A Shared-Memory Multiprocessor System Using the Raw Tiled Architecture

by

Levente Jakab

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering

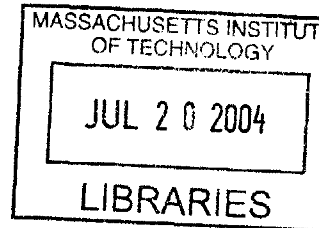
and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

© Massachusetts Institute of Technology 2004. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 13, 2004

Certified by ....
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

BARKER

A Shared-Memory Multiprocessor System Using the Raw Tiled Architecture

by

Levente Jakab

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Recent trends in microprocessor design have moved away from dedicated hardware mechanisms to exposed architectures in which basic functionality is implemented in software. To demonstrate the flexibility of this scheme, I implement a shared memory system on Raw, a tiled multiprocessor. A traditional directory-based cache coherence system is used. The directories are fully resident on several tiles, and the remaining tiles act as users, with cache maintenance routines accessed via interrupt. Previous implementations of shared-memory systems have mostly relied on a combination of dedicated hardware and user-enabled software hooks, with one notable exception, Alewife, combining basic hardware with software support for corner cases. Here, the focus is moved to placing as much support for basic cases into software as possible. The system is designed to minimise custom hardware and user responsibilities. I prove the feasibility of such a design on an exposed architecture such as Raw.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to begin by thanking Anant Agarwal for supervising me, prodding me into productive directions (such as writing instead of debugging), and coming up with lots of good ideas.

Michael Taylor and David Signoff cleared many initial hurdles and demonstrated the workability of the concept of a shared-memory implementation in software, and were always happy to answer late-night emails. The rest of the Raw group, especially Jim Psota, Patrick Griffin, Dave Wentzlaff, Ian Bratt, Ken Steele, and Jason Miller, were instrumental in demonstrating correctness leading me away from pitfalls, and suggesting nifty hacks.

All my friends, for all their triumphs. Cristina, for far too much to name here. This thesis is long enough already. Matt, for forcing me to make sense at all times, even when I'm driving. Katie, and her Texas-sized repository of common sense, for getting me to realise that graduation was a Good Idea. Ali, for the all-star game. Shouka, for forcing me to use airplanes every once in a while. Alex, for the burning of Rome. Aerik, for rainbow skies. Tim, for Irvington. Cube, for being my nemesis. We'll see what October brings. Boggs, for his impulse decisions. Kate, for Felony Red and other brilliant ideas. Justin and Shawn, for getting me inebriated sufficiently often to keep me out of real trouble. Archibald, for Fenway Park. Dan, for 9W in New York, and not getting me killed thereon. Jen, for assuming I'm a model employee. Kyle, for the vilest of insults. Nate, for being awesome. If I forgot you... it's not like anyone's reading this anyway. See you all in California.

Those that inspired me to excel, or at the very least, to stay out of the wrong half of the bell curve. Marty Badoian, my high school math teacher, who managed to instill the concept of hard work into a fourteen-year-old punk kid. Ted Williams, baseball legend, for being a perfectly valid ideal standard of humanity. All the great metal bands that play on "11", for playing on "11".

Anne Hunter deserves monster amounts of recognition, not just from me but from all EECS students, for her tireless work in making sure everyone graduates, even in the face of looming bureaucracy.

My research is supported by the benevolence of the MIT Department of Electrical Engineering and Computer Science, and by the U.S. Department of Defense.

Contents

1 Introduction **15**

1.1 Shared Memory as a Type of Architecture 17

1.2 The Cache Coherence Problem 20

1.3 Migration to Software-Based Shared Memory Management 21

1.4 Overview of the Raw Processor Architecture 24

2 High-Level Overview of a Shared Memory System **27**

2.1 Shared Memory in the Presence of Local Caches 28

2.1.1 Conditions which Cause Cache Incoherence 29

2.1.2 A Snoopy Cache Scheme 29

2.1.3 A Directory-based Cache Scheme 30

2.1.4 Ways of Maintaining Cache State 32

2.1.5 State Transitions in a Full-Map Directory 34

2.2 Processors and Directories 35

2.2.1 Directory Responsibilities 36

2.2.2 Processor Responsibilities 36

2.3 Private and Shared Memory 37

3 Implementation on a 16-Tile Raw Processor **39**

3.1 Raw Hardware Overview 39

3.1.1 The 16-Tile Raw Chip 41

3.2 Design Objectives 47

3.3 Implementation Redux 47

3.3.1 Read of Shared Data from Main Memory 48
3.3.2 Store of a Word to Local Cache 49
3.3.3 Read of an Exclusive Word 50
3.3.4 Multiple Writes of Exclusive Lines 50
3.4 Overview of FPGA Functionality 53
3.5 Communication between Users and Directories 55
3.5.1 Interrupts on the User Tile 55
3.5.2 Interrupt Controllers 55
3.5.3 Communication from a Directory to a User 56
3.5.4 Communication from a User to a Directory 58
3.6 Low-Level Analysis of System Tile Functionality 58
3.6.1 Multitasking 59
3.6.2 Directory Management 65
3.6.3 Interrupt Control 70
3.6.4 Memory Requirements of the System Tile 73
3.7 Low-Level Analysis of User Tile Functionality 73
3.7.1 External Interrupts 74
3.7.2 Internal Interrupts 75

4 Correctness of the Raw Implementation 79

4.1 Transparency of Interrupts 80
4.2 Absence of Buffer Overflows 81
4.3 Correctness of State Transitions 83
4.4 Deadlock Avoidance 85
4.4.1 Dependencies Among Components 85
4.4.2 Dependencies Involving Cache Line Sharing 88
4.4.3 The General Dynamic Network is Read Sufficiently Quickly . 89
4.4.4 Correctness of the Deadlock Bypass 90
4.4.5 Execution in a Deadlock-Free Environment 92
4.4.6 Absence of Further Deadlocks 93

4.5 Sequential Consistency 94
4.5.1 Single-tile Ordering 94
4.5.2 Sequential Consistency Across Multiple Processors 94
4.6 Summary of Programmer Responsibilities 96
4.6.1 Known Network Hazards 96
4.6.2 Hazards due to Raw Hardware Restrictions 97
4.6.3 Restrictions on Stores due to Event Counter Limitations . . . 97
4.7 Implications of Programmer Responsibility 98

5 Performance of the Shared Memory System 99

5.1 Methodology 99
5.2 Results 100
5.2.1 Low-level Results 100
5.2.2 Service Times for Requests 103
5.2.3 Read Requests 104
5.2.4 Write Requests 104
5.3 Application Results 105
5.3.1 Application Overview 105
5.4 Analysis of Results 107
5.4.1 Profiling of User Tile Activity 108
5.4.2 Overall Frame Rate 109
5.4.3 Why the Application is Slow 109

6 Further Development 111

6.1 Hardware Changes in Raw to Facilitate Shared Memory 111
6.1.1 Handling Cache Misses in Software 111
6.1.2 Improved Awareness of Writes 112
6.1.3 Identification of Interrupt Sources 112
6.2 Expanding to Larger Raw Processors 113
6.2.1 Necessity of More System Tiles 113
6.2.2 Routines that Fail for More than 16 Tiles 115

6.3 Support for Application Development 116
6.3.1 Compiler Support for Shared Memory 116
6.3.2 Support for the OpenMP Library 118

7 Conclusion 121

A Raw Code 123

A.1 System Tile Code 123
A.2 User Tile Code 157
A.3 Application Code 174
A.3.1 Distributor Code 175
A.3.2 Reassembler Code 183
A.3.3 Processor Code 185

List of Figures

1-1 Traditional and Modern Parallel Processors 16

1-2 An Abstract View of a Shared Memory System 18

1-3 The Fundamental Difference between Shared Memory and Message-Passing Systems[1] 19

1-4 Shared Memory Architecture with Local Caches on Each Processor . 21

1-5 Distributed Shared Memory with Directories Enforcing Coherence . . 22

1-6 Various Cache Coherence Schemes 24

1-7 One Corner of an Arbitrarily Large Raw Fabric 25

2-1 A Snoopy Cache Coherence Scheme 30

2-2 A Directory-based Cache Coherence Scheme 31

2-3 The Censier and Feautrier[7] Directory-based Cache Coherence Scheme 34

3-1 High-Level Schematic of the Raw Handheld Board 40

3-2 A Single Raw Tile 41

3-3 Dynamic Routing of a Message from Between Processors 43

3-4 Routing on the Static Network 44

3-5 Servicing a Cache Miss on a Load 45

3-6 Various Cache Coherence Schemes, Revisited 47

3-7 Dividing the Raw Handheld Board into Components 48

3-8 Read of Shared Data from Main Memory 49

3-9 Store of a Word to Local Cache 50

3-10 Read of an Exclusive Word by Another Tile 51

3-11 Multiple Writes of Exclusive Lines 51

3-12 Functionality of the FPGA 53
3-13 Directory and Interrupt Controller Responsibilities of System Tiles . 56
3-14 Communication from Directory to User Tile via Interrupt Controller . 57
3-15 The Main Loop of a System Tile 60
3-16 Context Switches in the User-Directory Communication Sequence . . 61
3-17 Intermediate States to deal with Non-Atomic Transactions 62
3-18 Handling of MDN ("New Writer") Messages from a User Tile to the Directory 65
3-19 Handling of GDN Messages from a User Tile to the Directory 68
3-20 Handling of Messages from the FPGA to the Directory 69
3-21 Interrupt Controller State Transitions 71
3-22 Deadlock Resolution by the Interrupt Controller 72
3-23 Interrupt Handler for External Pings (Interrupt 3) 74
3-24 Interrupt Handler for Event Counter Events (Interrupt 6) 76

4-1 All Dependencies in the Shared Memory System 88
4-2 Cycle-Free Dependency Graph 93

5-1 An Edge-Detection Application on the Shared Memory System 106

6-1 A Possible Design for an 8x8 Raw Processor 114
6-2 Division of Labour among System Tiles on an 8x8 Shared Memory System 114

7-1 A Software-based Cache Coherence Scheme compared to Other Schemes 122

List of Tables

2.1 Performance of Various Directory-Based Cache Coherence Schemes . 32

3.1 State Transitions due to Asynchronous Inputs 63

4.1 Dependencies Between Networks 82

4.2 Responses to All Stimuli, Possible or Otherwise 83

4.3 Components of the System 86

4.4 Dependencies Among Components 87

5.1 Cycle Times of Various Routines on the System Tile 101

5.2 Cycle Times for the Components of the Interrupt 6 Handler 102

5.3 Cycle Times for the Components of the Interrupt 6 Handler 103

5.4 Latencies of Service Calls in Various States 103

5.5 Time Spent on Various Activities 108

Chapter 1

Introduction

Parallel processor architectures have historically been divided into several major classifications, based on the manner in which they transmit information between processing modules. Historically, the most prevalent classifications have been the *shared memory*, *message-passing*, and *data parallel* models [1]. Of these three, the shared memory and message-passing models feature the largest amount of independence between processing elements. There exist other models, such as the *vector* or *dataflow* models, but those tend to share many features with one or more of the models described above.

In the data parallel model (and its close relatives, the *vector* and *systolic* model), control tends to be centralised to a single instruction issue unit. The other two models can have multiple independent issue units, and thus be viewed as a set of uniprocessors operating in parallel.

Architectures with independent processing units tend to implement features from both the shared memory and message-passing models, and historically these features were fixed at design time. More recent processors, however, have been designed with flexibility in mind (Figure 1-1). One of these designs is the Raw[50] architecture. The underlying fabric of interconnects is exposed to software, and therefore this architecture can be configured to be a wide variety of parallel-processor models, including the ability to support an arbitrary combination of shared memory and message-passing structures. In this sort of architecture, fundamental computational operations, such

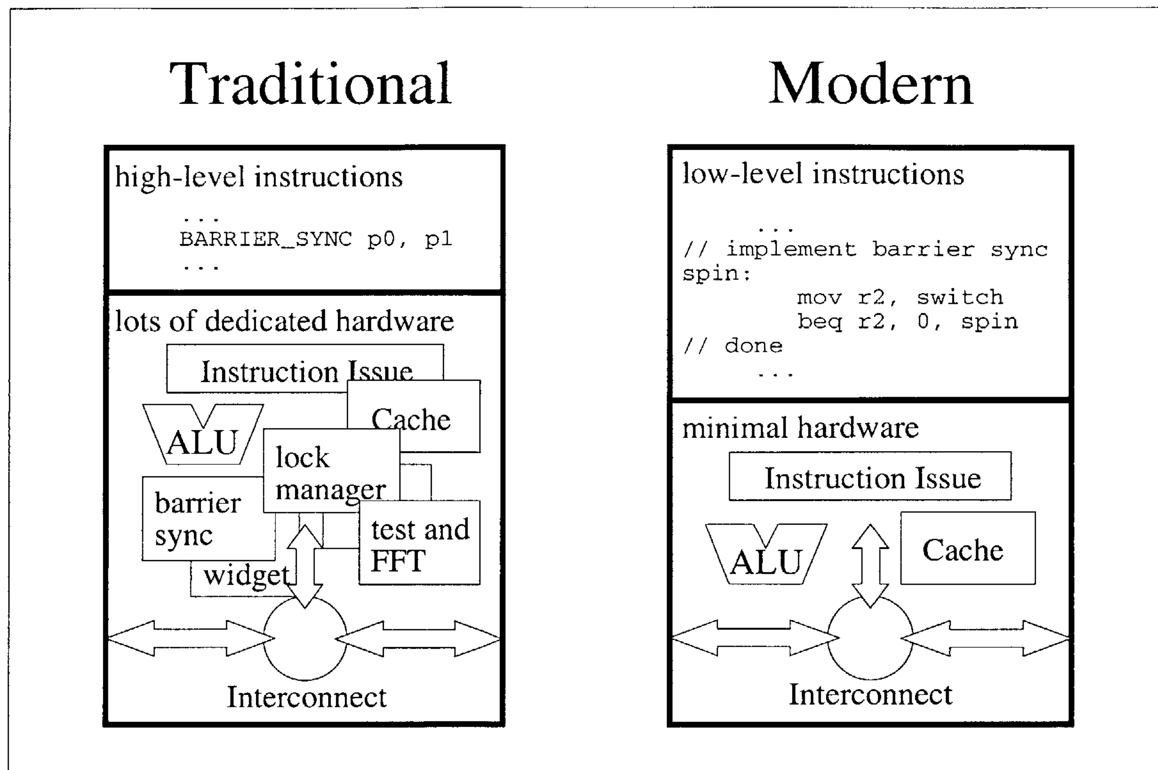


Figure 1-1: Traditional and Modern Parallel Processors

as the read-modify-write cycle that is the staple of a shared memory system, are implemented entirely in software. The hardware provides only the basic ability to process and transmit data.

Therefore, the Raw architecture can be split into two layers of abstraction: a software layer on top of a hardware layer. At this time, the Raw group has only started to explore the idea of building the fundamental nature of a machine in software. A streaming processor model has been developed first: the programming langauge StreamIt[29, 19] was developed on the Raw architecture and expanded to support other communication-exposed architectures. Work is being done on a message-passing system[40] as well.

In this thesis, I describe an implementation of a shared memory system for the Raw architecture. First, I outline some background in the way of shared memory development and functionality, including the recognition and solution of the cache coherence problem, and then the migration of system specifics from hardware to

software, and development of processors that support this flexibility.

In subsequent chapters, I describe in more detail the traditional implementation of a shared memory system, as well as how this implementation is executed specifically on the sixteen-processor Raw Handheld[48] architecture. I then analyse this implementation, both in terms of correctness and performance. First I prove that the cache coherence problem has been fully solved: namely, that this system is as correct as one that does not have cache coherence hazards at all. Then, I show that this system supports certain inter-processor communication primitives. Finally, I provide some performance numbers both in terms of fundamental processing operations and structures, as well as applications that make use of them. I then conclude with a chapter on future development, and the expansion of the system to related Raw architectures.

1.1 Shared Memory as a Type of Architecture

The shared memory model, in its purest form, is a set of processing nodes that communicate amongst themselves by reading and writing data and control values in a single, homogenous, block of memory (Figure 1-2)[1]. Each location is equally accessible by all nodes, so there is an interconnect network that can be visualised as existing between the processor and the main memory.

This is in opposition to the message-passing model, in which processors communicate with each other through an interconnection network. Main memory is distributed among the processor nodes, and a request for a "distant" address (i.e. one not on the local node) must be made to the processor that controls it[1].

In practice, shared memory systems also have memory blocks that are tied to processors physically, but the interconnection network still accesses them directly. The key difference between the two frameworks is best shown in Figure 1-3.

The concept of shared memory is as old as that of parallel processing itself. Computers with multiple logic units, but controlled by a single instruction stream, were built as early as the 1950s (the IBM 704[24]). However, it was a few years later

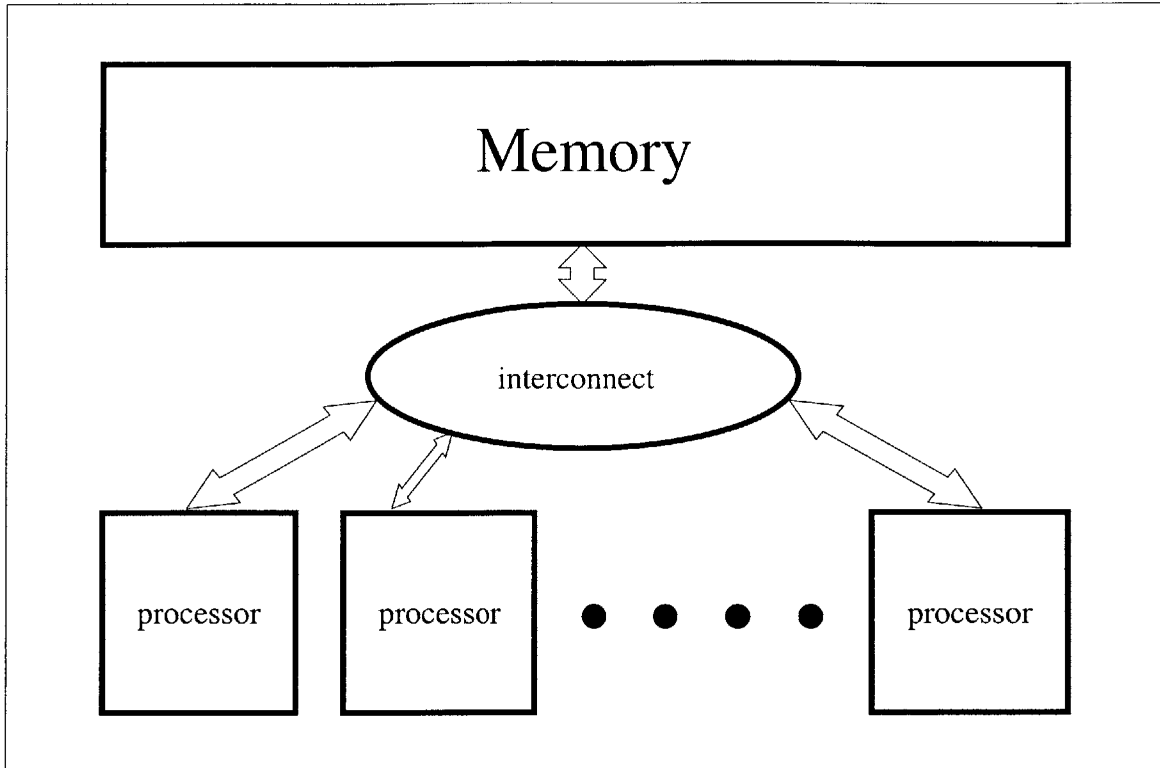


Figure 1-2: An Abstract View of a Shared Memory System

that the idea of multiple computational units was developed. E. V. Yevreinov's OVS ("Homogeneous Computing System") [56], a theoretical framework designed in 1960, is remarkably similar to modern parallel processors, in that it describes several identical computing tiles, geographically aware, and connected via a tunable interconnection network[53]. It was a message-passing machine, with each node containing both processing and memory units.[1] The first shared memory computer was probably the Burroughs D825[54], developed in 1962.

Soon enough, theoretical analysis caught up with *ad hoc* design, and the protocols and limitations of shared memory were explored and formalised. In 1962, Dekker[12] first devised an algorithm by which two processes could contend in a fair manner over blocks of shared memory without damaging each other's consistency.

[1]On an abstract level, Yevreinov's OVS looked very much like today's tiled parallel processor architectures. However, as a practical implementation, it resembled more a Beowulf cluster[32]. The first working OVS system, the "Minsk-222" from 1966, connected one Minsk-2 and two Minsk-22 machines, each a fully functional unicomputer[53]. The inhomogenity of the three nodes was a logistical matter: three entire computers represented a substantial portion of the USSR's technological resources!

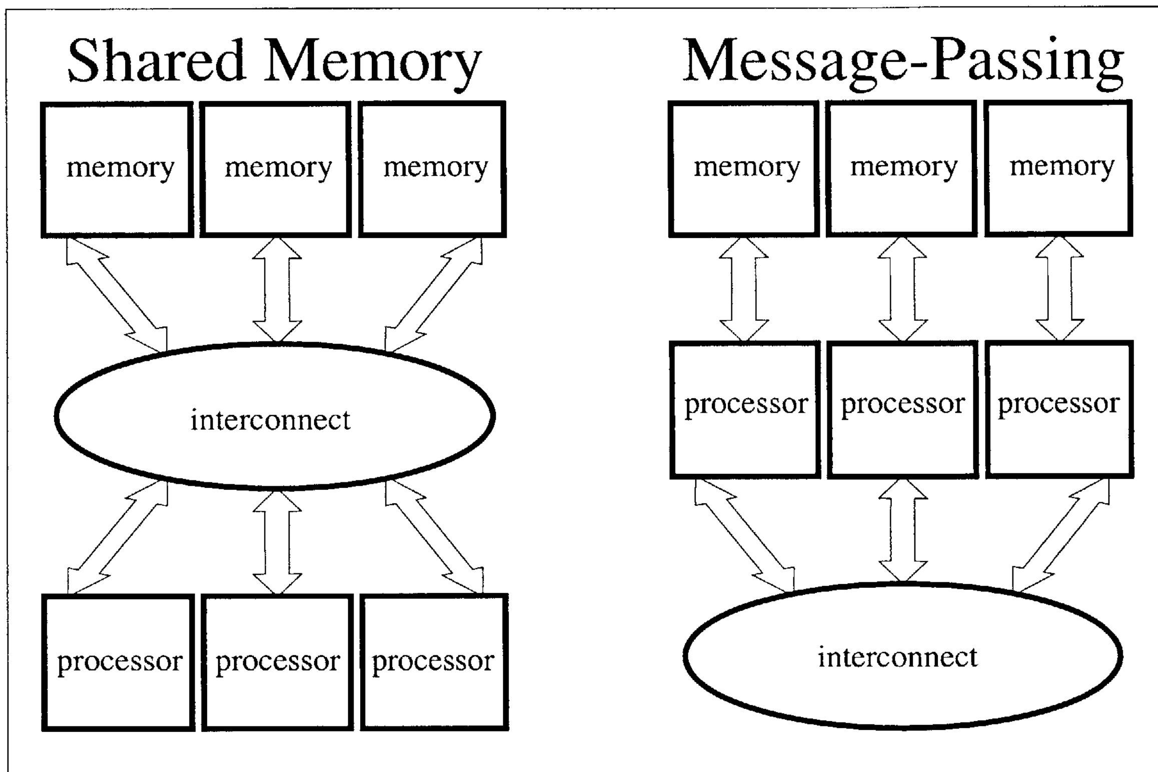


Figure 1-3: The Fundamental Difference between Shared Memory and Message-Passing Systems[1]

In 1965, Edsger J. Dijkstra formalised Dekker's research, and expanded it to an arbitrary number of processes. He introduced the concept of "critical sections"[13], which described how processes (either on separate physical processors, or time-shared on a single one) can share memory in such a way that no two of them are attempting to access a single location at the same time. By this time, "computer coupling"[13] had become a well-known field of research.

