block of memory is kept.

Snoopy

Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block. The caches are all connected to a broadcast medium. Each cache contains a controller that monitors or *snoops* on the medium to determine whether or not it has a copy of the block that it is being requested on the broadcast medium.

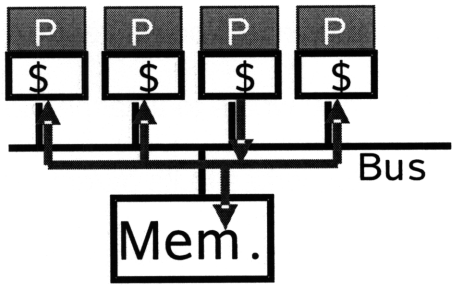

Figure 4-3: Snoopy Protocol

Directory based

The sharing status of a block is kept in a central location, called the *directory*. Messages are sent only to caches that have the block involved.

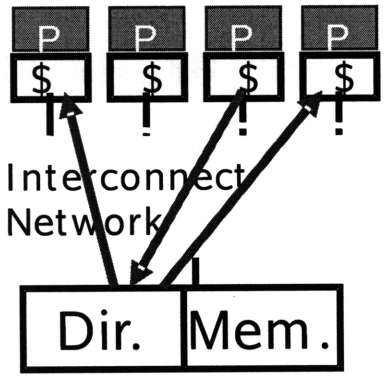

Figure 4-4: Directory Protocol

Although the directory based protocol has more implementation overhead, it is better at scaling to large numbers of processors and does not require a broadcast medium (such as a bus). Therefore, we decided to implement a directory based protocol.

4.2 Microarchitecture & Implementation

Our memory subsystem implements a *directory-based* cache coherence protocol, meaning that the status of a block is kept exclusively in the directory which is found in the parent. Any writes to a cache block result in a simultaneous write to the parent (who, in turn, writes to its parent, and so on), which implies a *write-through* system. A cache block is always exclusive, and thus the processor of that cache has full privileges on that block. Our cache model is *no-allocate,* implying that a store to an address does not result in that address being imported into the cache. Only a load can bring a line into the cache. Finally, our system implements a *random replacement policy;* when a cache miss occurs, the cache controller randomly selects a block to be replaced with the desired new data. The caches in this system are *blocking* in the sense that they only work on one request at any time and only move onto the next request when the current request is satisfied. This guarantees sequential consistency in the memory model.

Every cache is "blind to the coherence problem" on its level and is indifferent to the existence of any other caches (siblings) on the same level. In our system,

every parent only concerns itself with the cache coherence problem of its children, and not with any potential cache coherence problems on its own level with siblings. Thus a cache with no children will not implement any cache coherence mechanism. However, every child is completely obedient to its parent (who, in turn, is obedient to its parent and so on) resulting in a memory system that is coherent.

A simple cache coherence protocol was chosen for implementation in this project because correctness and parametrizability were the two major priorities and expectations of the system. Furthermore, a basic protocol allowed us to side-step tedious formal verification procedures in the interest of implementing and integrating the system in a timely manner.

All the cache and memory dimensions are parameterized, including:

Number of cache lines

Size of Cache lines

Size of Cache entries

Cache associativity

Number of caches

All these are static parameters.

4.2.1 Directory Structure

Every line in a parent is always is one of the three states:

1. *Uncached*: This data block is currently not in any child cache. The block only contains the data.

2. *Cached*: This data block is currently in ONE child cache. The block contains the data and the identity of the child cache holding a copy (hereafter id_c).

3. *Pending*: This data block is currently in a transition state; it is being invalidated at a child cache. The block only contains the data.