1.2 The Cache Coherence Problem

The next relevant development in processor architectures involved the development of memory hierarchy. IBM's J. S. Liptay in 1968[33] came up with the idea of a small, high-speed memory that would store recently accessed words from main memory, so that they could be reaccessed much faster. Due to cost issues, the entire memory could not be made fast, but this small *cache* could take care of most of the memory accesses.

This memory hierarchy made its way to parallel processors, and the most common shared memory architecture became as shown in Figure 1-4. Each processor had a local cache in which were stored the values it had accessed most recently.

However, this architectural model manifests the *cache coherence problem*[7]. If main RAM is read-only, then the data stored in any cache will be the latest version. However, introducing writes allows for one processor to have its local data value be fresh, while the corresponding data in another processor is stale. Correct behaviour of the system would imply that "the value returned on a load instruction is always the value given by the latest store instruction with the same address"[7]. Clearly, if one processor's cache has a later version, and another processor is unaware of this, then the system will behave incorrectly. Even writing back to main RAM is not sufficient: the reading processor's cache must be explicitly notified of the invalidity of its data.

There are two main families of solutions to the cache coherence problem. One involves having each cache "snoop" transactions going on in main RAM, as well as in other processors. This known as a *snoopy scheme*[18]. The other method is the

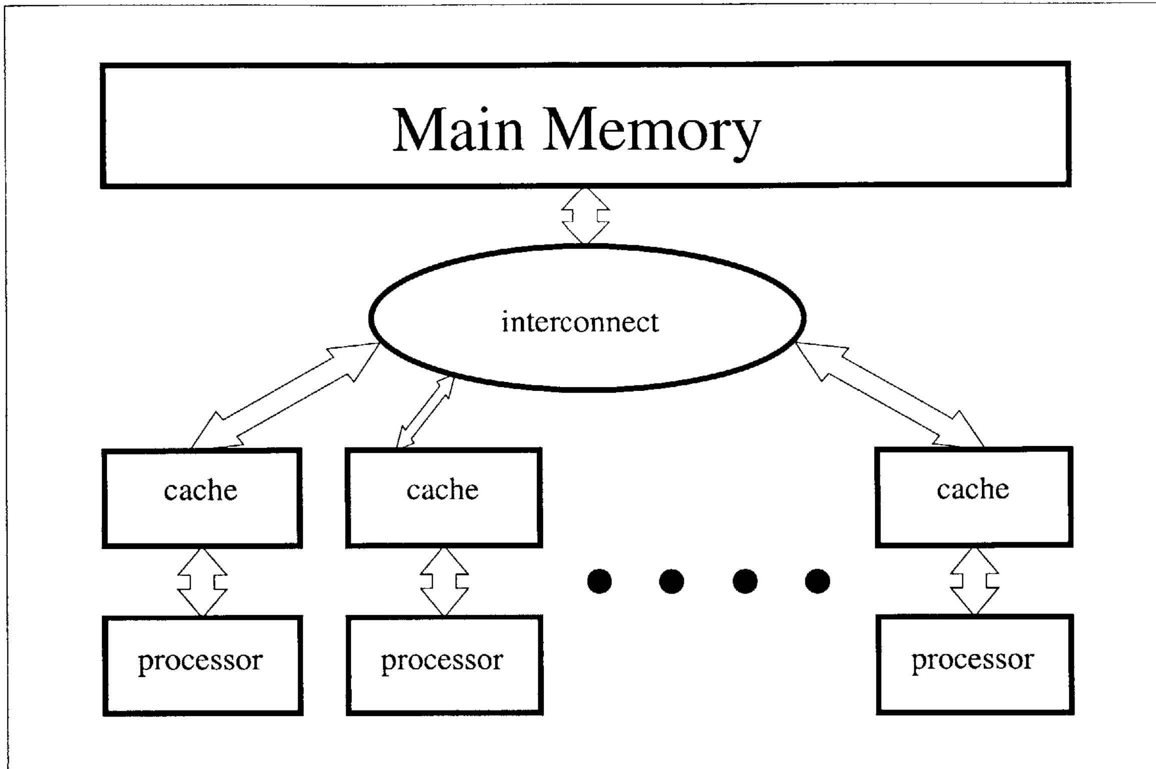


Figure 1-4: Shared Memory Architecture with Local Caches on Each Processor

directory scheme[46, 7]. In this design, hardware elements actively monitor transactions and maintain information about which processor is caching which addresses. In the event of incoherence, the directory hardware actively reestablishes coherence by communicating with the caches. The snoopy scheme requires smaller amounts of overhead for a small number of processors, but the directory scheme scales better as the number of processors increases. I will analyse both of these schemes in further detail in the next chapter.

1.3 Migration to Software-Based Shared Memory Management

The first solutions to the cache coherence problem involved dedicated hardware. This method proceeded as late as 1990, (the Stanford DASH[31]), with software support coming in the way of user-level hooks that determined either at compile

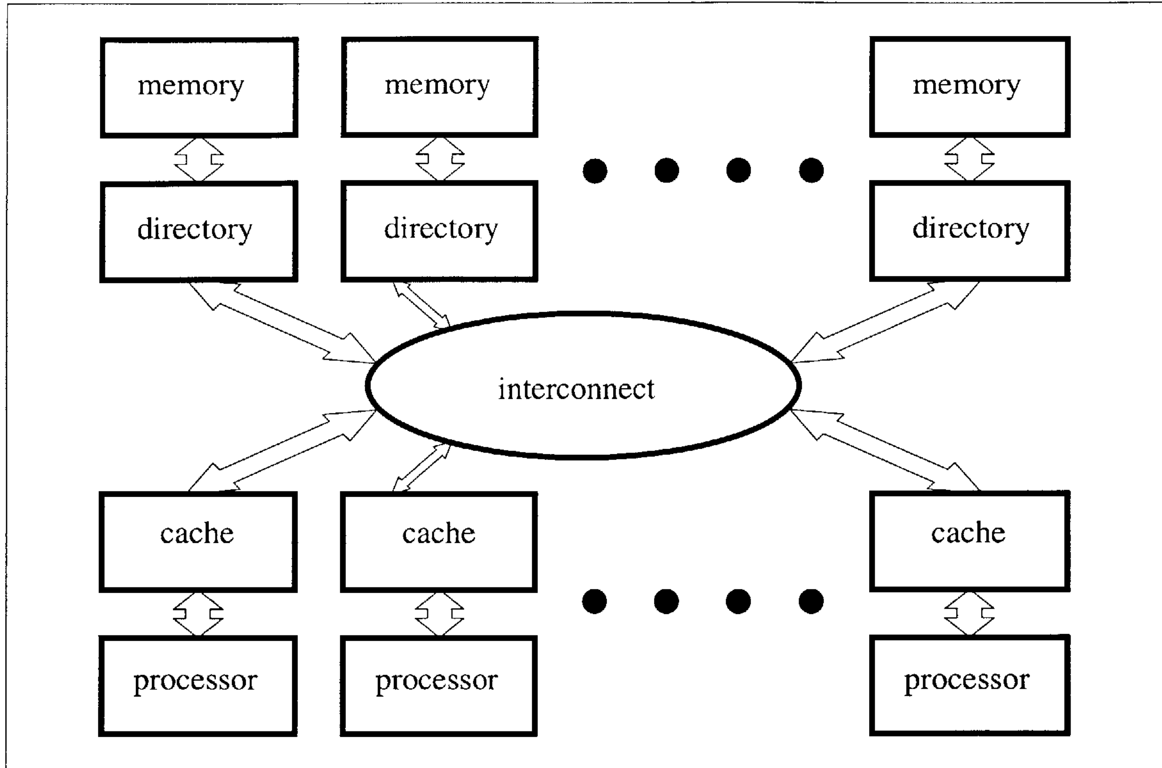


Figure 1-5: Distributed Shared Memory with Directories Enforcing Coherence

time which memory would or would not be shared (Carnegie Mellon's C.mmp[17], NYU's Ultracomputer[15], McGrogan *et. al.*[35]), or allowed the user to insert explicit cache operations (Smith *et. al.*[45], Cytron *et. al.*[11], UIUC's Cedar[27], Compaq's Hector[52]). An evaluation of some of these software-based schemes was done by Owicki and Agarwal[39]. It was left to the hardware, however, to process the actual memory requests.

By the early 1990s, shared memory architectures put not only the processing elements, but also the main memory, into interconnected nodes. Abstractly, the memory would still be viewed as one monolithic entity, but it physically was distributed among the nodes. Furthermore, directory hardware was distributed as well (see Figure 1-5), thus alleviating the bottleneck caused by ever-growing central structures[4].

However, the standard directory model still implied a total complexity that would increase with both the amount of main memory, and the amount of caches to be taken care of (an effective $O(n^2)$ scaling factor), and thus hardware complexity and

the lack of scalability became more and more of an issue. Hardware could support common cases, involving data existing only in a small number of caches, but planning for extenuating cases became too overwhelming.

The idea of handling these cases in a scalable manner through user-transparent software was first developed by Chaiken *et. al* in the Alewife system at MIT[9, 8]. Alewife's hardware-based directories allowed for a small number of caches to be monitored, and when this number was exceeded, software would be called to take care of the rest. This development differed greatly from previous software schemes. By setting the number of pointers (the number of caches kept track of in hardware to zero), the system effectively became almost completely software-based, with the only hardware necessity being an event detector. This was a first.

A slightly different approach was taken by Stanford's FLASH architecture[28], which had dedicated hardware that could be programmed to support a variety of communication schemes, including shared memory or message-passing. However, this hardware was not general-purpose, so I treat the programming not as software, but as firmware.

Figure 1-6 shows the nature of various past implementations. The total level of functionality is split into three components: hardware[2], user software, and system software. It is important to note that most of the solutions are on one edge of the triangle, as they lack system-level, user transparent software hooks. The software support consists of either the relatively easy task of determining at compile time which memory is to be shared, and not caching it, or the more involved insertion of user-level cache operations[3]. The only system that is not on that axis is the Alewife system[9, 8].

Therefore, the idea of managing shared memory *entirely* in user-transparent software. The Raw processor, having been designed with flexibility in mind, is capable of this sort of memory management. In the next section, I give a brief overview of the Raw architecture.

[2]This includes the firmware of the FLASH architecture.

[3]The borderline case of having no cache coherence at all is also shown: in this case, the user must discover all of the coherence for himself!

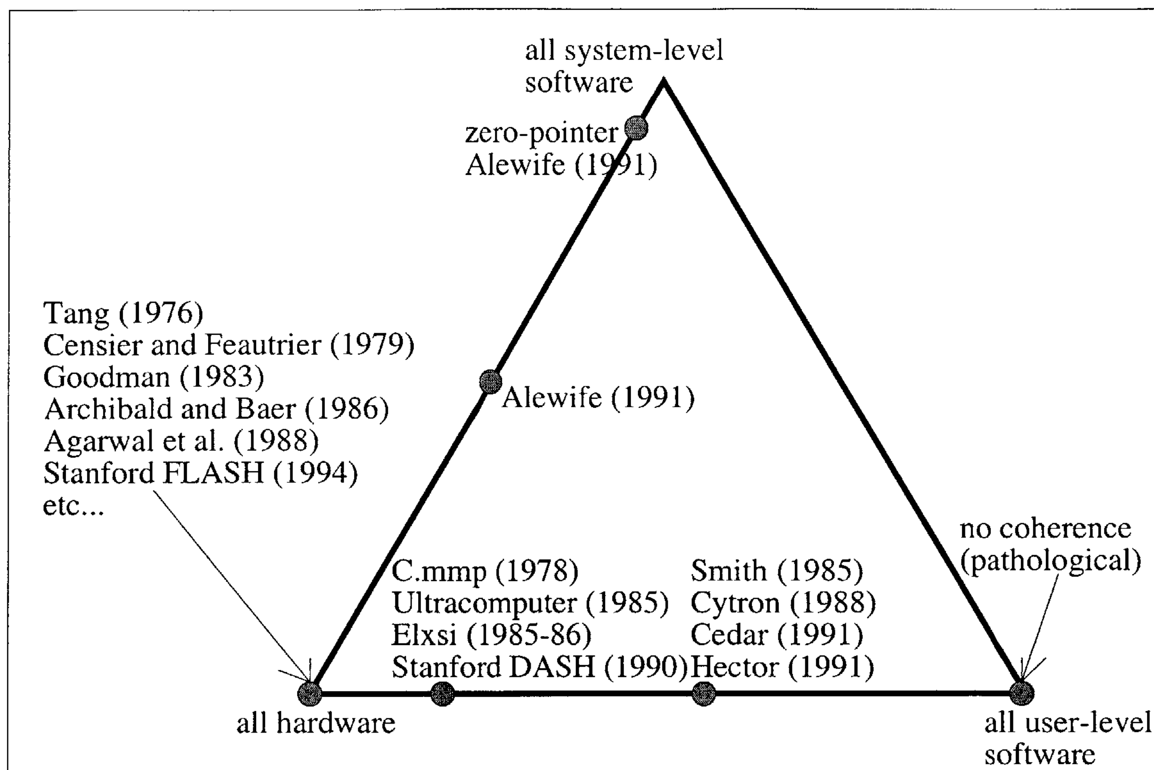


Figure 1-6: Various Cache Coherence Schemes

1.4 Overview of the Raw Processor Architecture

The Raw architecture consists of a two-dimensional square mesh of identical interconnected tiles (see Figure 1-7). Each tile consists of a main processor and a switch to communicate with other tiles.[48] This architecture is, in an abstract sense, similar to other recent tiled architectures, such as Wisconsin's ILDP[26], Stanford's Smart Memories[34], and UT Austin's Grid[37, 42].

Many of the design decisions of Raw were motivated by the increasing importance of wire length in VLSI design. Thus, the tile is made as small as possible, so that the distance between two tiles is minimised. This allows the clock speed to be increased.

The switches communicate with each other, and with the processors they serve, via a set of routing networks, which are either "static" or "dynamic". The dynamic network is a standard dimension-ordered network that allows communication between any arbitrary node, but with a small overhead of header generation. The static network allows fast near-neighbour communication of single-word scalar operands[51].

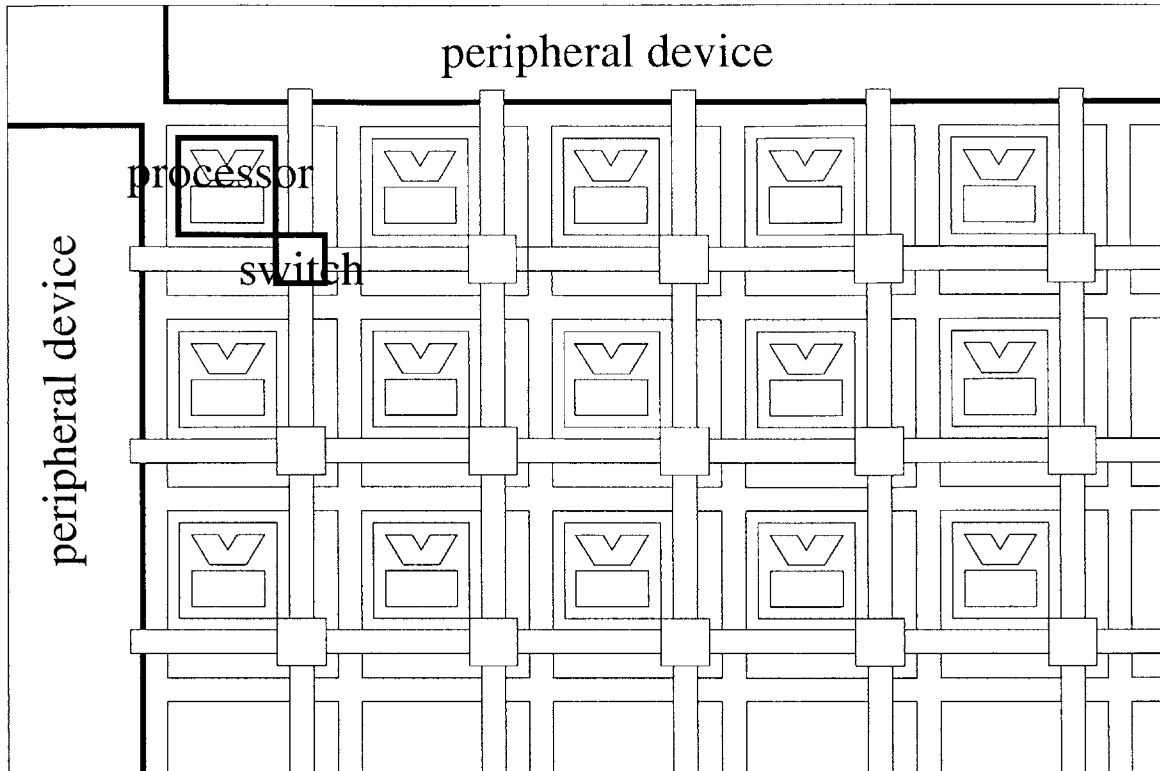


Figure 1-7: One Corner of an Arbitrarily Large Raw Fabric

The switch is its own small processor, and the routing of the static network is compiled as a separate instruction stream. The four networks also extend off the edges of the mesh, allowing for communication with peripheral devices.

Another motivation for the design of Raw is its flexibility. The low-level elements (networks, ALUs, caches, etc) are all exposed to software, allowing for fine-grained control. There are, in fact, no high-level hardware elements, such as directory controllers or lock managers. Thus, Raw is a modern architecture, as described earlier in this chapter. It can be made to be any traditional architecture, simply by loading new software.

Therefore, a software-based shared memory system is a possibility on the Raw architecture. In the next chapter, I give a somewhat more detailed overview of the fundamentals of a shared memory system, detailing various requirements that must be met. In the chapter following, I implement a design on Raw that meets all of those requirements.

Chapter 2

High-Level Overview of a Shared Memory System

In this chapter, I describe the fundamental operations of a shared memory system. I take as an abstract ideal the system in Figure 1-2: N identical processors access, via an interconnection network, a monolithic block of RAM. I first note some of the problems inherent even to this abstract model, and then discuss the implementation of low-level hardware that makes the real system appear like the ideal one, from the perspective of each processor.

In the ideal shared memory model, all communication between processing nodes takes place by reading and writing of a large block of memory. There is no direct inter-processor message passing.

Fundamental to the idea of shared memory is that of *sequential consistency*[30]. This is summarised in short as: "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"[30]. In the context of shared memory, the operations are reads and writes. All of the processors must see the same sequence of reads and writes, and the accesses by individual processors must appear in the entire sequence in the same order that they do in the program flow of the individual processor.

The events that are asynchronous to each other (because they are issued by differ-

ent processors) are declared unorderable, because, due to variable delays in routing, it is impossible to predict which will reach a controller first. This is a difficulty of physics as a whole. Given the existence of a minimum time of causality (the time taken by a message to go from one point to another), we must declare two events to be unorderable if, from *any* point's perspective, the two events could be reversed in order due to delays in communication. This idea was expressed in a slightly different manner by none other than Albert Einstein[16][1].

The particular order chosen for the events is not important, so long as the order is reflected equally on all processors. This is a function of the shared memory controller. It must not allow a processor to continue to new instructions until the ramifications of its current instruction are felt by all other processors[30, 3].

Some care must be taken in utilising this model. For example, it is clear that there must be some cooperation between two processes[2] writing to the same address. If this writing is non-deterministic, then so is the final value written. Therefore, we must note the use of synchronisation mechanisms [13, 14], and the underlying properties needed for them to work correctly.

2.1 Shared Memory in the Presence of Local Caches

The main problem that must be solved by all modern shared memory implementations is that of cache coherence. A cache, by definition, replicates state, and when that state changes, all the reflections must change as well, or otherwise the system becomes internally inconsistent. I first analyse the conditions under which cache coherence can arise. I then discuss the snoopy-cache[18] and directory-based[46, 7] solutions, choosing to focus on a directory-based protocol, due to its improved scaling properties.

In this thesis, main memory has a granularity of a "cache line", which is one,

---

[1] I cited Einstein in my thesis. My life is complete.

[2] In this chapter, I use the term "process", instead of "processor". I assume that processes are maximally distributed: namely, at most one per processor. Allowing multiple processes on a single processor would trivialise the discussion somewhat, as certain ordering properties would be made implicit by the serial instruction issue nature of a single processor. I note the existence of superscalar processors, but note further that they have deterministic ordering properties, if designed correctly.

or usually more, words that are sent to a cache from main memory upon a single cache miss[3]. In every cache-coherence scheme, the state of every possible cache line is maintained, but not any substate of individual words. If one word in a cache gets modified, then the entire line is considered "dirty" (modified, and must be written back to main RAM)[22].

2.1.1 Conditions which Cause Cache Incoherence

For the purposes of this discussion, I will assume a block of main memory, a writing processor, and a set of reading processors. This memory will have one address (one cache line), which is initialised to some value A. Since each address is treated individually, and may or may not be in a particular cache independent of every other address, the discussion scales to multiple addresses without modification. First, all of the processors (including the future writer) read in this value, and thus A is shared among all readers and main memory. Since main memory is reflected accurately in each cache, our system is coherent.

Now, let us assume that one processor writes a new value, B, to the address. This write will be reflected in its local cache, and possibly in main RAM. This depends on the nature of the cache. If it is "write-through"[22], then immediately upon the write, the data is written back to main RAM. If it is "write-back", then the data is only written back when the cache line is evicted. However, under no circumstances is the data written to the other caches. Therefore, the system is incoherent. Either the other caches, or the other caches and main RAM, are out of date. Therefore, an active correction mechanism is needed.

2.1.2 A Snoopy Cache Scheme

The snoopy-cache scheme is implemented most easily on a system that has a bus between all of the processors and the main memory. Each cache is set to monitor all

[3]Sending multiple words on a single miss is, of course, advantageous due to anticipated spatial locality[22]. If we want a word at a given address, we will probably want the words at nearby addresses too.

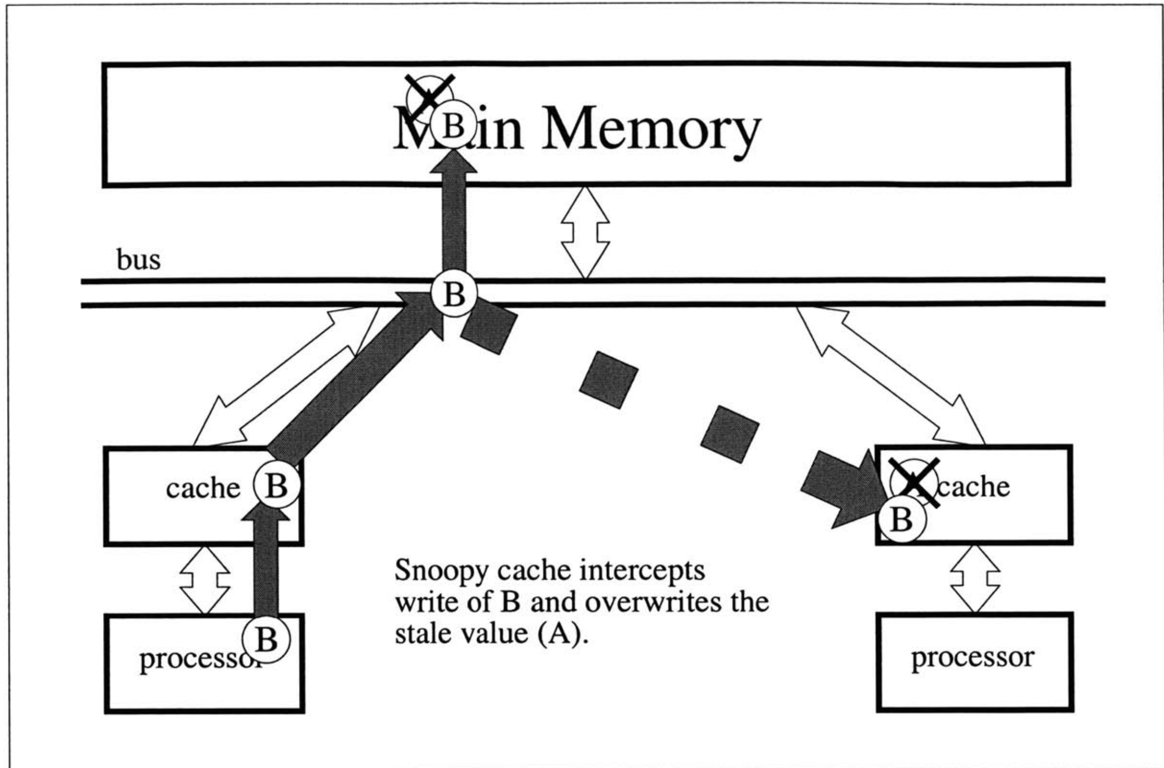


Figure 2-1: A Snoopy Cache Coherence Scheme

writes on the bus, and in the case of a write causing incoherence, the cache grabs the freshest value and stores it locally. This scheme clearly works only in the case of a write-through cache, since the caches are not expected to snoop the wires between each processor and its corresponding local cache. The operation is shown in Figure 2-1. Main memory starts with value "A", and the processor on the right reads it. When the processor on the left writes "B", and this is written through to main RAM, the cache on the right intercepts and replicates the write. Coherence is maintained.

The main disadvantage of the system is that it scales just as poorly as any bus-based multiprocessor does. Furthermore, since I will be implementing a shared memory system on Raw, which is not bus-based, we will no longer consider it.

2.1.3 A Directory-based Cache Scheme

Figure 2-2 shows a directory-based cache scheme for a write-back cache. The directory stands between the processors and the memory that it serves; thus, all memory

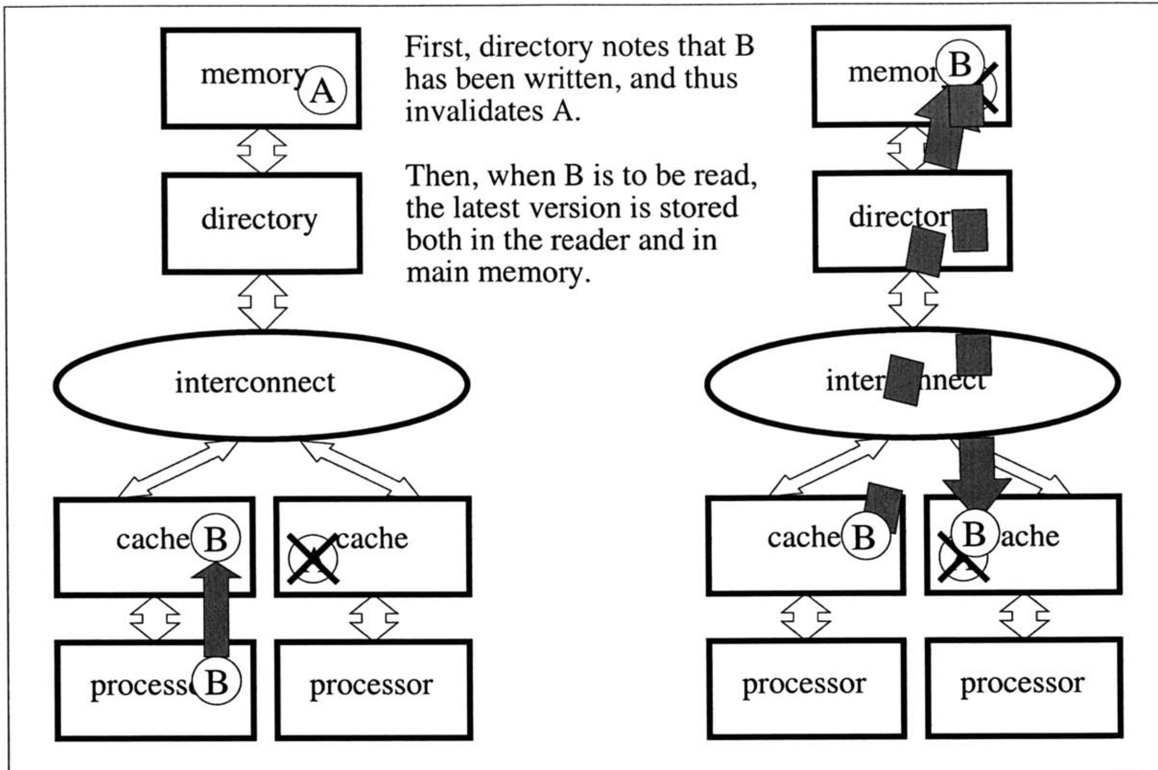


Figure 2-2: A Directory-based Cache Coherence Scheme

requests must go through it. In the case of multiple banks of memory, there are multiple directories. Each directory is mapped to a particular set of addresses, and serves all processors[4].

Again, the memory starts with data "A", which is read by a processor. When the value "B" is written to a local cache, the directory notes this, and also invalidates the stale "A" in the other processor. Main memory is not modified, because the scheme is lazy - there may be, in practice, another local write from the writing processor, or perhaps a third processor that writes an even fresher version to its local cache, and thus the intermediate write-through would be a waste of resources.

Then, when someone reads a value that is not fresh in main memory, the directory requests, from the relevant processor, a write-back of the latest version, stores it in main RAM, and also forwards it to the reader.

Not shown in the diagram is the internal state of the directory, which must main-

[4]Contrast with being mapped to a particular set of processors, and serving all addresses, which is the case of a cache.

Author	Hardware Size	Burst Rate	Burst Size
Archibald and Baer[5]	$O(1)$	writes only	all
Agarwal *et. al.*[4]	$O(1)$	all	one
Censier and Feautrier[7]	$O(N)$	writes only	readers only
Chaiken *et. al.*[9]	$O(1)$	writes only	readers only

Table 2.1: Performance of Various Directory-Based Cache Coherence Schemes

tain this information on a per-cache-line basis, as each cache line is independent of every other one. Therefore, each cache line must have some auxiliary state associated with it, not including the familiar "valid" and "dirty" bits. I discuss several methods of encoding and manipulating this state in the following section.

2.1.4 Ways of Maintaining Cache State

In the abstract, the state that must be maintained boils down to knowing exactly who has the latest version of a cache line, or an overcautious estimation thereof. In the case of a write-back cache, this line may exist only in a single cache. This state is "exclusive". If the line exists in main RAM, and zero or more caches, then the state is "read"[22]. This is sufficient to completely describe all of the possibilities. In a correct program, an exclusive line will exist in precisely one processor. The presence of multiple caches containing the same dirty address, but with different values, implies that individual words of a cache line are being written[6], and the directory has the responsibility of reassembling the freshest cache line from the components.

Directory-based cache coherence is a solved problem, and thus there are many well-tested ways of maintaining directory entries. Tradeoffs exist between the size of the state space, the frequency of directory-to-processor communication bursts, and the number of processors that must be covered in a burst. I describe several directory state structures. A scheme in which the state space per cache line varies as $O(N)$ for an N-processor system is known as a "full-map" protocol, while one that uses less space is referred to as "limited".

The performance of various directory-based schemes is summarised in Table 2.1. I will now detail their implementations to clarify the values in the table.

The scheme of Archibald and Baer[5] maintains no specific information on *which* cache contains a particular value, only if it is exclusive to the cache, or shared between the cache and main RAM. If an exclusive line is needed by another processor, then a request is broadcast to all processors (the aforementioned overcautious estimation), and *someone* will respond to the request, note that its cache has the dirty bit set, and write back the line. In the case of a line needing invalidation, the broadcast is again made to everyone: in every processor receiving the broadcast, either the line becomes invalid, or it stays so. This scheme requires $O(1)$ bits per cache line, and the constant factor is remarkably small. However, each time a request is made, it must be broadcast to N processors. This request is made only when writing is done: multiple reads are supported transparently.

A small modification, contingent upon careful programming, allows for the state to keep track of whether there is one cache reading a line, or multiple caches. If a clean-to-dirty transition is detected, and there is one reader, then it is assumed that it is indeed the reader that wrote to the address, and a broadcast is avoided. This reduces significantly the frequency of requests.

Another method, by Agarwal *et. al.*[4], allows for at most one cache to contain a particular line. This increases the frequency of communication to be approximately equal to the number of memory requests, not just the number of writes, but both the communication size and the state space per cache line are $O(1)$.

The method of Censier and Feautrier[7] maintains a more fine-grain awareness of exactly which cache contains what. This system is most useful on a processor without broadcast ability, as the invalidate and write-back requests may be sent only to the processors that need to receive them. In this scheme, the cache line state consists of $N+1$ bits. One bit encodes whether the line is exclusive or shared, and the remaining N determine whether each particular cache has the line or not. If a line is dirty, at most one processor is allowed exclusive access. This is the most intuitive scheme, but also the one that requires the most state space: $O(N)$ bits per cache line. The frequency of bursts is proportional to the number of writes, and the size of each burst is proportional to the number of readers.

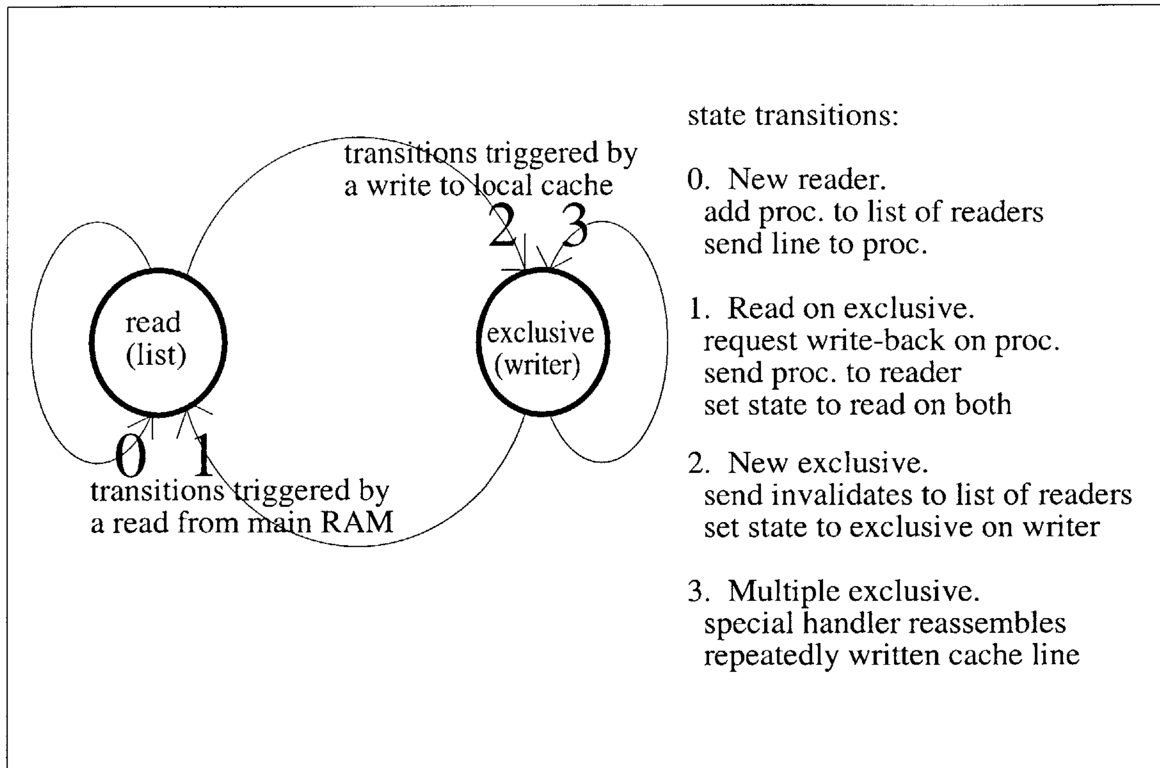


Figure 2-3: The Censier and Feautrier[7] Directory-based Cache Coherence Scheme

Chaiken, Kubiatowicz, and Agarwal[9] show a joint hardware-software directory scheme, in which $O(1)$ readers are tracked in hardware, and any excess is handled in software. This "*Limit*ed directory that is *L*ocally *E*xtended through *S*oftware *S*upport" (*LimitLESS*) scheme has $O(1)$ hardware complexity, and the same activity level (frequency and size of bursts) as the Censier and Feautrier method. However, in the case of needing a software call, the performance is diminished.

At this point, I will focus on the Censier and Feautrier full-map model, as this one is the most intuitive, and also the one I will use, with very slight amounts of modification, in the implementation on Raw.

2.1.5 State Transitions in a Full-Map Directory

The textbook[22] example of directory-based cache coherence is that of Censier and Feautrier. A state diagram of this model is shown in Figure 2-3. The two basic states, "read" and "exclusive", are shown, but each state has associated data. There are two

states[5], and four possible state transitions.

Transition 0 is read-to-read. This occurs when a cache line is being read by zero or more processors, and a new processor wishes to read the line. In this case, the directory forwards the line along from main RAM, and adds the new reader to the list of readers.

Transition 1 is exclusive-to-read. In this case, a processor wishes to read a cache line that is resident not in main RAM, but in the cache of another processor. This writing processor is requested to write back the data, so it is now in its cache and in main RAM. Then, the requesting processor is sent the line. Thus, the new state is read on the two processors.

Transition 2 is read-to-exclusive. This is triggered when a cache line goes dirty due to a processor writing to its local cache. In this case, main RAM and any caches reading it are now invalid, so each reading processor must be sent an invalidate request. The new state is exclusive on the writing processor, as its cache is the only place the freshest data can be found.

Transition 3 is exclusive-to-exclusive. This is more complicated, and may be handled in a variety of ways. Censier and Feautrier do not mention this case explicitly. I will present a solution devised by myself, Ian Bratt and David Wentzlaff[6] (though we are probably not the first to come up with it), in which the cache line is reassembled, word by word. The final state actually ends up as "read".

Now that I have established a state transition protocol, I will discuss the responsibilities of both the processors and the directory in maintaining cache coherence.

2.2 Processors and Directories

This section is a quick overview of the actual implementation of the state transition set described above. It details the separation between processors and directories.

[5]Most discussions add a third state, "none", but I treat it as a subset of "read", as it is simply the "read" state with zero readers.

2.2.1 Directory Responsibilities

In short, the responsibility of the directory is to maintain coherence. This is done through the maintenance of the state of each cache line, and also through the sending of two different types of requests: "write-back" and "invalidate".

It is not necessary to send a write-back request all of the time. Doing so would simulate a write-through cache, but at a possibly very high loss of performance. The exclusive state is preserved for as long as no processor other than the writer attempts a read, and thus no processor knows that there is incoherence between the writing tile and main RAM. Only upon a read request is the incoherence fixed.

The invalidate state must be sent all the time, as the directory does not see cache hits upon local reads. Thus, the directory cannot see when stale data would be read from local cache, and therefore any processor that becomes incoherent due to a read-to-exclusive state transition must be notified of its incoherence immediately.

In short, if a processor becomes incoherent, this must be rectified immediately. If main memory becomes incoherent, the correction can wait.

Each cache line's state is initialised as read with a list of zero tiles upon startup, as no cache has any valid data in it. It is not clear if main RAM has any valid data, but clearly an exclusive state cannot be correct to start with.

2.2.2 Processor Responsibilities

The processor's responsibility is to respond to all directory requests in a timely manner. Since write-back requests are made only when they are certainly needed, any failure to heed a request will result in incoherence. Invalidate requests may be disobeyed at the processor's peril. If it knows it will never attempt to read the data again, it does not matter if the cache is valid or not. The processor's cache stores simply "valid" and "dirty" bits. Upon a dirty bit being set, the processor must notify the directory immediately, as it now contains fresh data, and other locations are stale.

2.3 Private and Shared Memory

Up to this point, the discussion has assumed that all memory is shared. This may not be the case - a particular address may be known, *a priori* to be either shared or private. The directory can also implement private RAM, and also a state that I will call "free-for-all".

I will assume that the status of an address (private, shared, or free-for-all) is a function of the address itself, and the directory is set up ahead of time to check the address and can make, with complete certainty, a decision on the status.

If the memory is shared, handling proceeds as discussed before. If the line is private, then it must have an owner. In the case of a read on a private address by its owner, the line is sent along unequivocally.

If another processor wishes to read this private data, the choice to send it along is an implementation decision. In the case of private RAM, only one cache should, theoretically, contain the cache line. However, if another cache wishes to contain it, it may be acceptable to send along the request in a "caveat emptor" fashion: the stealing processor will not ever be notified if the line goes stale, as the RAM is technically not shared. Alternately, it may be prudent to trip up the violating processor and force it to throw an exception. An incorrect address is probably, but not definitely, a result of a programmer error. The compromise here is one of pedantic correctness versus functionality: if we throw the exception, we may very well be retarding a perfectly good hack.

If the request is a write, and the processor doing the writing is indeed the owner, then the request succeeds. Otherwise, the request must fail. Whether the processor is notified of this failure is another detail of implementation. It is more than likely a programmer error if an incorrect address is generated by the processor, and therefore it should be the programmer's responsibility to check all addresses. Since I cannot imagine a hack that would *increase* performance by making superfluous, completely transparent but time-wasting, writes to addresses it does not own, I advocate throwing the exception on a bad write, just to help with debugging.

The final possible state of a RAM address is "free for all". In this state, the directory does not stand in the way of any reads or writes, and whatever happens, happens. I only mention this possibility because it is the default memory management model of the Raw processor.

At this point, the description of the system is complete. Given cache coherence, a system with local caches behaves identically to the ideal model of the monolithic RAM that is uniformly accessible by all. In the next chapter I will detail the implementation if this cache-coherent scheme on the Raw architecture.

Chapter 3

Implementation on a 16-Tile Raw Processor

The Raw architecture, due to its exposed nature, is an ideal platform for developing software mechanisms that previously could only be done in hardware. The functionality of each processor and each interconnection network can be manipulated on a very fine level, and thus it is logical to implement a directory-based cache coherence scheme as a program running on one or more Raw tiles.

The Raw Handheld board is the first fabricated Raw processor[49]. It is a a 16-tile (4 by 4 square) multiprocessor with further customisable functionality provided by FPGAs around the borders of the tile. In this chapter, I will first describe in more detail the Raw Handheld board. Then, I will detail a mapping of the previously described full-map directory cache coherence scheme onto this architecture.

3.1 Raw Hardware Overview

A high-level schematic of the Raw Handheld board is shown in Figure 3-1. The central element is a grid of 16 Raw processors, which is implemented as a custom ASIC. The word size of Raw is 32 bits, and all networks, on the Raw chip as well as along the perimeter, have this channel width.

Along the perimeter of the Raw chip are six identical Xilinx Virtex II FPGAs[55].

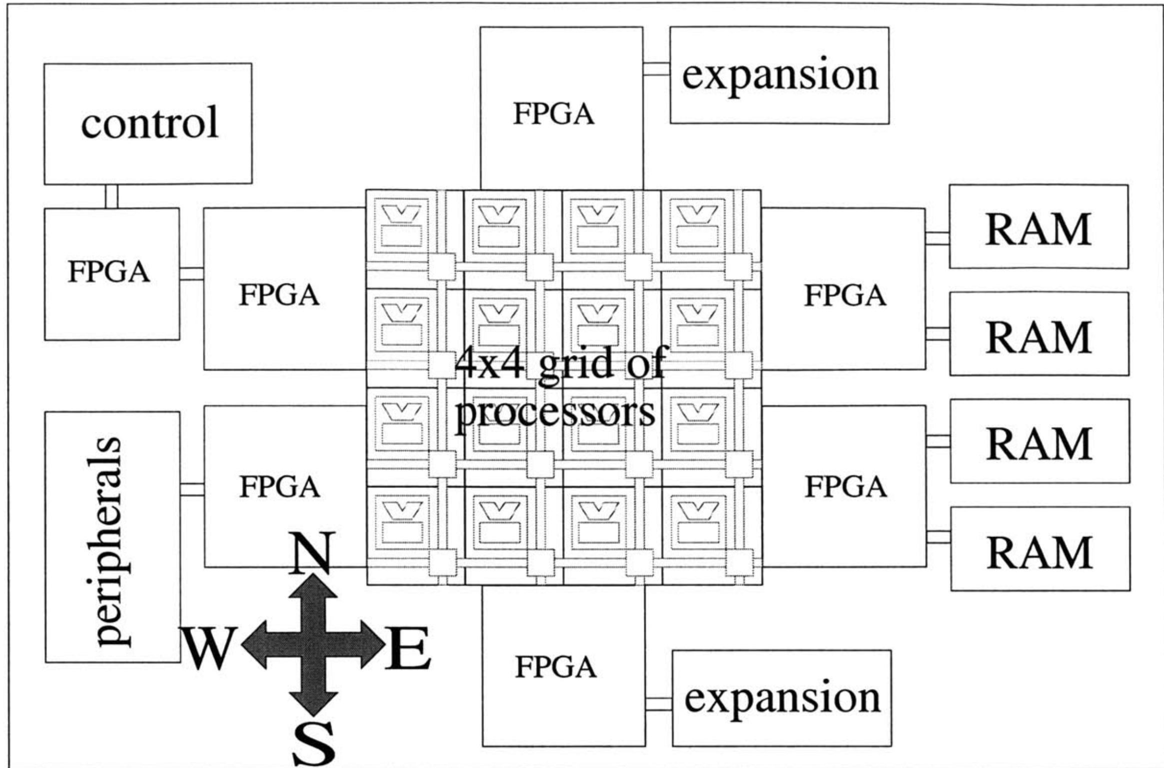


Figure 3-1: High-Level Schematic of the Raw Handheld Board

These FPGAs are connected to a variety of hardware devices that allow for the Raw processor to take on a large set of functionalities. Two, on the north and the south, are connected to various expansion cards to allow, for example, streaming data from A/D converters. The FPGA at upper left (west) is connected to another, smaller FPGA that allows for run-time control of certain fundamentals, such as clock speeds. The FPGA at lower left is connected to a set of peripheral interfaces, including a keyboard interface and a USB interface, through which Raw programs can be booted.