The following diagram illustrates the state transitions of the status of a block within a directory. In the diagram, req_c refers to the identity of the child cache that is requesting the block from the parent:

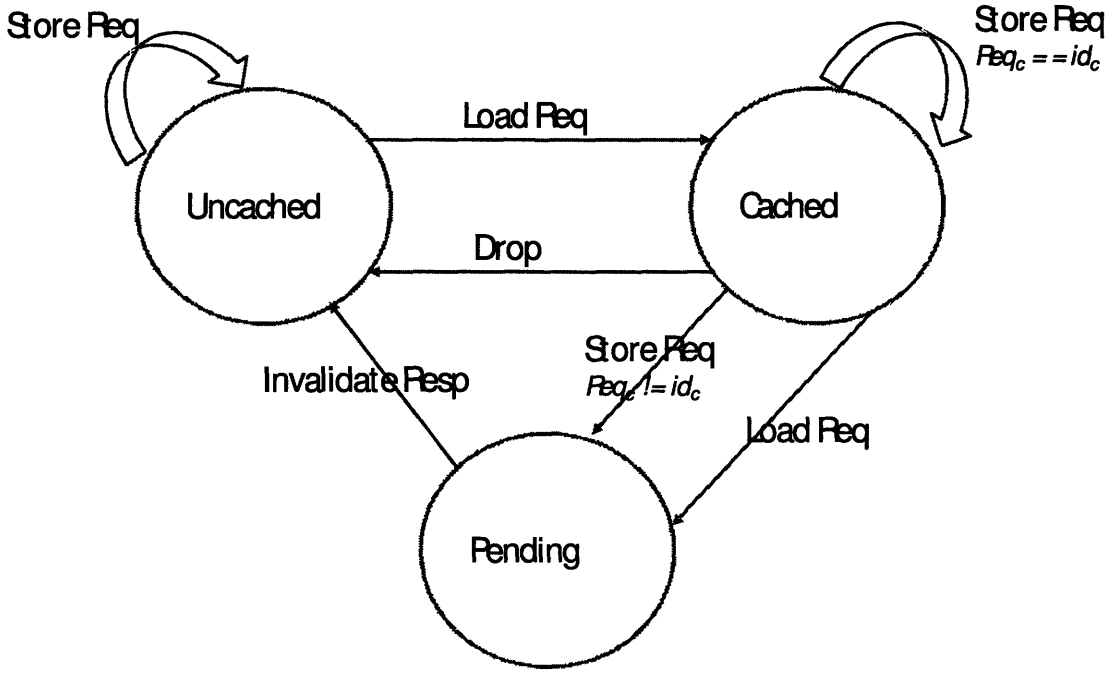

Figure 4-5: Directory State Transitions

In addition, every entry in a child cache can be in one of two states:

Valid: This data block is up to date.

Pending: This data block is currently transitioning from one state to another and is waiting for a response from the parent.

The following diagram illustrates the simple state transitions for a data block within a child cache:

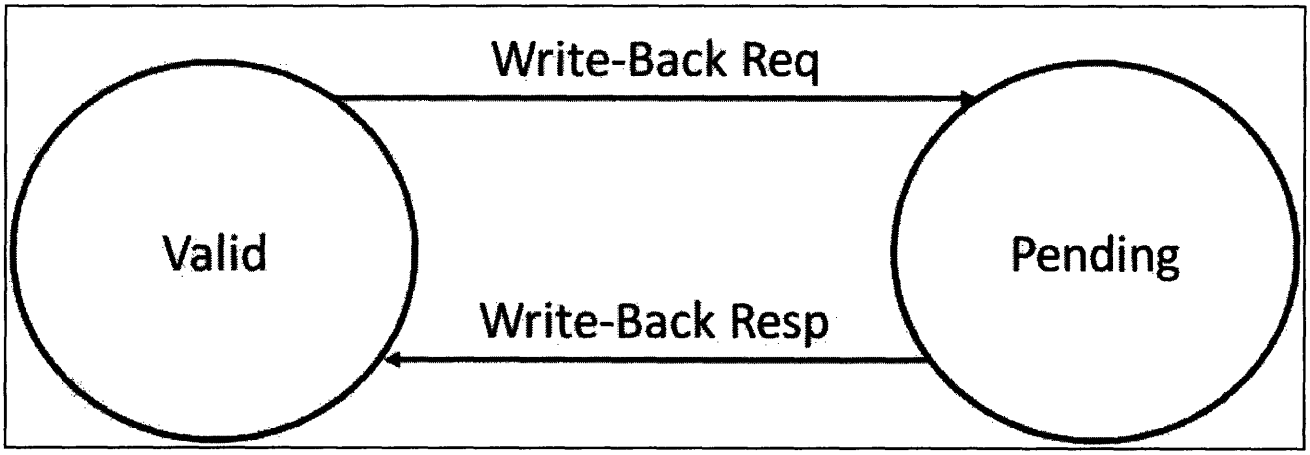

Figure 4-6: Child Cache Entry State Transitions

It is possible for a cache to be both a child and a parent. In this case, each cache entry will have both state as a parent and as a child, and those states will change as shown in both finite state machines.

4.2.2 Memory Hierarchy

The following diagram illustrates the microarchitecture of the memory subsystem. The number of cores (processor and L1 cache group) is parameterized, and can assume any value as long as it is defined at static elaboration. All arrows symbolize FIFO queues.

There are two communication paths between each cache and the memory (or the parent). One is for requests (and associated responses) that originate in the

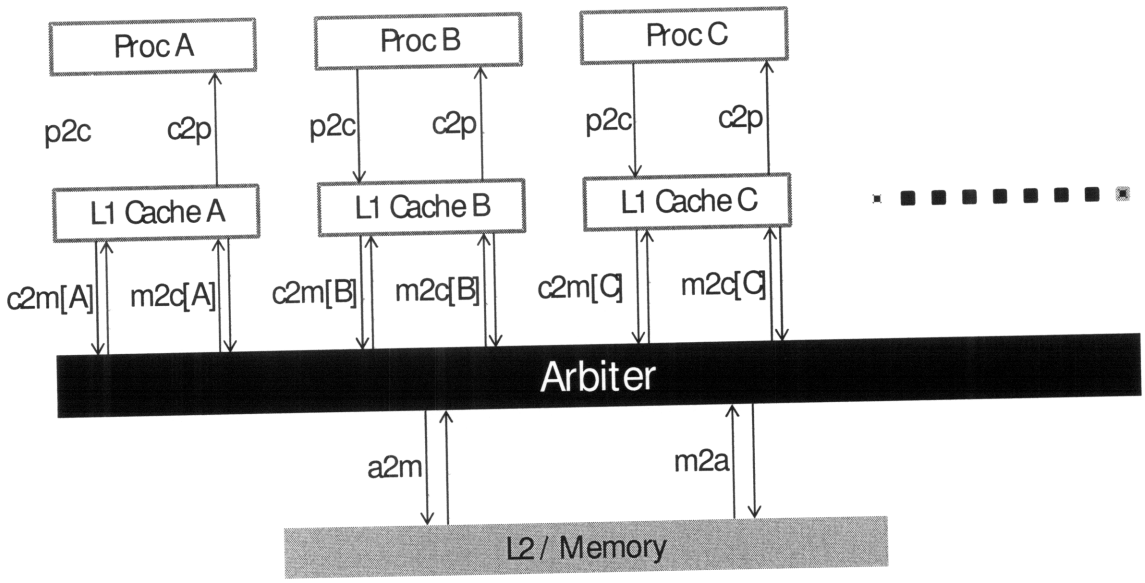

Figure 4-7: Memory Subsystem Architecture

cache and the other is for requests (and associated responses) that originate in memory. Both communication paths are bidirectional (request and response). It is necessary to have both communication paths to avoid deadlock. Otherwise, if only one communication path is used for both types of requests, the following deadlock scenario is possible; if Cache A requests a data block that is currently

in Cache B and Cache B simultaneously requests a data block that is currently in Cache A, each cache will wait for the other to invalidate the appropriate block and will never invalidate their own block. With two distinct communication paths, however, requesting a cached data block from memory and responding to an invalidate request from memory are independent processes that do not interfere with the each other's execution.