Finally, the two FPGAs along the east side form an interface between the Raw chip and four banks of DRAM. Each of these FPGAs has completely custom firmware that may be updated on the fly. We will focus only on the two memory controller FPGAs - the others are not part of the shared memory interface, and can be left for the user to program in any way. I will now describe the Raw chip itself.

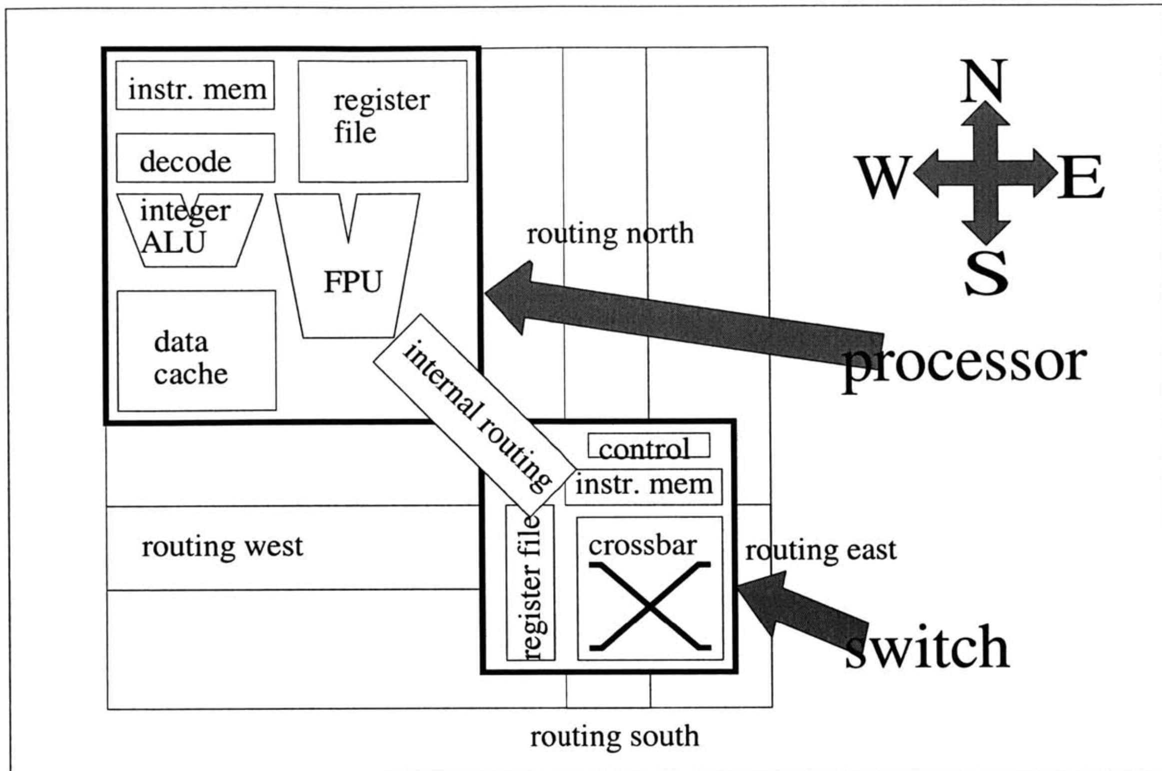


Figure 3-2: A Single Raw Tile

3.1.1 The 16-Tile Raw Chip

Each identical Raw tile consists of a processing subcomponent (the "processor") and a routing subcomponent (the "switch")[48]. The switches form four independent interconnection networks, which also extend off the chip. A single tile is shown in Figure 3-2. The processor is very similar to a MIPS R4000 RISC processor. The switch contains various routing controllers. First, I will detail the functionality of the switch and the interconnects, and then that of the processor.

Routing Networks

The four interconnection networks come in two basic flavours. Two of the networks are "dynamic": standard dimension-ordered packet routing networks[2]. Of these, one (the "general dynamic network", or GDN) is left entirely under user control. The other, the "memory dynamic network" or MDN, may also be accessed by the user, but is also accessed by the hardware. Upon a cache miss, the MDN is used to

communicate with the DRAM banks. Therefore, the default behaviour of the FPGA memory controllers is to communicate with the MDN.

A dynamic message consists of a header word and zero or more data words. The header contains fields for the source and destination, the number of upcoming data bits, a final routing field that allows a packet to be routed off the chip (once the target tile is reached, the final switch routes the word either into its corresponding processor, or one hop over in the final direction), and a "user" field. The user field has a variety of purposes. In this thesis, its only purpose is to identify interrupts.

The dynamic networks are accessed directly by the processor through a register-mapped interface. A read from a special purpose register pulls a word off the network, and a write to one pushes a word onto the network. It is the processor's responsibility to encode headers correctly and write and read the registers. Once the data is on the network, it is routed in a user-transparent manner by the switches. When the message reaches its destination, its header is swallowed by the final switch, and only data may be read by the processor. In the case of an off-chip routing, the header is preserved.

Dynamic routing is illustrated in Figure 3-3. The source processor generates a header and writes it, and the data, to the appropriate register. The switches route the message vertically first, then horizontally, and then the final routing can choose either to send the message to the processor, or off the edge. In this example, there are two valid final routing choices: east or to the processor. A final route that does not leave the tile or go to the processor is invalid.

The other two networks are "static". They are scalar operand networks[51], meaning that each packet is one word long. The processor accesses the scalar operand networks by writing to or reading from special registers, similarly to how the dynamic networks are accessed. There are no headers for the static network, and routing is determined by a program that executes on the switch. The switch has its own instruction memory, fetch and execute unit, a small register file, and the ability to perform compares and branches, and therefore is a small processor in its own regard. Each line of switch code is a VLIW instruction, consisting of an operand and a series

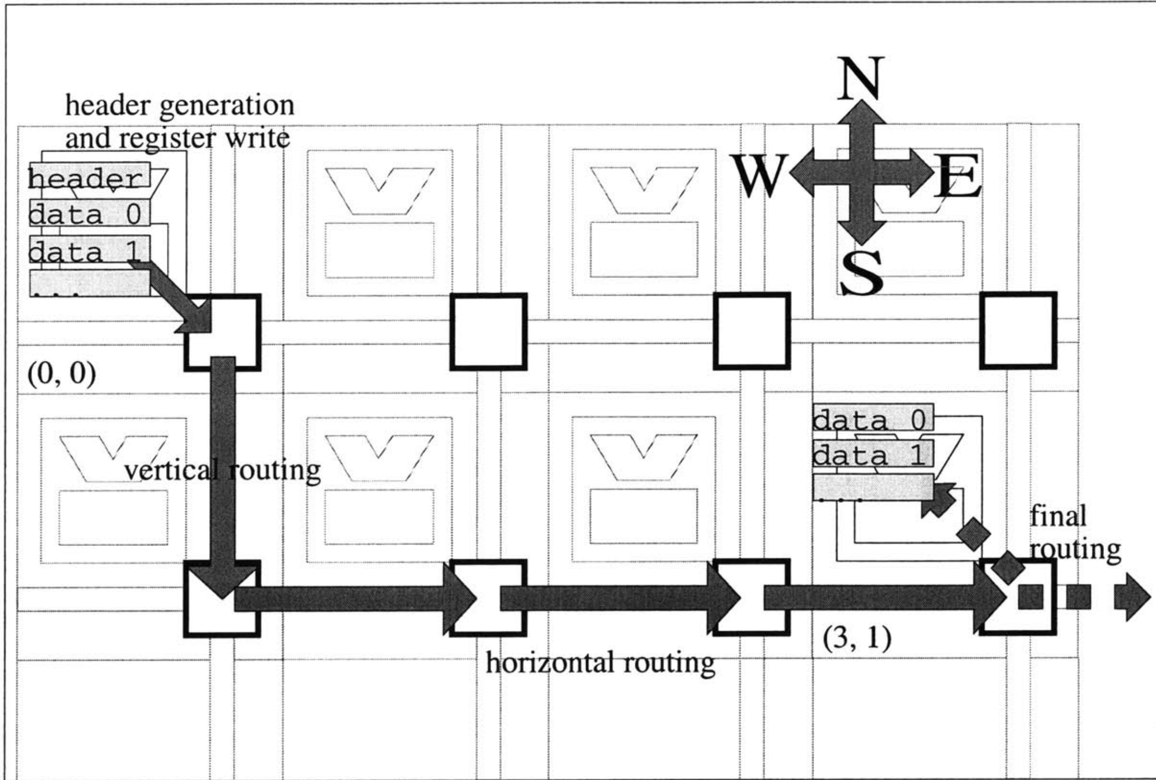


Figure 3-3: Dynamic Routing of a Message from Between Processors

of routes. The operand determines the control flow of the switch program, and the routes are performed on incoming data. For example, a valid switch instruction is:

```
j .  route $csto->$cWo, $cEi->$csti.
```

This instruction executes repeatedly, due to the unconditional jump to itself. Then, a word is routed from the processor (csto) out the west port (cWo), and another word routed in from the east (cEi) to the processor (csti). Not all routes are possible - a full protocol is described in [48].

All routes, including register reads and writes in the processor are blocking. If there is no space in a buffer to be written to, or there is no data in a buffer to be read from, then the instruction stalls. On the switch, *all* routes to be executed in parallel must be possible before the routing is done the instruction stream proceeds. An example is shown in Figure 3-4. A single data word is routed to three different targets via the static network: two destination processors, and also off the east edge. Note the static network's multicast capability: a single input is routed to multiple

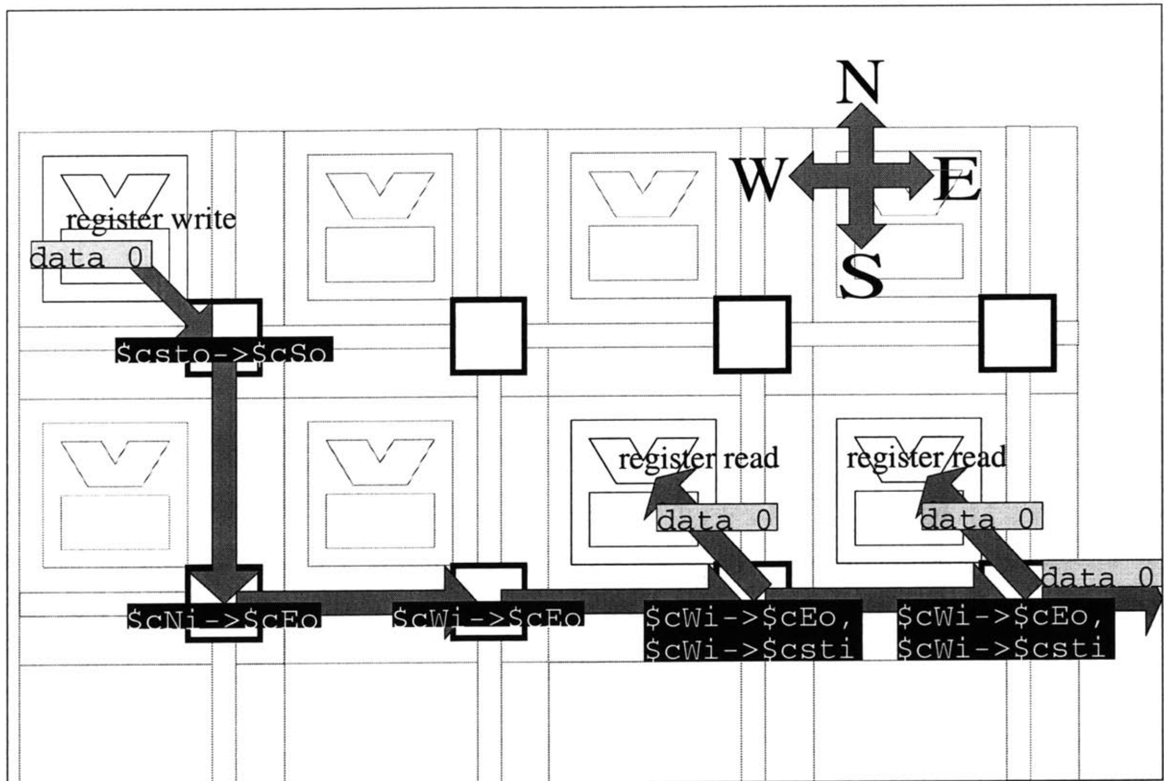


Figure 3-4: Routing on the Static Network

outputs in parallel.

There are two independent static networks. In this thesis, only one is utilised, thus I will make reference to "the" static network.

Raw Processor Functionality

As mentioned before, each Raw tile contains a fully functional RISC processor, roughly similar to the MIPS R4000. Its exact functionality is detailed in [48]. I will only describe two functions relevant to this thesis: interrupts and cache misses.

Cache misses are actually handled in hardware on the Raw Handheld board. A future version of Raw will include a software handling mode[48], in which a cache miss throws an exception, allowing a routine to be run on the processor. However, the current functionality is as shown in Figure 3-5. On either a read or a write miss, the relevant cache line must first be read in. The missing instruction (a load or a store) stalls, and a MDN message is generated. This message goes to the FPGAs

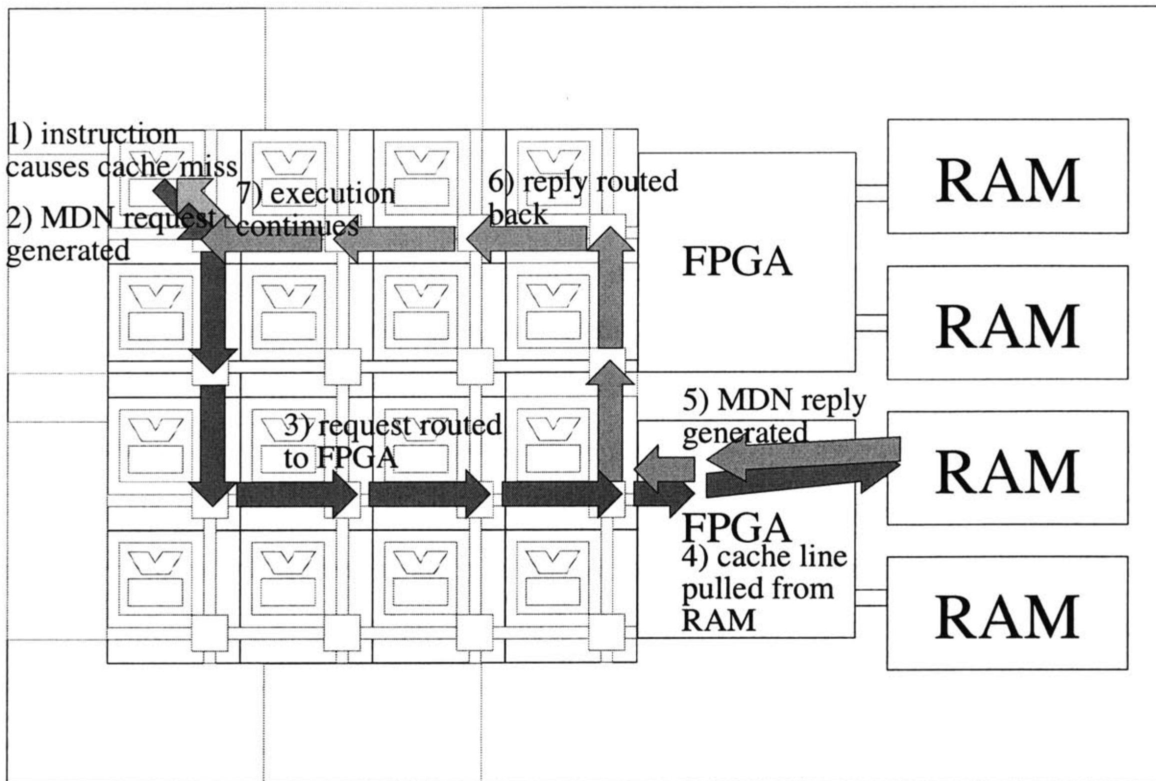


Figure 3-5: Servicing a Cache Miss on a Load

that sit on Raw's eastern edge, which fetch the relevant cache line. Each cache line is eight words long, so the processor stalls until it receives eight words via the MDN. It is important to note that it cannot differentiate between an MDN message from the FPGA and from another source. In the case of a store, the newly read line is modified in local cache immediately.

In the case of a write causing a line to be evicted from the cache, another MDN message is generated, with the address and data to be stored. This message also goes to the FPGA.

The other important functionality of the Raw processor is its ability to take interrupts. There are two levels of interrupts: user and system. System interrupts have a higher priority, in that a user interrupt handler can be interrupted via a system interrupt. The system interrupt handler cannot be interrupted at all.

A single interrupt of each of the seven kinds may be buffered. Outstanding interrupts are stored in a special-purpose register, and every time an interrupt is received,

a bit in the vector is ORed with 1. If the bit was not set, then the interrupt is recorded. If the bit was already set by a previous interrupt, the new interrupt is effectively ignored.

When an interrupt is handled, the corresponding bit is cleared. Interrupts are handled as quickly as possible. However, if the processor is stalling, due to waiting for a routing-network message (a special purpose register read), or an interrupt handler of equal or higher priority is running, then the new handler will not be called. The two types of interrupt we will be looking at are Interrupt 3 ("External", system level) and Interrupt 6 ("Event Counter", user level).

The external interrupt is generated by sending the processor to be interrupted a message of length 0 on the MDN, with the User bits set to "1111". Since headers are not routed to the receiving processor, the message is effectively extinguished once its task is complete. When the interrupt handler is invoked, the processor (which does *not* know who sent the interrupt) will likely communicate with an interrupt server, which is located either on another tile or on an FPGA. This communication, as well as the design of the interrupt handler, is left up to the user. I will detail the interrupt handlers for the shared memory system in a later section.

Event Counters

The other type of interrupt is generated by the triggering of an event counter. An event counter is a particular hardware module that looks for a specific type of event, and if it notes one, it decrements a special purpose register. When the register's value hits zero, the event counter fires off an interrupt. The user may write to these special-purpose registers. This allows the event counter to be configured to trigger after a specific number of events.

There are eighteen event counters on every Raw tile. We will be concerned with precisely one: it counts every time a cache line goes from clean to dirty. The use of this event counter to detect cache line dirtyings was originally discussed in [44] by David Signoff and Michael Taylor.

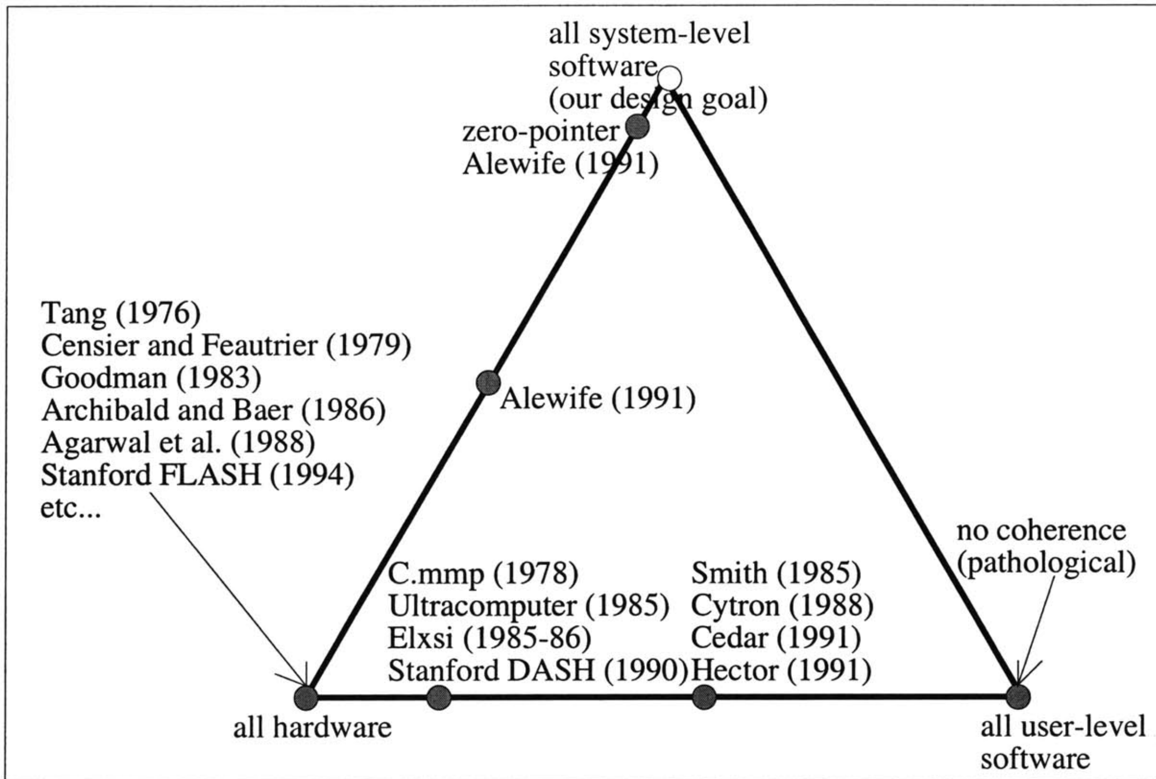


Figure 3-6: Various Cache Coherence Schemes, Revisited

3.2 Design Objectives

The objective of this system is to be implemented in software as much as possible, and to be as transparent to the user as much as possible. The combination of these two is a viable research goal.

As discussed in a previous chapter, most cache coherence solutions, with the exception of the Alewife system[9, 8], have been a combination of hardware and user-level software. We return to the figure showing the distribution of functionality of various systems. The design goal is shown in Figure 3-6. I add an extra data point at the ideal "all system-level software".

3.3 Implementation Redux

Before diving into the low-level details, I quickly overview how the directory-based shared memory system is implemented in Raw. The physical layout is shown in

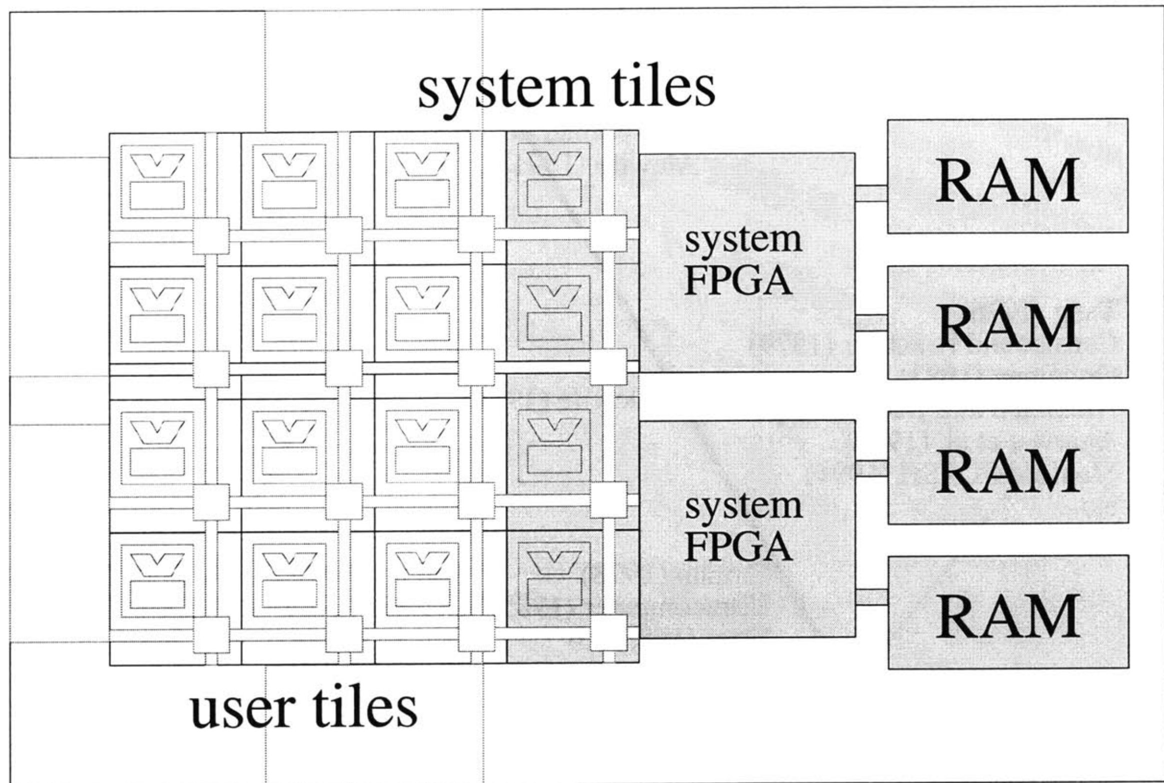


Figure 3-7: Dividing the Raw Handheld Board into Components

Figure 3-7. User tiles run the application, and act as the "processors". System tiles act as the "directories". The combination of RAM and FPGAs acts as "memory".

I will demonstrate how four basic shared memory primitives are executed on Raw. They are the four state transitions shown in Figure 2-3. The first is the reading of a word that is in main RAM. The second is a store of a word to local cache, and the resulting state change in other processors. The third is the more complicated reading of a word that is exclusive to a processor. The fourth is the most complicated of all: when two tiles attempt exclusive access to a single line.

3.3.1 Read of Shared Data from Main Memory

The simplest shared memory function is a read of main memory. This is shown in Figure 3-8. The tile in the upper left corner reads the data value "A", which is in main memory, and also being read by several other processors. The request starts out exactly as in the non-shared version shown in Figure 3-5, with an MDN message

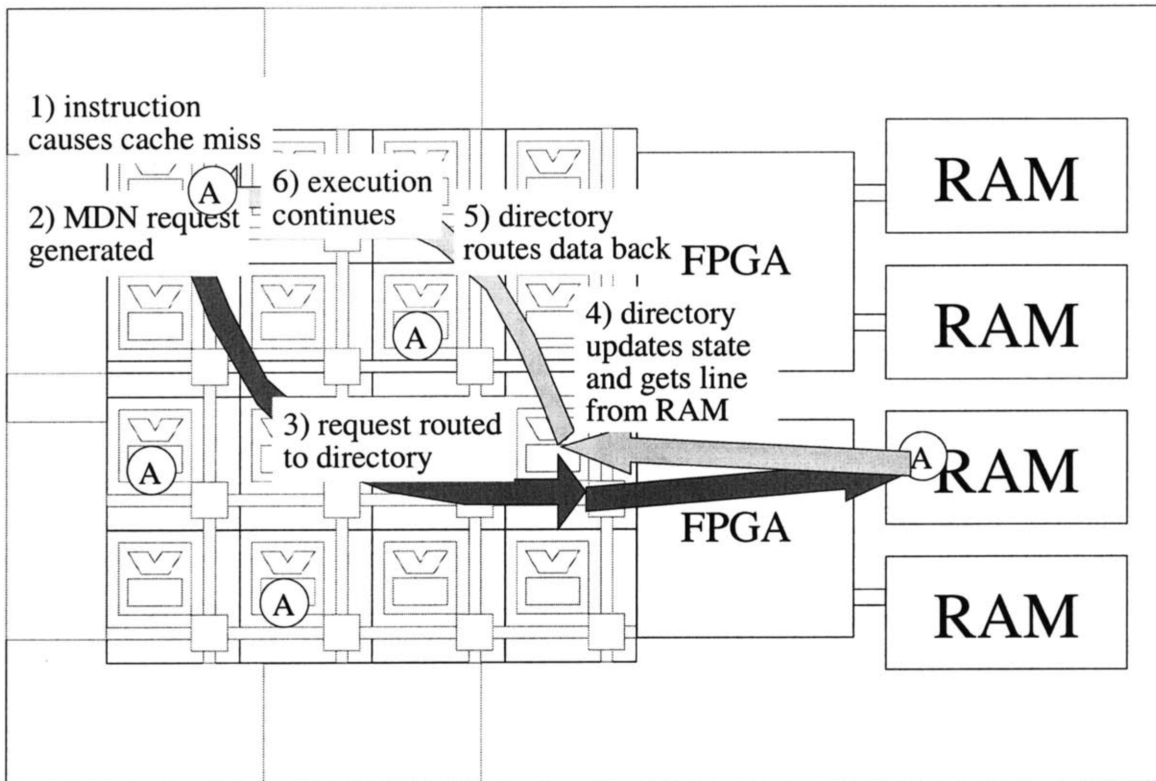


Figure 3-8: Read of Shared Data from Main Memory

getting sent to the directory. This message contains the address requested, and the requesting tile. The directory can therefore update the state. Since the data was in main RAM, the state remains "read", but now there is one additional reader. The directory then fetches the line from main RAM and sends to the new reader.

3.3.2 Store of a Word to Local Cache

In Figure 3-9, the processor in the upper left writes a new value, "B", into the cache line it had loaded in the previous step. Now, the other tiles have stale cache data. Upon writing the new word to cache, the writing processor sends a message to the directory, and stalls until this message is acknowledged.

The directory then tells the other reading processors to invalidate the line in question. When all of these invalidations are sent and acknowledged, the state is set to exclusive. Main RAM is thus marked invalid, and the processor doing the writing is sent an acknowledgement and allowed to proceed.

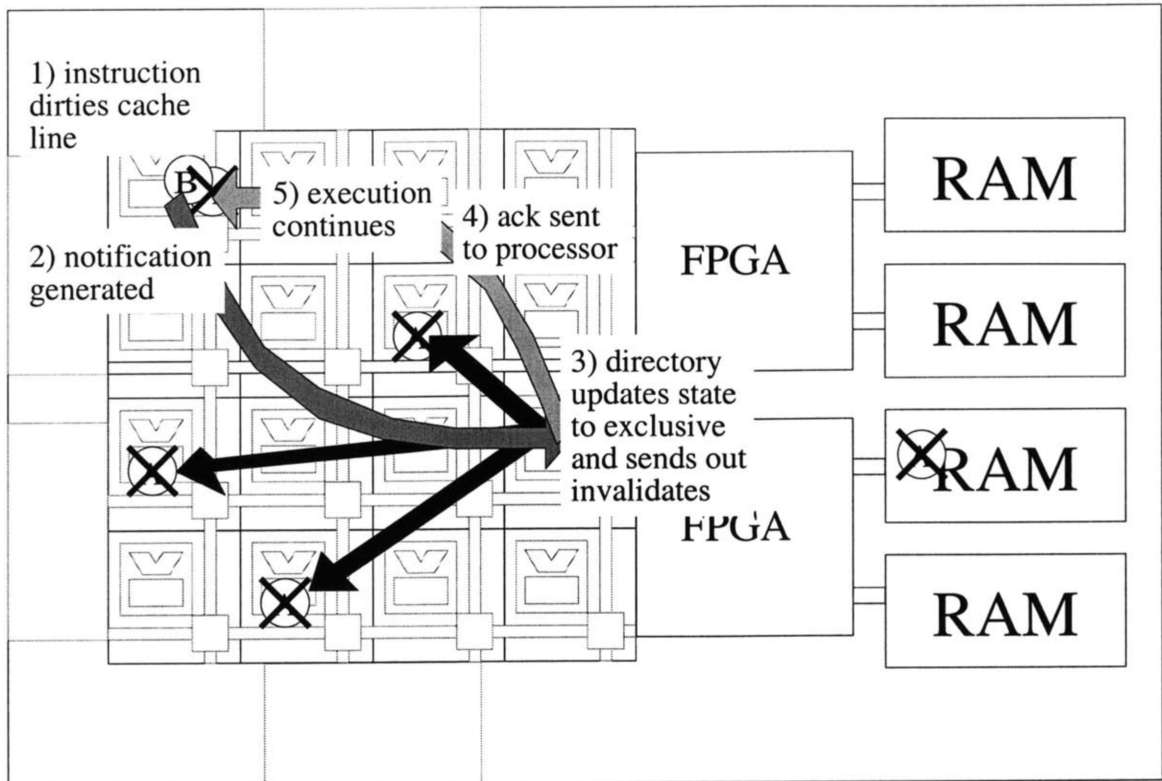


Figure 3-9: Store of a Word to Local Cache

3.3.3 Read of an Exclusive Word

The third important process is the extraction of a word that is local to one cache, and sharing it with other readers. This is shown in Figure 3-10. First, a cache miss occurs in the reading tile, which is in the lower left. This results in the hardware sending a message to the directory, requesting a load of a cache line. The directory knows that the line is exclusive to the upper left tile, so it asks for a writeback, which results in the line being sent to it. The directory sets the new state to "read" on the two tiles, since the line is fresh in both of their caches and also in the RAM. It also writes back the line to main RAM.

3.3.4 Multiple Writes of Exclusive Lines

The final case that I discuss is the most complex. This occurs when a line that is exclusive to one tile is written by another tile, giving rise to the contradiction of multiple processors having "exclusive" control of a line.

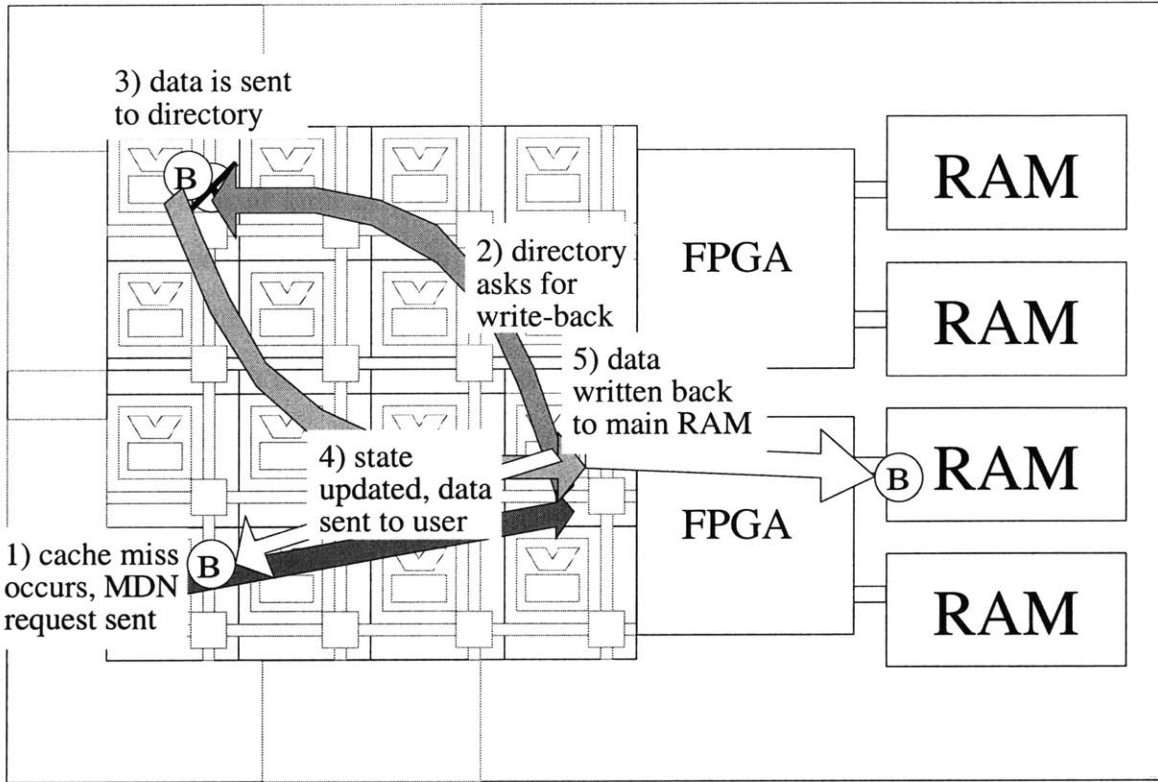


Figure 3-10: Read of an Exclusive Word by Another Tile

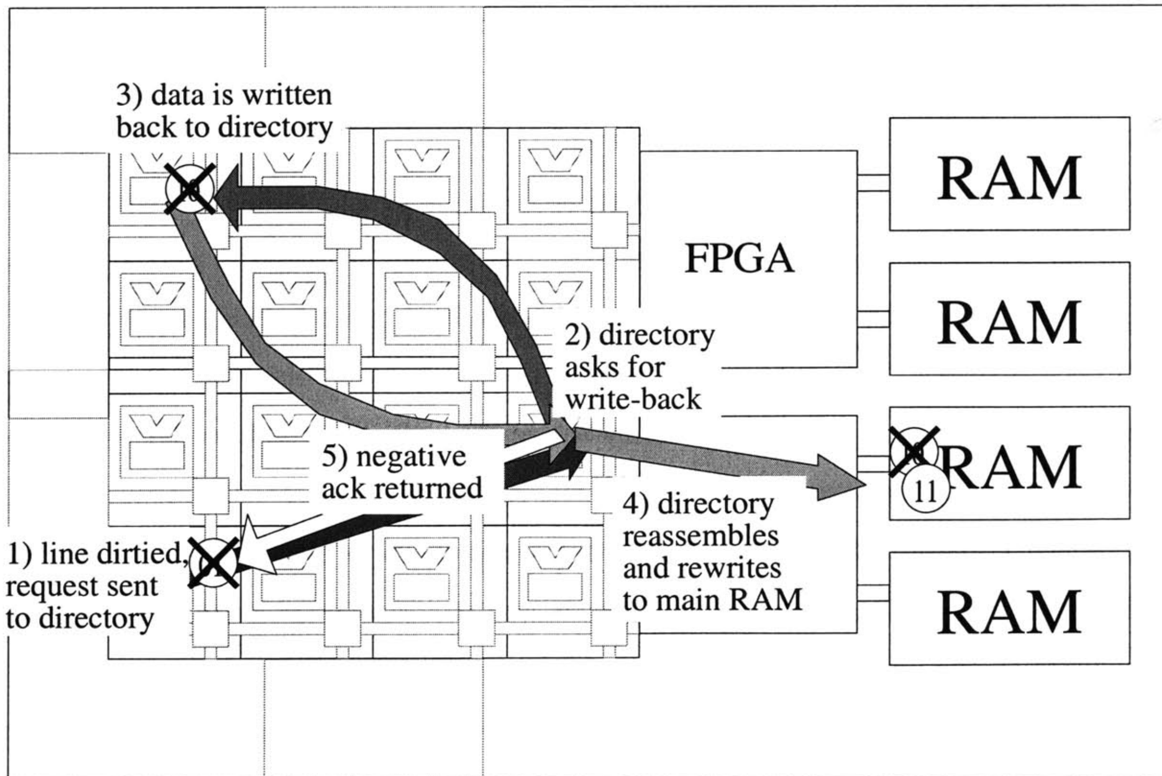


Figure 3-11: Multiple Writes of Exclusive Lines

This scenario may or may not be a write-after-write hazard, given that a single line has multiple words. One processor writes one word, another writes another word, and the freshest cache line is actually a combination of what exists on either processor.

The implementation described here was developed in final form by myself, but I must acknowledge a discussion with Ian Bratt, who discovered the problem, and David Wentzlaff, who came up with the idea of "patching" as a solution to it [6].

We start with a cache line in the "read" state, with both the upper left and lower left tiles reading it. Then, both write a particular word. The upper tile writes a "1" into the first slot on a cache line, and the lower into the second. This is shown in Figure 3-11.

If the time between writes is sufficiently long, then the second writer gets an invalidate from the directory and will read a fresh line before writing it. However, assume that the writes occur "simultaneously", meaning so close together that the directory gets the notifications late.

The directory receives the notification from the first writer, setting the line to exclusive. Then, it receives the notification from the second writer. This notification includes the precise address being written, and the data word. Therefore, the directory keeps this data word as a "patch".

The directory then sends a write-back-and-invalidate request to the first tile, and the line is written back to main memory. At this point, the directory patches the line with the information from the second tile (so the final result is "11") and rewrites it to main memory. The state is then set to "read". At this point, the processor that wrote second is sent a negative-acknowledgement, and it invalidates its cache line. Neither tile has the latest version of the line in cache.

I will prove in a later section that this scenario works even in the case of more than two tiles fighting over a single line, as well as in the presence of incoming read requests on the line being assembled.

Those are the basic possibilities that may take place in the directory-based cache coherence model. Now that I have established how the system works as a whole, I will detail the interaction of all of the components in more detail. First, I describe the

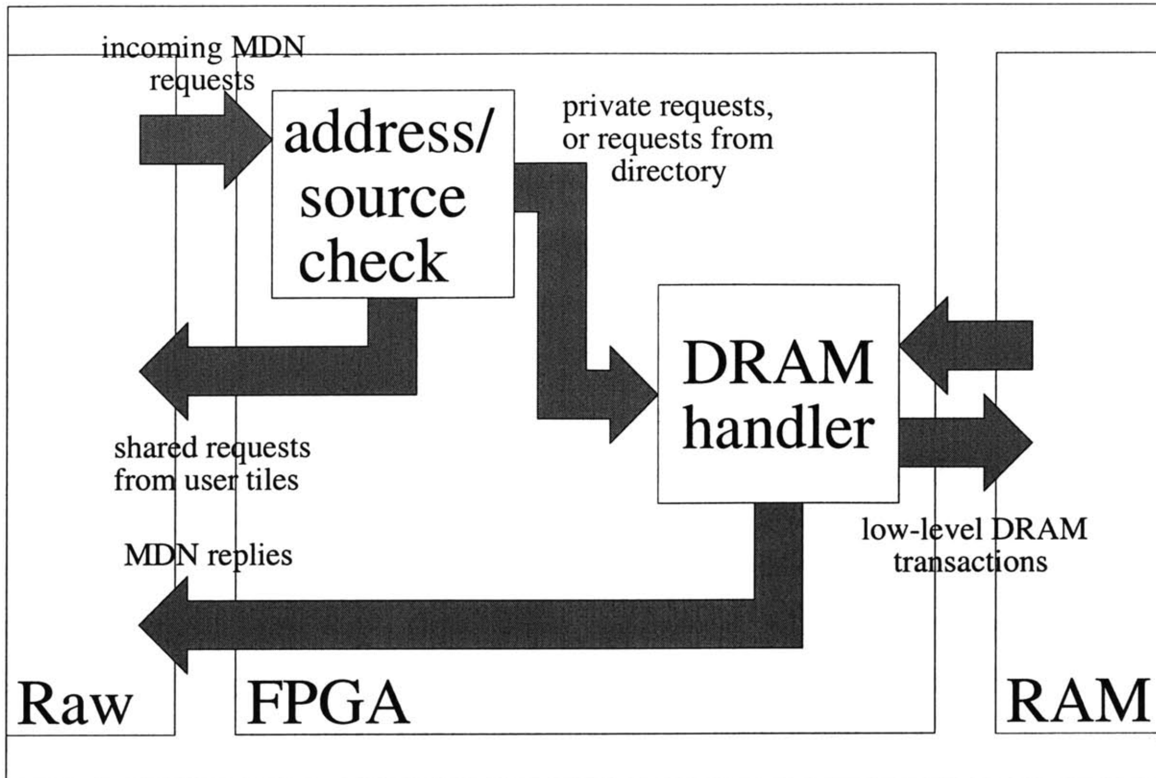


Figure 3-12: Functionality of the FPGA

responsibilities of the FPGA. Then, I review the communications between directories and users. Finally, I go through the low-level details of implementing both user and system tile functions.

3.4 Overview of FPGA Functionality

The FPGA is responsible for maintaining the boundary between the Raw processor and the DRAM. This consists of interactions on two separate boundaries. There are two FPGAs, each serving two DRAM cards. The firmware in each FPGA is identical, and in fact is replicated twice, allowing for four separate memory controllers operating in parallel. A single memory controller is shown in Figure 3-12.

On the RAM side, the FPGA takes care of the very-low-level details of interacting with a DRAM card. This includes things like clock boundaries, meeting setup and hold times, and the like, and slightly more abstract functions like reading and writing

the correct number of words per cache request (eight, in our case). This interface has already been designed[43].

The other end of the FPGA interacts with Raw. Currently, the FPGA operates in a "free-for-all" mode, in that every store request is dutifully written to DRAM, and every load request is dutifully read. These requests are read off of, and written to, the MDN. Since MDN routers are abstracted away from the user, this free-for-all scheme does not allow for a directory, implemented in software on a system tile, to stand directly between the processors and the RAM.