A client's request is only dequeued from the appropriate FIFO when the server (which could be either parent or child) has satisfied the request. This is possible thanks to the dual communication path implementation. The major advantage of this technique is that no extra state is needed to recall incomplete tasks. FIFOs will perpetually "remind" the server of any unsatisfied requests. As a whole, this method is simple, efficient and reliable, and is easily expanded to cater for more complicated cache coherence protocols.

The arbiter is responsible for controlling message traffic between children and their parent (in this case between the cores and the L2 cache/memory). Its first major responsibility is to take messages (both requests and responses) from the children and send them, one at a time, to the parent via the appropriate FIFO queues. The arbiter serves all the children in round-robin fashion. Its second major responsibility is to take messages (both requests and responses) from the parent and send them to the right child via the appropriate FIFO queue. The

arbiter deduces the correct recipient of a message from the parent by reading a "destination tag" in the message.

The cache coherence protocol is based on state transitions and message passing. It is implemented as numerous rules specified in terms of guarded atomic actions. I am referring to a guarded atomic action as a set of state updates that must occur atomically with respect to other rules. The main rules of the protocol are described next.

4.2.3 Coherence Protocol Rules

Caching Rules

All following abbreviations refer to figure 5-7.

Requests from Processor/Child

Miss Rule:

```
(p2c.first == LoadReq &&& Miss)
>>>   c2mReq.enq[LoadReq{Addr}];
```

Hit Rule:

```
(p2c.first == LoadReq &&& Hit)
>>>   c2p.enq[LoadResp{Data}];
      p2c.deq;
```

Write-thru Rule:

```
(p2c.first == StoreReq &&& Hit)
>>>   cache.update[Pending {Data}];
      c2mReq.enq[StoreReq {Addr, Data}];
      c2p.enq[StoreResp{Addr}];
      p2c.deq;
```

Write Rule:

```
(p2c.first == StoreReq &&& Miss)
      >>>   c2mReq.enq[StoreReq {Addr, Data}]
```

Responses from Parent

Load Response Rule:

```
(c2mResp.first == LoadResp)
>>>   cache.update[Data];
      c2mResp.deq;
      c2p.enq[LoadResp{Data}];
      p2c.deq;
```

Store Response Rule:

```
(c2mResp.first == StoreResp)
>>>   cache.update[Valid {Data}];
      c2mResp.deq;
      c2p.enq[StoreResp{Addr}];
      p2c.deq;
```

Requests from Parent

Invalidate Rule:

```
(m2cReq.first == InvalidateReq)
>>>   cache.invalidate;
      m2cReq.deq;
      m2cResp.enq[InvalidateResp{Addr}];
```

Directory Rules

Caching Rule 1:

```
(c2mReq[x].first == LoadReq) &&& (memLine.state == Uncached)
>>>   mem.setState(Cached, x);
      c2mResp[x].enq[LoadResp{Data}];
      c2mReq.deq;
```

Invalidate Rule 1:

```
(c2mReq[x].first == LoadReq) &&& (memLine.state == Cached[id_c])
>>>   m2cReq[id_c].enq[InvalidateReq{Addr}];
      mem.setState[Pending];
      c2mReq[x].deq;
```

Write Rule 1:

```
(c2mReq[x].first == StoreReq) &&& (memLine.state == Uncached)
>>>   mem.update({Data});
      c2mResp[x].enq[StoreResp{Addr}];
```

c2mReq[x].deq;

Write-thru Rule:
(c2mReq[x].first == StoreReq) &&& (memLine.state == cached[id_c]) &&& (id_c != req_c)
>>> mem.update{Data};
c2mResp[x].enq[StoreResp];
c2mReq[x].deq;

Invalidate Rule 2:
(c2mReq[x].first == StoreReq) &&& (memLine.state == Cached[id_c]) &&& (id_c != req_c)
>>> mem.setState[Pending];
m2cReq[id_c].enq[Invalidate {Addr}];
c2mReq[x].deq;

Validate Rule:
(m2cResp[x].first == InvalidateResp) &&& (memLine.state == Pending)
>>> mem.setState[Uncached];
m2cResp[x].deq;