Since this hardware restriction cannot be changed, a workaround is devised. The message goes to the FPGA, which then *immediately* (after checking to see if the address is shared) bounces it to the directory[1] via the static network. The directory then may choose to forward the request via the MDN. This way, the directory is logically between the memory and the processors, as in the abstract model.

Private memory requests, or shared requests coming from the directory, are sent to the DRAM handler, which takes care of the transaction from main RAM. The replies (in the case of loads) are all sent via the MDN back to their requestor.

We can thus sum up the FPGA's responsibilities as twofold. First, it takes care of the ugly details of DRAM management. Second, it filters requests and sends shared requests to the directory for further processing. Therefore, in the context of shared memory, it acts simply as a buffer between a DRAM element and the directory responsible for it.

Next, I describe in more detail the specifics of communication between a user tile and a directory.

[1] I use the term "the directory", even though there are four. For each address, the directory is uniquely identified, so therefore we are allowed to discuss it as though it were physically unique.

3.5 Communication between Users and Directories

The purpose of this section is to give a slightly lower-level overview of the communications that go on between the various parts of the system. The previous discussion did not mention any specifics of how certain messages are passed between these various components. In this section, I discuss the two types of interrupts that arise on the user tile in response to stimuli, motivate the necessity for interrupt controllers as an intermediate between users and directories, and then describe the protocols by which the various state transitions and update notifications are made.

3.5.1 Interrupts on the User Tile

Under most circumstances, the user tile is running application code. Therefore, in order for the system to communicate with it, the tile needs to run one of two interrupt handlers. There are two interrupts that we will worry about. Interrupt 6 ("Event Counter") is triggered upon a cache line going from clean to dirty. Interrupt 3 ("External") is triggered by a special interrupt message on the MDN from another tile. When a user tile is put into the Interrupt 3 handler, it triggers a series of communications by which requests from directories are resolved.

3.5.2 Interrupt Controllers

Interrupt controllers are necessary because of a quirk in Raw's design. The External interrupt does not convey any information about its source. Therefore, an interrupted tile must, in order to receive further information, send a message to a specific interrupt controller. The assignment of interrupt controllers to user tiles is done *a priori*, as shown in Figure 3-13. Each of the four system tiles has both directory and interrupt controller responsibilities. The directory aspect of each tile serves a particular subset of addresses, and each directory can communicate with each user tile. However, each interrupt controller serves a particular subset of user tiles, and communicates only

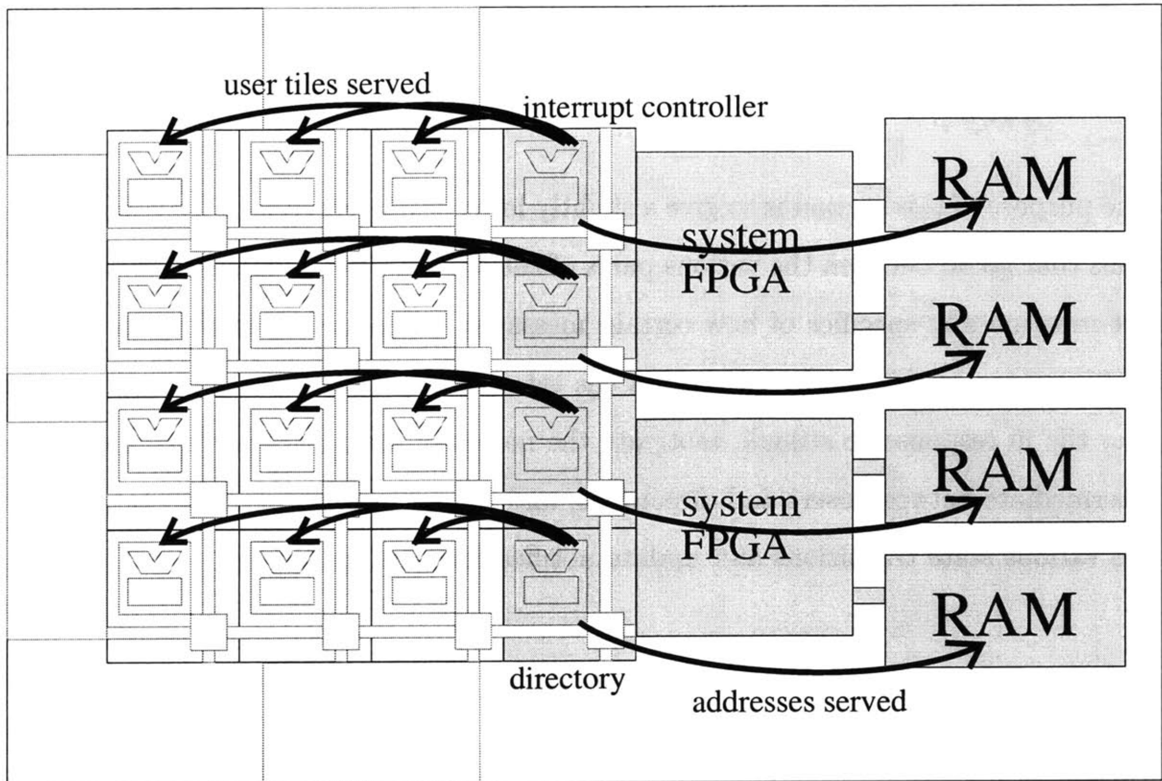


Figure 3-13: Directory and Interrupt Controller Responsibilities of System Tiles

with them. In this implementation, every interrupt controller serves the user tiles in the same row.

3.5.3 Communication from a Directory to a User

The purpose of the interrupt controller is to completely facilitate a communication between a user tile and a directory. It does this by keeping track of when it believes a user tile is not in an interrupt handler, and thus can be interrupted, and in the case of a reply not coming back in a timely manner, taking further measures.

This communication sequence is shown in Figure 3-14. It is started when a directory needs to communicate with a user. All of the communications are done via the GDN, except for the actual Interrupt 3, which is sent via the MDN. Furthermore, if the directory and the interrupt controller are physically on the same processor, then communication between them is done locally[2].

[2]This is not just a matter of saving time, actually. It is very easy to deadlock the GDN by sending

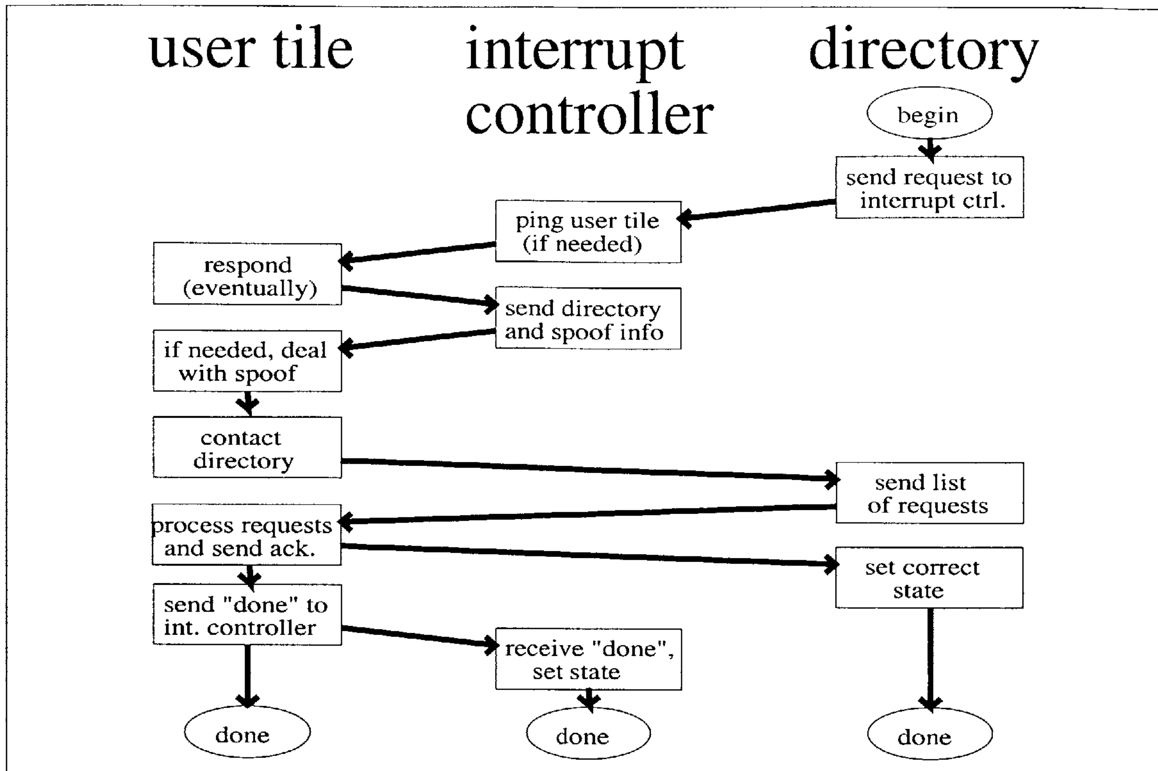


Figure 3-14: Communication from Directory to User Tile via Interrupt Controller

The directory sends a request to the user tile's interrupt controller, and the interrupt controller then starts a process by which it interrupts the user tile. The user is only pinged once, even if multiple directories send a request to its interrupt controller. When the user tile responds, the list of all directories with requests is sent. Some auxiliary information may be sent as well, if the interrupt controller had to take certain deadlock-resolution measures (these will be discussed in a later section) to coax the user into an interrupt handler.

At this point, the user tile deals with the ramifications of deadlock, and then contacts the directories, one at a time. Each directory sends along a list of addresses and requests (invalidates and/or flushes), and the user tile handles these.

When the user tile is done servicing directories, it sends the directory tile a final "done", acknowledging all of the transactions, and then sends the interrupt controller one final message, by which the interrupt controller knows that the user tile is no

a message to one's self!

longer in an interrupt handler, and therefore may be interrupted again.

3.5.4 Communication from a User to a Directory

In contrast with the previous chain of events, a communication initiated by a user tile is very straightforward. The user tile needs to communicate with the directory under two circumstances: whenever it has a cache miss, or whenever it dirties a cache line.

In the case of a cache miss, Raw's low-level hardware sends the data directly to the FPGA, and the FPGA then bounces it to the directory via the static network. When a cache line goes dirty, the user sends this news along to the directory in control of the address. Then, it waits for an acknowledgement from the directory. When it receives it, it either continues normally, or (in the case of a negative acknowledgement) invalidates the cache line it has just written. This occurs when the cache line is exclusive to another tile, in which case advanced measures are being taken by the directory to restore consistency. After receiving this acknowledgement, the user exits the interrupt controller.

Next, I detail the inner workings of the system tile, which contains the directory and the interrupt controller.

3.6 Low-Level Analysis of System Tile Functionality

This section deals with the low-level implementation of the various system tile functions, both as the directory and as the interrupt controller.

System tiles do not run user-level code. Furthermore, they do not get interrupted. Any communication from the FPGA to the directory is done via the static network, and any communication from the user to the system tile, or from system tile to system tile, is done via the GDN and the MDN. The system tile monitors incoming activity on all three networks and launches handling routines in response to these events, updating state and notifying user tiles of looming incoherence.

The system tile's code is presented in Appendix A. In this section, I describe the full functionality of this code. First, I overview the event-driven multitasking nature of the system tile, including the ramifications on sequential consistency. Then, I detail the two main tasks of the system: directory management and interrupt issue. Next, I discuss the possibility of deadlock, and how the system resolves it. Finally, I make note of some methods used to improve performance.

3.6.1 Multitasking

The main design goal of the system tile is "timeliness". The system tile may be communicating with multiple user tiles at the same time, and it is necessary that all requests are processed in a timely manner, and no particular thing is allowed to pile up. In order to achieve this, multitasking is employed, where long-latency communications are broken up into small transactions. Each burst is quickly handled and then context is switched.

This allows for processing of asynchronous requests in the order that they arrive in. If a system tile were waiting for a particular message in a sequential communication protocol, then it would have to buffer other messages arriving from other tiles, which would require large amounts of resources, and also be contrary to the timeliness goal.

In this section, I first describe the main loop of the system, including points at which context is switched. Then, I describe how these context switches result in state transactions no longer being atomic, and show the solution to the resulting sequential consistency problems. This description entails the addition of several new states in addition to "read" and "write", and also the use of an auxiliary structure to handle intermediate state: the pending address table.

The Main System Loop

The main loop of the system tile is shown in Figure 3-15. Data arriving on either the static network or MDN may imply an incoherent state The static network handler and MDN handler detects this, and notes the relevant flushes and invalidates that

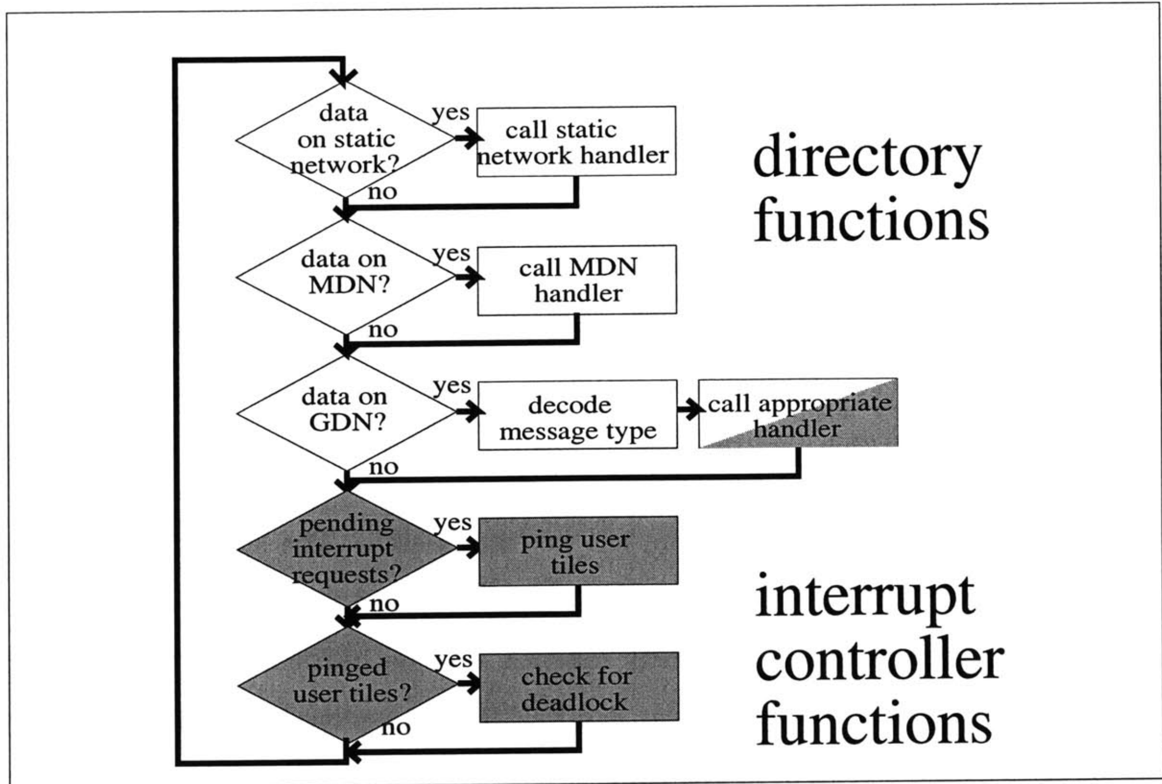


Figure 3-15: The Main Loop of a System Tile

must be done. Then, a communication with an interrupt controller is set up.

A message on the GDN implies one of several stages of a communication between a user tile and a directory, and may be aimed at either an interrupt controller or a directory. Each GDN message is uniquely tagged and the appropriate routine is called, which advances the state of the communication by sending a further message if needed, and then context swapping to do other things while the reply arrives. This is shown in Figure 3-16.

The last two functions are part of the interrupt controller's code. If a user tile needs to be interrupted, then the controller takes care of these requests. It is especially important to wait for a user tile to respond in a non-blocking fashion because there may be a deadlock that needs to be resolved, which can only be distinguished from an ordinary stall condition after a long period of time.

Therefore, the system tile can be seen as a pipeline that handles requests in small amounts. No subroutine on the system tile is blocking - while the system tile waits

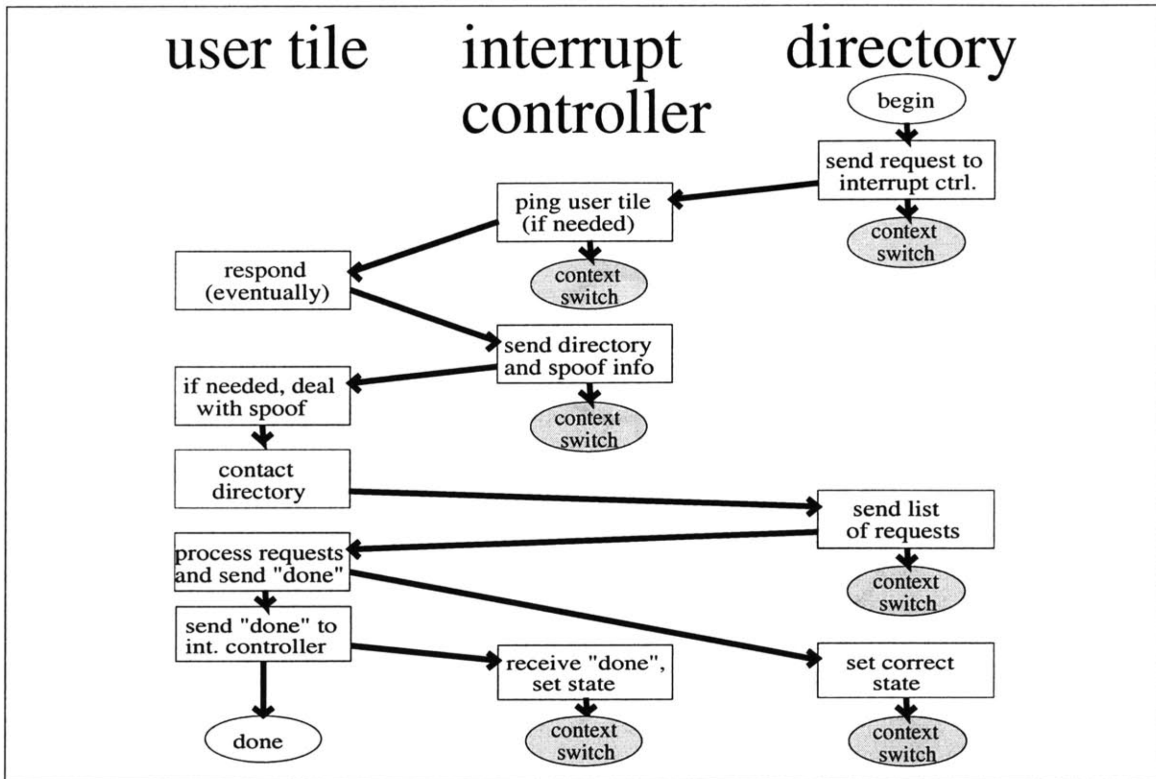


Figure 3-16: Context Switches in the User-Directory Communication Sequence

for a particular segment in a communication, it is busy processing other segments of other communications.

Intermediate States

One of the consequences of context switching is the presence of intermediate states, as transitions are no longer atomic. While the directory waits for a user tile to respond to a particular request, other requests will arrive asynchronously, and need to be handled in a proper manner. Three new states are shown in Figure 3-17: "read-lock", "read lock bypass" and "exclusive-pending".

One new state is "exclusive-pending". This extra state notes a write-back request has been sent from a directory on an exclusive line, so that multiple requests are not sent out.

The state of a line is set to read lock when the state is read, but a tile wishes to make it exclusive. Before this exclusive access is allowed to happen, all reading tiles

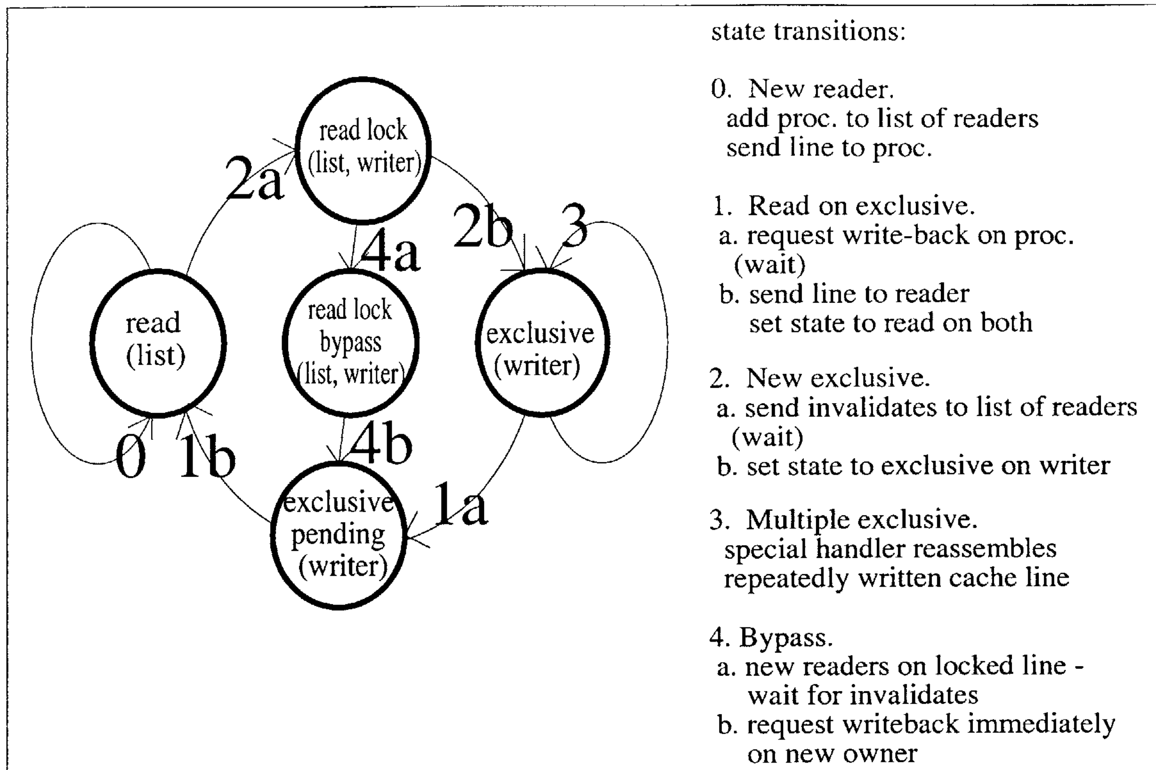


Figure 3-17: Intermediate States to deal with Non-Atomic Transactions

must be invalidated. To fail to do so is a violation of sequential consistency[30, 3]. If all goes well, the writing tile is sent an acknowledgement, and the state is set to exclusive on it. If, however, a request arrives that would require the to-be-exclusive tile to write back the line, the state instead goes to "read lock bypass", in which we note that once the invalidates of the old readers complete, the state should no longer be exclusive, but rather "exclusive-pending", and the writing tile should immediately write back the line it gained exclusive access to.

There are four possible requests that arrive from user tiles: write-back (a cache line is written back to main RAM), acknowledge invalidate (a line is invalidated on request), new writer (a cache line goes dirty, implying the user wishes to gain exclusive write access), and new reader (a read request on the line). The first two are mainly expected, and the last two are not.

The responses to these requests, given a particular state, are summarised in Table 3.1. If a stimulus is not listed for a particular state, then it is not possible.

State	Stimulus	Response	New State
read	new reader	add to list, send the line	read
read	new writer	send invalidates to readers	read lock
read lock	new reader	note request in pending address table	read lock bypass
read lock	new writer	send writer an invalidate, note patch in pending address table	read lock bypass
read lock	last inv. ack.	send writer an acknowledgement	exclusive
read lock bypass	new reader	note request in pending address table	read lock bypass
read lock bypass	new writer	send writer an invalidate, note patch in pending address table	read lock bypass
read lock bypass	last inv. ack.	send writer a flush-and-invalidate request	exclusive pending
exclusive	new reader	note request in pending address table, ask for write-back from owner	exclusive pending
exclusive	new writer	send invalidate, note patch, ask for write-back from owner	exclusive pending
exclusive	write back	spontaneous writeback (eviction?) - set read on writer	read
exclusive pending	new reader	note request in pending address table	exclusive pending
exclusive pending	new writer	send invalidate, note patch	exclusive pending
exclusive pending	write back	patch line in main RAM, send to readers	read

Table 3.1: State Transitions due to Asynchronous Inputs

It is important to note the following trends in the state transitions:

- It is not possible to write back a line that is in a read state. If a cache line is in read, then no tile has it exclusively, so therefore it is not dirty. Any acknowledgement of a cache line going to dirty, therefore, must be met with a subsequent state transition to an exclusive state.
- The first writer to a line is allowed to gain exclusive access. Once this process is started (the state is anything but "read"), subsequent writers are not allowed to have the line in local cache. Instead, the modified words are recorded by the directory.
- There is a fence point between old and new readers. All old readers must send an invalidate to get to an exclusive state. Then, the exclusive state must be resolved, via a write-back request, to a read before new readers are allowed access.
- An unrequested write-back implies that no other tiles had requested either read or write access, so therefore main RAM contains the latest version. Whether the flushing tile does as well is unknown. To err on the side of caution, we note that we should send it an invalidate should the line be modified.
- In the case of an impossible stimulus, the response is undefined.

One small complication arises in the case of a "new writer" being the same tile that already has exclusive access. This technically is a "can't happen" state, but due to event counter delays, it may be possible to squeeze a second store in on the same cache line. In the case of the line being exclusive or exclusive-pending, the "new" writer must not be told to invalidate the cache line, because data would be lost. Therefore, this equality check must be made.

Auxiliary Data Associated with Non-Atomic Transactions

Along with the three new states, there is a new data structure[10] that handles intermediate requests. The "pending address table" contains a list of addresses that are

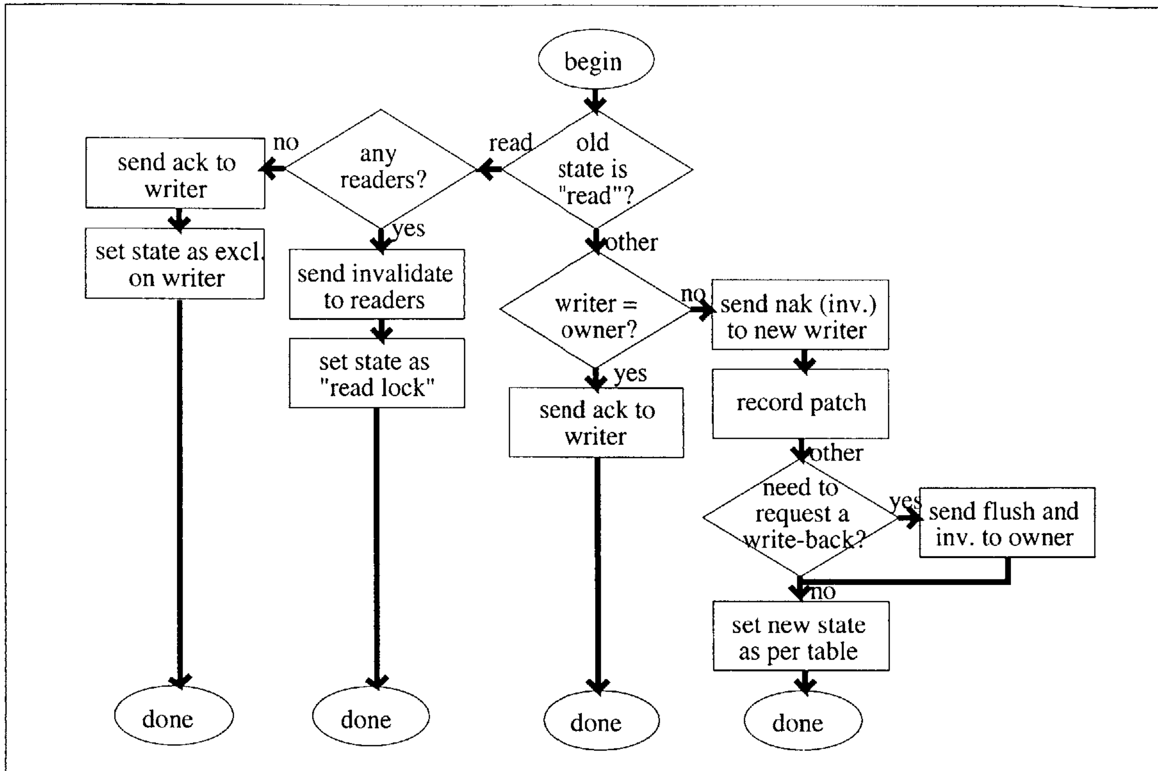


Figure 3-18: Handling of MDN ("New Writer") Messages from a User Tile to the Directory

either exclusive in one processor, or would like to be exclusive in one processor[3].

For each of these addresses, a list of pending readers is stored, as well as the patches applied[6]. When the address is written back to main RAM, the patches are applied and the new readers are sent the line.

3.6.2 Directory Management

Directory management takes place upon either a static network message from an FPGA, an MDN message from a user tile, or a GDN message that has been resolved to not be an interrupt request. I will discuss the responses to all of these stimuli.

MDN Input to System Tile

Any message that is received by the directory from the MDN is asynchronous to any ongoing GDN communications. It is the result of a user tile writing to its local cache, causing a detection of a clean-to-dirty event. This results in the user tile going to Interrupt 6 and sending the address of the line that it dirtied. The directory must then take action. This routine is shown in Figure 3-18.

If the old state is "read", then the directory must must send a flurry of invalidates to the previously reading tiles, as their local copies of the cache line are no longer valid. This is state transition 2a in Figure 3-17.

The send of an invalidate to a user tile is actually an involved process. It is not prudent to do this request in a blocking manner, so we context-switch. We leave the writing tile waiting until the invalidates return[4]. We record the list of invalidations to be made as outstanding requests.

Each tile has associated with it a small data structure, the outstanding request table. Whenever a directory wishes to make a request of a user, it writes to that table. If the number of requests increases from zero to one, then we actively must interrupt the user tile in question. However, this responsibility (to interrupt) is not done by the directory, but rather the interrupt controller. Therefore, we send a request to the interrupt controller of the relevant tile, and the interrupt controller will get the user tile in contact with the directory. When this happens, the list of outstanding requests is sent as a series of GDN messages[5].

Given that the old state is not "read", there is a tile that either has, or wants, exclusive access. At this point, we see if this "new writer" is actually the *same* tile wishing to write another word to the cache line. If it is the same writer, we are done,

[3]All auxiliary data is stored in instruction memory on each Raw tile. This is convenient because we do not have a hazard of a cache miss on this critical data. Jason Miller [36] came up with this good idea.

[4]In the case of there being no readers to invalidate, we immediately proceed to the exclusive state and acknowledge the writer.

[5]I will not describe the exact encoding of messages. They tend to be pretty arcane, grossly sacrificing clarity in the name of saving a cycle in a tight loop here and there. They may be deciphered directly from the appendix. All of the nifty hacks are documented in the comments, for the benefit of the reader.

as the first writer may make as many modifications as it wants to. If it does not have exclusive access, it will upon the readers being invalidated, and therefore it has the freshest version of the cache line.

If we have a new writer, then we must send this writer a negative acknowledgement, since it was beaten in attempting exclusive access. Its modification to the cache line is dutifully recorded as a patch, and the tile is sent an invalidate. If we have not yet requested a write-back of the tile, then we must make a note to do so. If the tile is exclusive, we do so immediately, generating another outstanding request. If the tile is not yet exclusive (read lock), then we will send a flush request instead of a simple acknowledgement once every reader is invalidated, and thus we go to the "read lock bypass" state.

Then, we advance the state as per the state table, ending up in either "exclusive pending" or "read lock bypass".

GDN Input to System Tile

A GDN message may come from either a user tile or from a system tile serving as a directory, and may have as a destination either a directory or an interrupt controller. All of these possibilities are encoded in the GDN message. In this section, I mention the possibility of a GDN message from a user tile to a directory. The other GDN messages have as their target the interrupt controller, which is discussed in another section.

GDN Input to Directory

All messages via the GDN from a user tile to a directory are expected, as they are a function of a previously established communication channel. A GDN message from a user to a directory is one of two things: either a request to the directory to send the user tile a list of outstanding requests, or an acknowledgement that all the requests were handled.

This process is shown in Figure 3-19. The reply to a request for a list is made on the GDN: namely, the list. The directory copies the addresses of invalidate requests

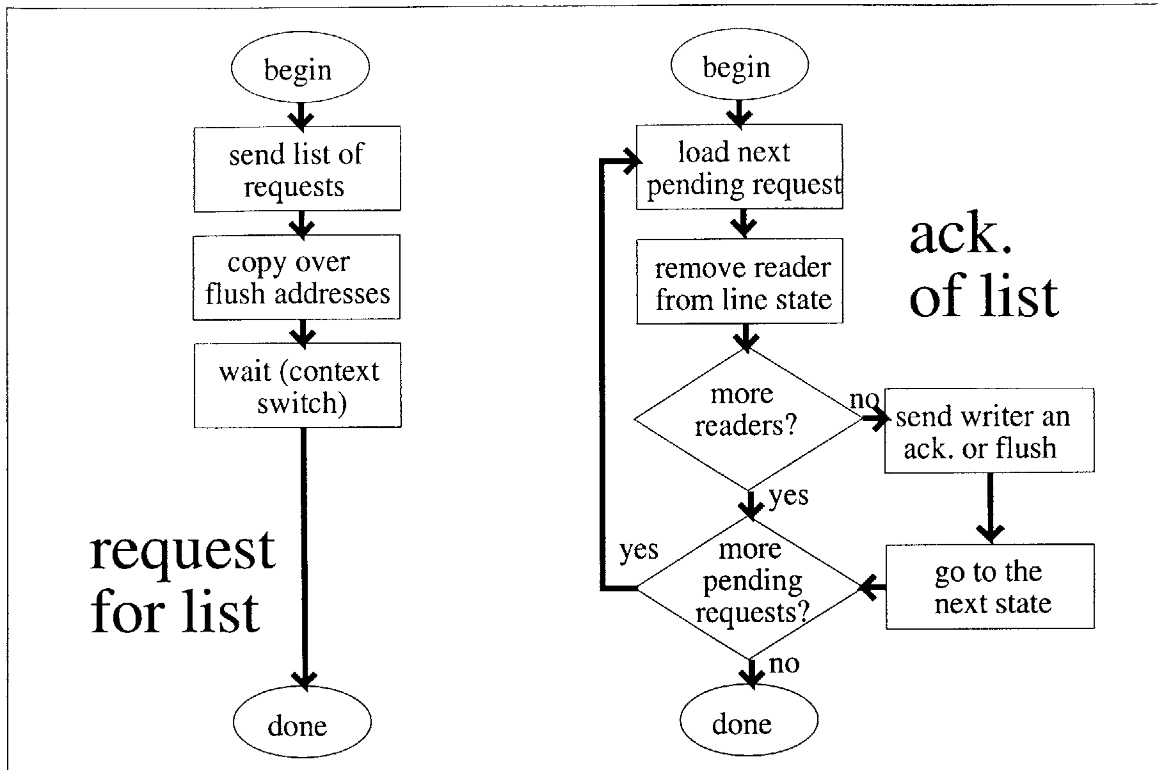


Figure 3-19: Handling of GDN Messages from a User Tile to the Directory

to a new buffer, the pending requests buffer, and then context-switches.

When the acknowledgement returns, the pending requests are all handled and erased. The acknowledging tile is taken off the list of readers in read-lock or read-lock bypass state. When the list of tiles is empty, we finally acknowledge the writer, and move on to the next state: either exclusive or exclusive-pending.

Static Network Input to Directory

The FPGA generates a static network message every time it either loads or stores a cache line. This message contains all of the relevant information about the transaction: originating tile, load or store, address, and data. A load is handled very differently from a store, so the routine immediately forks one way or the other. A flowchart of the static network handler is shown in Figure 3-20.

In the case of a load, the directory first checks to see if the tile is in the "read" state. If this is the case, the line is sent to the requestor, and that processor is added

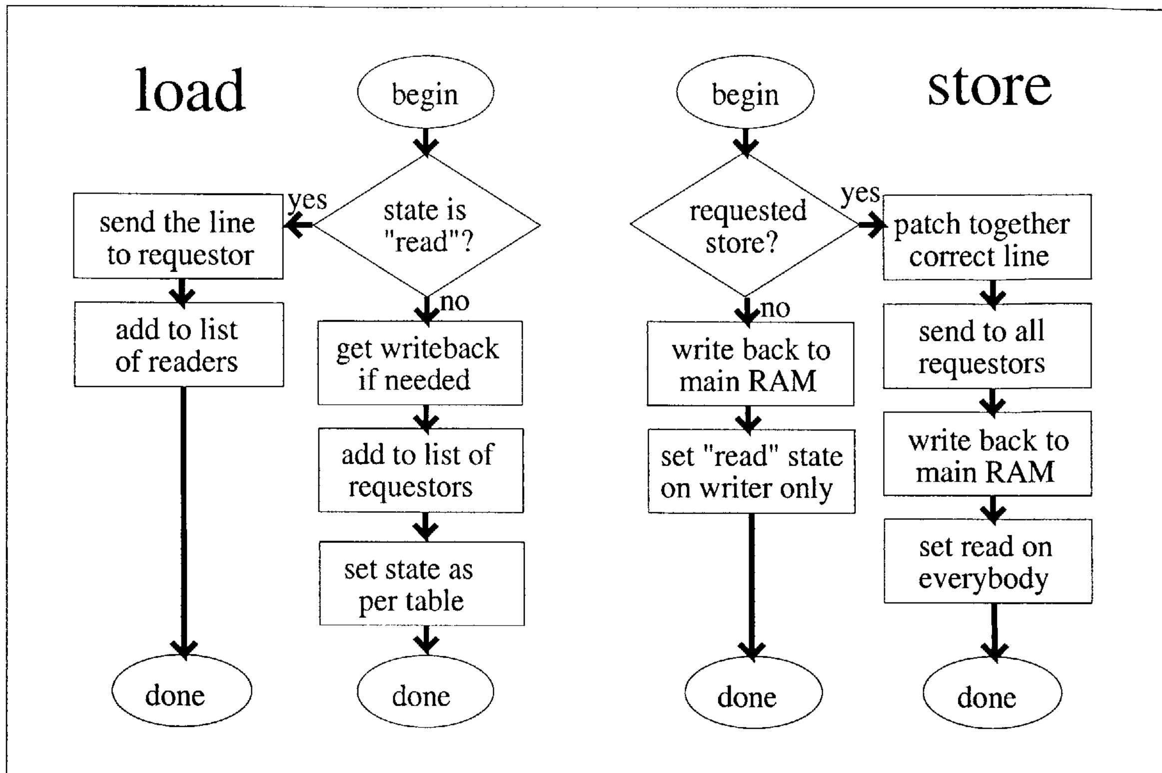


Figure 3-20: Handling of Messages from the FPGA to the Directory

to the list of readers. In any other case, the cache line exists somewhere else, so if a write-back has not yet been requested, that is done. Then, the requestor is added to that list in the pending address table.

Actually, if a read is done in a state other than "read" by the supposed owner, then the tile will stall forever if it does not receive *something*. It is therefore sent the main RAM version of the line. This is an error case, and usually represents the user invalidating an exclusive line[6] for whatever reason, and this is the most correct response.

In the case of a store, the directory checks to see if it was a requested write-back. If not, then there are no other tiles waiting on the line, or any other tiles wishing to modify it. Therefore, it simply sets the read state on the writer (as the line is now resident in both main RAM and possibly the user's cache). If the store was requested, then needed patches are made. If there are patches to be made, then main RAM is

[6]Instructions that do this are available on Raw, and in fact the shared memory system makes critical use of them! The user may, of his own volition, call these instructions too.

rewritten with the correct value. Then, all requestors are sent the line. The final state is read on both the writer and the new readers. The entry is also cleared from the pending address table.

That concludes the tasks of the directory. I now move on to the function of the interrupt controller.

3.6.3 Interrupt Control

The job of the interrupt controller is to facilitate communication between user tiles and directories. Since a user tile is busy executing its own code, it must be interrupted. In this section, I first describe the state maintained in an interrupt controller with regards to what it knows about the user tiles that it may interrupt. Then, I discuss a potential deadlock situation in which the user tile cannot be normally interrupted, and how that is resolved.

User Tile State in the Interrupt Controller

Each interrupt controller serves the three tiles in the same row, and to the west. Each of these tiles has associated with it a state machine on the controller, that describes when it is probably safe to send an interrupt to this tile. All of these state machines are identical, and consist of three states, as shown in Figure 3-21. The three states are "not busy", "pinged", and "busy", with the initial state being not busy.

If the state is not busy, then the interrupt controller believes that the user tile may be interrupted, because it is not servicing a system-level interrupt. When an interrupt is needed, and a tile has this state, then the interrupt controller goes ahead and sends the interrupt. It sets the state to pinged, and notes an associated timestamp. If all goes well, the interrupted tile soon responds to the controller, and the state is set to busy. At this point, the series of communications occurs shown in Figure 3-14 occurs, by which the directory communicates its requests to the user. The last communication in this chain is from the user to the interrupt controller, letting it know that the transaction is done. At this point, the state goes back to not busy.

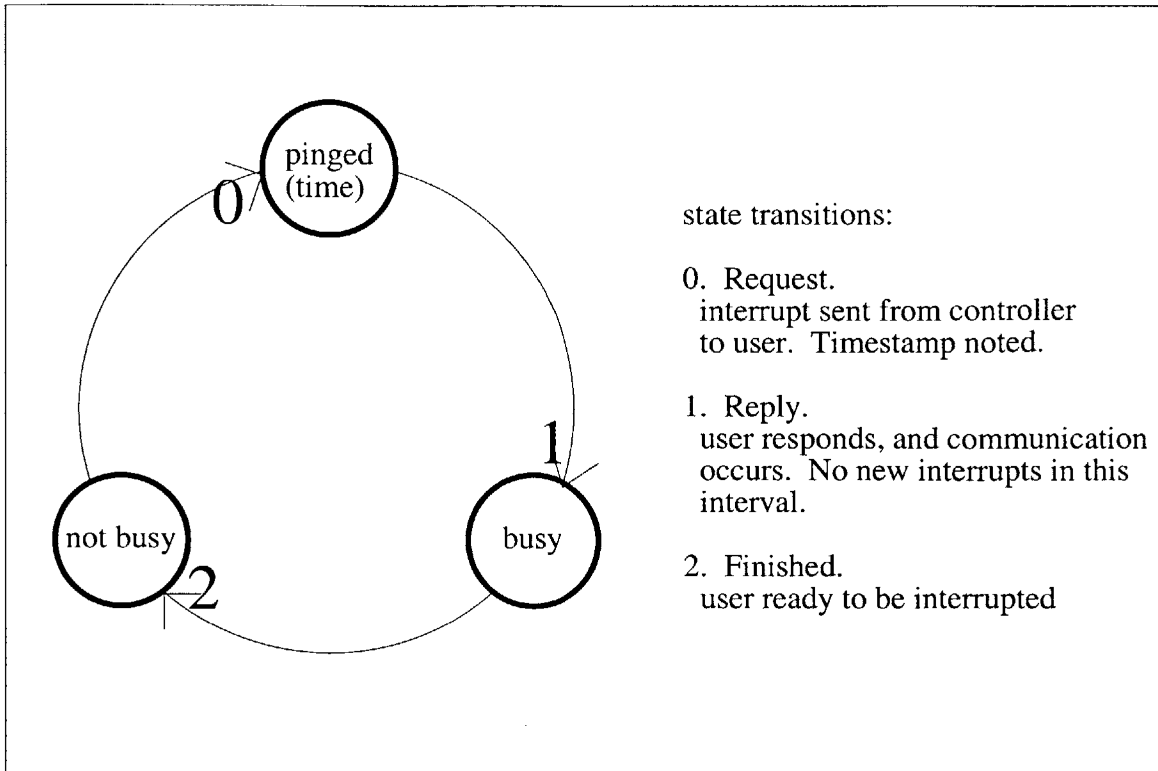


Figure 3-21: Interrupt Controller State Transitions

If an interrupt request arrives from a directory when the tile is either busy or pinged, then the request is delayed until the state returns to "not busy".

Deadlock Resolution

Associated with the "pinged" state is a timestamp. Every time a tile enters the pinged state, this time is checked, and if the tile does not respond within a given interval, a deadlock is assumed. This checking and resolution process is shown in Figure 3-22.

I prove exhaustively the correctness of this deadlock detection and resolution method in a later chapter. For now, I quickly describe the circumstance under which it can arise. Let us assume that tile A has, exclusively, cache line X, and tile B has cache line Y, also exclusively. At the same time, A makes a read request on Y, and B on X. Both of these requests make it to the directory, which then has to send a write-back request to both A and B.