Drop Rule:
(c2mReq[x].first == DropReq) &&& (memLine.state == Cached[x])
>>> mem.setState[Uncached];
c2mResp[x].enq[DropResponse];
c2mReq[x].deq;

4.3 Testing

Since the implementation of this memory subsystem occurred simultaneously and in parallel with the development of the processor pipeline, the prototype cache and memory setup was tested using an SMIPS processor. Moreover, the memory module was interfaced with an existing SMIPS. Standardized test code was run on the processor which involved extensive memory accesses. The SMIPS processor's register file was loaded with starting state. At the end of the test code, the contents of the register file of the SMIPS were compared with a correct "image" of the register file state.

Chapter 5

Conclusion

The main challenges in this endeavor concerned parameterization, interfacing and FPGA feasibility. In creating the memory hierarchy and its cache coherence system, the task of parameterizing all dimensions proved to be a difficult task. This is because there is a complex relationship between the different parameters. For example, the tag size depends on the address field size which depends on the size of the memory. Each of these is a parameter. Additionally, there is a complex relationship between the parameters and various processes. For example the replacement process depends on the associativity, the latter being a static parameter. In order to correctly design the system, it was necessary to find all these intricate interactions, understand them and cater for them.

Secondly, interfacing the cache coherence system with the processor pipeline is a challenge primarily because the two systems work at different speeds and have different latencies. It can be seen as the job of linking two asynchronous entities. There must be enough tolerance and book-keeping on both sides of the bridge to enable this interface to function correctly. Finally, placing our synthesized instruction decode function onto FPGA was tricky because the Bluespec interpretation of our initial implementation led to the synthesis of a large area of

combinational logic trees. In order to trim its size, it was necessary to rewrite the instruction decode stage several times. The main approach was to group all inputs that had similar outputs into categories rather than blindly test all inputs for all cases. This synthesized into several smaller decoding trees being synthesized rather than one enormous decoding tree. The sum of the sizes of the smaller trees was significantly less than the size of the original large tree, which led us to choose the categorical approach of decoding an instruction.

As mentioned in the introduction of this document, this work is part of a larger ongoing project, therefore there are many possible expansions and improvements that are worth considering. First of all, the cache system would gain much flexibility if it were to support siblings within the hierarchy. Secondly, the performance of the memory system as a whole would improve if it were to implement non-blocking operations, where fast processes could overtake slower ones. Additionally, a MESI protocol would improve the performance of the memory system at the expense of adding complexity.

Lastly, a challenging but worthy venture would involve the parameterization of the policies and protocols being exercised by the memory subsystem. Such features could include coherence protocol, replacement policy, allocation policy (write-allocate vs. no-allocate) and write-through vs. write-back protocols. Although much of the implementation required to achieve such parameterization would involve the separate and explicit description of the various protocol and

policy options, the area of intersection between descriptions may be large enough to compensate such an endeavor.

References

[1] Arvind and X. Shen, *Using Term Rewriting Systems to Design and Verify Processors,* IEEE, Micro Special Issue on Modeling and Validation of Micro-processors Vol. 19(3): pp. 36-46, 1999.

[2] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Fourth Edition, Appendix A, pp 5-6.

[3] *PowerPC Architecture Book,* http://www.ibm.com/developerworks/eserver/articles/archguide.html

[4] Joe Wetzel, Ed Silha, Cathy May and Brad Frey, *PowerPC User Instruction Set Architecture,* Book 1, Version 2.01, September 2003.

[5] Joe Wetzel, Ed Silha, Cathy May and Brad Frey, *PowerPC Virtual Environment Architecture,* Book 2, Version 2.01, December 2003.

[6] Joe Wetzel, Ed Silha, Cathy May and Brad Frey, *PowerPC Operating Environment Architecture,* Book 3, Version 2.01, December 2003.

[7] Kattamuri Ekanadham, Jessica Tseng and Pratap Pattnaik, *IBM PowerPC Design in Bluespec,* March 25, 2008.