Unfortunately, both A and B are stalled, because the hardware memory controller

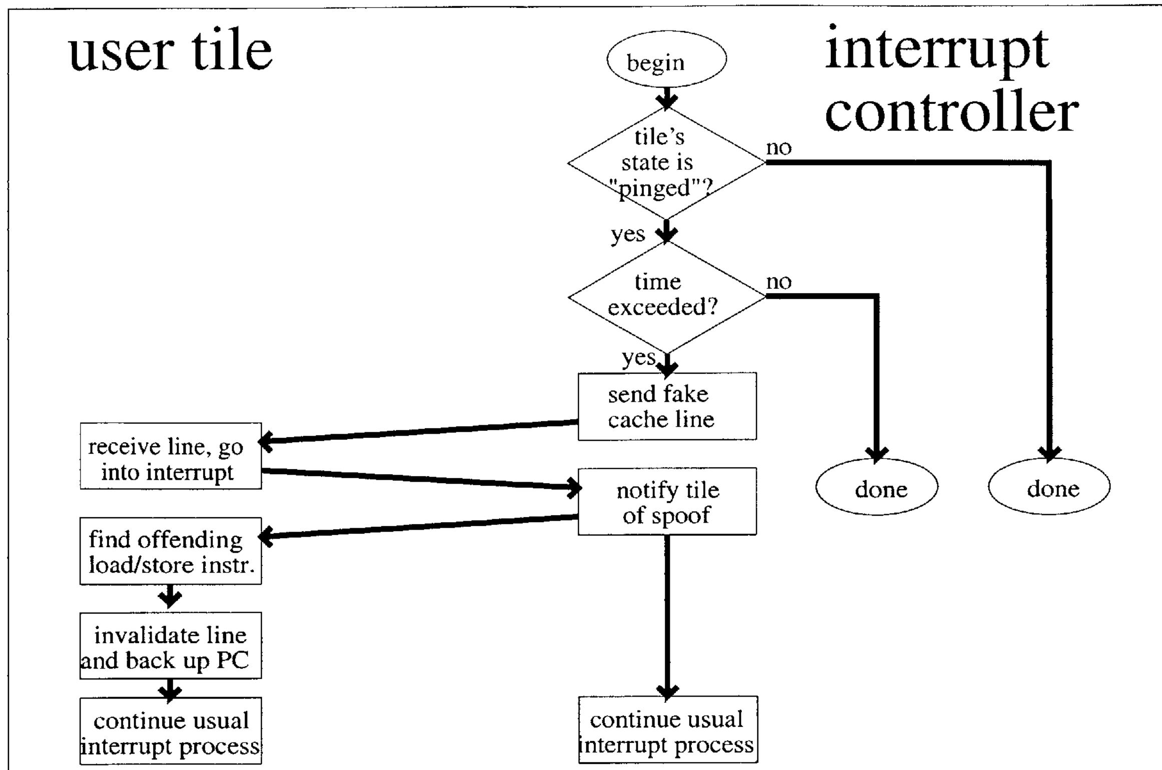


Figure 3-22: Deadlock Resolution by the Interrupt Controller

on Raw stalls until it receives eight words on the MDN. This is a deadlock, because the directory cannot send A's request until it gets it from B, and B will not send anything until it gets something. Thus, there is a cyclic dependency.

However, there is a solution. The stalling tile[7] does not care what exactly it gets, so long as it gets something. Therefore, the interrupt controller, seeing that the tile has stalled for an inordinately long period of time, will send it a false cache line. Satisfied, the tile will then enter interrupt, as an interrupt had been buffered while it was stalled. At this point, the tile is notified of its situation, and it invalidates the incorrect cache line that it received, and backs up its program counter, so that it may go back to waiting once it is done servicing the interrupt. It then services the interrupt as normal, including writing back some lines that are causing other tiles to stall.

I will prove in a later chapter that this deadlock resolution mechanism is correct.

[7]it does not matter which stall, A or B, is detected first, as breaking the cycle at any point will cause the whole thing to resolve.

3.6.4 Memory Requirements of the System Tile

I have not discussed up to this point where the state of the directory and the interrupt controller are stored. Some of the key values are stored in registers, but the cache state table, outstanding request table, interrupt controller's user state table, and pending address table are all stored in main memory.

This main memory is initialised upon startup, and this initialisation is done in such an order that some of the cache state table, and all of the other data structures are in the system tile's local cache. The addresses of these structures is carefully chosen to avoid cache collisions, so that they are not swapped out.

Since the entire cache-line state table cannot fit into local cache (there is one word per cache line, so therefore the entire table is one-eighth the size of all shared memory!), main memory is used as a swap space. Assuming a uniform distribution of all private memory among tiles, and the existence of only a limited (at most 25cache-line state table can fit easily into the main RAM allotted to the system tile. However, lines that are being accessed frequently have their information in cache.

What this means is that there is no explicit memory management on the system tile, other than judicious assignment of addresses. The cache, by its very nature, will store frequently used metadata locally, and therefore the system tile's performance is improved implicitly.

This concludes the analysis of the system tile. I now move on to analysing the user tile.

3.7 Low-Level Analysis of User Tile Functionality

The user tile plays the role of the "processor" in the abstract shared memory system detailed in a previous chapter. That means that most of the time it will be running a user-level application. However, at times a system tile will make certain requests of it, and also the user tile has the responsibility of notifying the directory of any cache lines that go from clean to dirty.

All of these responsibilities are taken care of via interrupts. Interrupt 3 on Raw

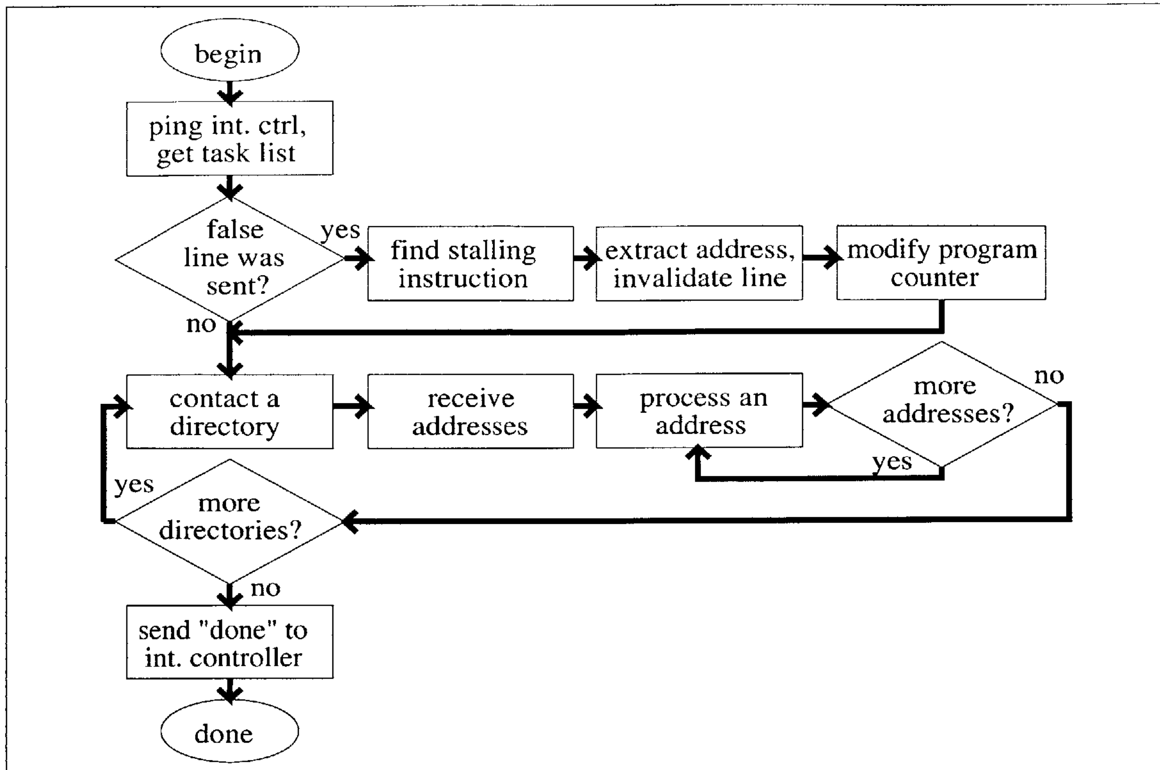


Figure 3-23: Interrupt Handler for External Pings (Interrupt 3)

is the External interrupt, which is the way by which a system tile may initiate communication with a user tile. The directory, by way of this interrupt, may make one of two requests: either an “invalidate” or a “write back”. The user invokes the Interrupt 3 handler, and makes the appropriate modifications to its cache.

The clean-to-dirty detection is done via Interrupt 6, the Event Counter interrupt. If we preset the relevant Event Counter to trigger after exactly one clean-to-dirty transition and call the interrupt handler, then we have a way of catching every transition. Each transition is noted, and the address that is dirtied is sent along to the directory.

The code for these handlers may be found in Appendix A.

3.7.1 External Interrupts

Figure 3-23 shows the interrupt handler for Interrupt 3. The sequence is initiated with an interrupt controller sending an interrupt to the user tile, which then goes

into the handler. The user tile sends a GDN message back to the interrupt controller, and gets a list of tasks. This list has the set of directories which wish to talk to this user tile, and also a possible signal that the latest load had been deadlocking, and thus was resolved by sending a false cache line.

If this signal is present, then the user tile must invalidate the cache line that was loaded, and prepare to load again. This is done by finding the offending instruction, which occured two cycles before the interrupt was triggered. The program counter of the interrupted instruction is grabbed, the appropriate offset is subtracted, and the resulting load is found. The address that was loaded is invalidated in the cache, and the program counter special register is set to retry this instruction when the interrupt handler finishes.

Now that the user tile has taken care of its responsibilities in a possible deadlock resolution scenario, it handles each directory that wishes to talk to it. A list of at least one directory was sent from the interrupt controller, and this directory is contacted via the GDN. When the directory responds, it sends a list of addresses that must be either invalidated or flushed. The user tile processes this list, and then moves on to the next directory, if necessary.

The process of communication is blocking: once the processor commits to the interrupt handler, it will spin at the various stages until it receives the data from the other processes.

Once all the directories are taken care of, the user tile sends to its interrupt controller a "done" signal. This is the end of the transaction, and the user tile returns from the interrupt handler and resumes executing its user code.

3.7.2 Internal Interrupts

Interrupt 6 occurs every time a cache line goes from clean to dirty. In this case, the user tile has the responsibility of sending along the address that was written to the directory that maintains that address. This process is made slightly complicated by the fact that a small number cycles elapse between the instruction that triggers the event, and the interrupt handler. In those instructions, multiple stores may occur,

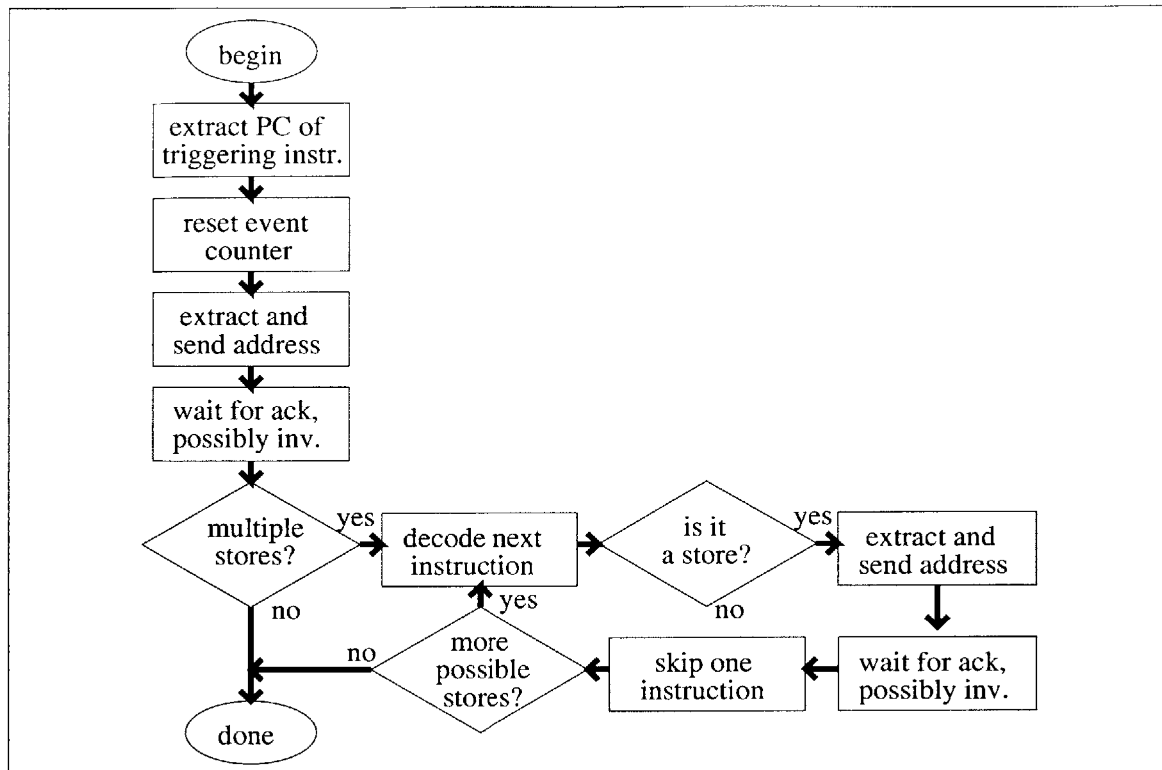


Figure 3-24: Interrupt Handler for Event Counter Events (Interrupt 6)

resulting in multiple addresses that need to be sent to their directories. The handler that uncovers all of these stores and sends along their addresses is shown in Figure 3-24.

It happens that the event counter continues to count even when the interrupt is about to be serviced. Therefore, we can detect the presence of multiple stores, and if this is not the case, bypass a loop that checks the next several (the exact number is the difference between the program counters of the triggering instruction and the instruction that was pre-empted by the interrupt handler - both are stored in special registers) instructions for stores.

The triggering instruction is unequivocally a store. However, other stores must be detected by explicitly testing their opcodes. Note that not all of these stores definitely caused a cache line to become dirty. In that case, the cache line was already dirty, and if we notify the directory twice, then nothing is lost in the way of correctness. Since a hardware limitation prevents consecutive stores, every time we find a store,

we do not have to look at the instruction immediately following it, so we skip that one in our search.

When these stores (both the first, and subsequent) are found, the address that they modify must be extracted and the actual data. This is done by checking the register value stored in the opcode, pulling the data out of that register, and adding the immediate offset (properly sign extended!). Then, the data is pulled out of another register. This address and data, along with the user tile number, is sent in a message to the appropriate directory via the MDN. Then, the routine spins until it receives an acknowledgement from the directory tile. If this is a negative acknowledgement, the line in question is invalidated.

That takes care of the entire description of the implementation of the shared memory system on Raw. The next two chapters are concerned with showing, mathematically, that the implementation is correct, and then analysing its performance through testing.

Chapter 4

Correctness of the Raw Implementation

In this chapter, I demonstrate that the shared memory system I have described is "correct" - in other words, that it behaves in the same manner as the abstract system I first showed in Figure 1-2.

I begin by assuming that the abstract model of a monolithic memory, uniformly accessed by processors without local cache, is correct. This implies that the user knows about the synchronisation and control hazards that are fundamentally present, and will take care to avoid them by using semaphores, locks, or other mechanisms. It is possible to put a correct system into an incorrect state through an incorrect input, such as committing a write-after-write error.

My proof will consist of five parts. In each part, I detail what aspects are inherently part of the system, and what parts require programmer cooperation. The five parts are: transparency of interrupts, absence of buffer overflows, correctness of directory state transitions, lack of unhandled deadlocks, and finally, presence of sequential consistency[30].

At the end, I will summarise again the restrictions placed on the programmer or compiler that wishes to utilise this system.

4.1 Transparency of Interrupts

The first part of the proof deals with the transparency of interrupts from the user's perspective. A transparent interrupt is one that leaves the state of the user tile exactly how it found it. This state consists of RAM and cache, registers, program counter, and networks.

First of all, RAM and cache are not left exactly as they are found - cache is purposely manipulated into a form where it is how the user *expects* it to be. In the interrupt controllers, no data RAM or cache access is done other than the specific invalidate or flush instructions that patch up the cache.

The instruction memory is modified through the storing of registers and network data in unused instruction space as a temporary swapout. The user must not attempt to execute that part of the instruction space. The data is written out at the beginning of an interrupt handler, and read back in at the end, so when the system is not in interrupt, it does not matter what happens to it.

The registers appear to be unmodified, because they are stored away at the beginning of the interrupt handler, and restored at the end. There are two separate banks of storage, in case an Interrupt 3 arrives in the middle of the Interrupt 6 handler.

The program counter may indeed be modified. In the case of a deadlock, the interrupt that resolves it occurs two cycles after the instruction that (falsely) resolves a load finishes. Therefore, this instruction must be repeated. The two instructions after it were already executed once, and thus also must be repeatable. The user has the responsibility of making all instructions within two after a load be repeatable.

The networks may be modified, since the interrupt controller and the directory use the GDN to communicate with the user tile. Therefore, the state of the GDN must be stored at the beginning of the interrupt handler, and then restored at the end before returning to user space. Raw allows for "dynamic refill" [48], in which the GDN is read, and then later accesses of the GDN instead use this buffer. Therefore, we can read up to 31 words off of the GDN. This "should be" sufficient, but still we run the risk of a GDN message arriving from a user tile at precisely the wrong time and

getting mistaken for an expected system message. The solution of this is left to the user: an abstract shared memory system does not have any dynamic message-passing network, and therefore does not foresee this problem!

The MDN has the exact same concerns as the GDN: both incoming and outgoing messages happen on the GDN under Interrupt 6. The MDN already has the restriction of not being used for standard transmissions in the presence of loads and stores. This restriction does not need to be expanded - no user-level message should be split by the arrival of MDN communications in response to a load or a store.

Since the static network is unaffected by an interrupt, we do not have to worry about it.

4.2 Absence of Buffer Overflows

In this section, I demonstrate that various buffers are not exceeded. These buffers all exist on the system tiles and comprise its entire state. They are: the cache line state space, the pending address buffer, the user tile state machines, the outstanding request buffer, and the pending request buffer.

The user tile state machines are a fixed size because there is a known number of user tiles controlled by each interrupt controller, and the state is encoded in one word. The cache line state space is also a fixed size, since there is a known number of addresses to monitor. Therefore, the sizes of these two buffers do not vary, and thus there is no need for bounds checking.

The pending address buffer has explicit bounds checking. This buffer contains two separate sets of addresses: those that need to be patched, and those that are being requested by reading tiles. An address appaers in this buffer at most once, and thus each entry has bits representing both patching conditions and requesting tiles.

Each user tile can only request one address, via the static network, and then stall. Therefore, N slots at the most may be occupied by these requests, where N is the number of user tiles.

There is no bound on the number of cache lines that need to be patched, because

Buffer	Filled By	Skip	Cleared By
cache line state	MDN, GDN, static	none	MDN, GDN, static
user tile state	GDN	none	GDN
pending request buffer	GDN	none	GDN
pending address buffer	MDN, static	MDN	static
outstanding request buffer	MDN, static	MDN, static	GDN

Table 4.1: Dependencies Between Networks

a user tile can issue a store-word that results in a patch, and then continue executing. Therefore, if we set a maximum number of K patches, then the absolute maximum number of addresses in the buffer is $N + K$. Patch requests arrive via the MDN, so we monitor the number of addresses needing patching in the address table, and if we are at the maximum, then we stop reading the MDN.

The outstanding request buffer also has dynamic bounds checking. The maximum number of outstanding requests is practically infinite: every cache line may need to have a transaction done on it, and it is not efficient to make the outstanding request buffer proportional to the number of cache lines, for reasons of parsing it quickly. Therefore, there is a maximum size. When this buffer is full, the system tile skips both the static network and the MDN check.

The pending request buffer is set to be the same size as the outstanding request buffer. At most, the number of pending requests moved into this buffer is equal to the maximum size of the outstanding request buffer, and this buffer is cleared entirely between writes. The transmissions are in two stages that strictly alternate (writes and clears) due to the nature of the communication between directories and user tiles: the user tile must ask for outstanding requests (filling the pending request buffer), then acknowledge (emptying it), and only then is it in a position to ask for more outstanding requests again. Therefore, no explicit control is needed over this buffer.

Table 4.1 shows the various dependencies between the networks. The first three entries are buffers which do not need active overflow prevention measures. The last two buffers are occasionally skipped. It is important to note that if the outstanding request buffer is full, then the static network is skipped, and the pending address buffer cannot be cleared. However, the outstanding request buffer is cleared by the

stimulus	R	RL	RLB	E	EP
read by reader/owner	R0	UI	UI	UI	UI
read by new tile	N	N	N	N	N
clean-to-dirty by owner	N	B	B	B	B
clean-to-dirty by someone else	R3	N	N	N	N
writeback by owner	F	F	F	N	F
writeback by someone else	F	F	F	F	F
invalidate ack. sent by reader	NS	N	N	NS	NS
invalidate ack. sent by someone else	NS	R7	R7	NS	NS

Table 4.2: Responses to All Stimuli, Possible or Otherwise

GDN, which is never skipped, so therefore eventually both buffer limitations will be resolved, and there is no possibility of deadlock.

4.3 Correctness of State Transitions

In this section, I will deal with the correctness of state transitions in a full-map directory. I have discussed the more normal state transitions in previous chapters. Here, I will focus on the abnormal ones, and show that they do not result in disaster.

Table 4.2 shows the responses to all possible stimuli, in all possible states. The key is as follows.

- N (normal) - these cases were covered by the discussion on expected state transitions.
- F (forbidden) - the user is forbidden from invoking operations with this result. In the 6 cycles between a store and the corresponding event counter interrupt, the address and data registers may not be changed, and furthermore no instructions (load, store, or flush) may be called that could flush the line back to main memory. This prevents a race condition of the event counter event arriving *after* the corresponding store to main RAM. After the event counter triggers, the writing tile is stalled until state is resolved correctly, so there is no race condition unless the event is started in the before-interrupt window.
- B (bypass) - a cache line may not be dirtied twice, by definition. However, it

may appear that this is the case due to multiple stores in the window between an event-counter-triggering store and the interrupt firing, so multiple stores to the same line will be reported repeatedly. In order to not invalidate data that is only in a single cache, and not in main memory, any clean-to-dirty event on a line already dirty and owned by the same tile must be ignored.

- NS (no solicit) - there are only certain states in which we expect incoming invalidate acknowledgements. We do not enter read-lock until we send out the invalidate requests, and do not leave read-lock or read-lock-bypass until all of them are acknowledged, so therefore in any other state, it is impossible to get that particular GDN message.

- UI (user invalidate) - Must be the result of a user invalidate, because the line was known by the directory to be cached - any flush would have been detected and handled. This is an error condition on the user. Send along the line from main RAM, because that is the most likely cause of this error (the user wanted a fresh copy, and therefore we send what we can), and if we do not send along anything, then the tile will stall. It is not correct to treat this as a read by another tile, because the line in question is no longer valid in the writer's cache, so we cannot start the process of writing it back, as the resulting flush would fail, with undefined consequences.

- R0 (misc. impossible) - not expected, but not fatal. This can only happen if the user purposely invalidates a cache line and loads it again. The requestor is sent another copy, and state is unchanged.

- R3 (misc. impossible) - not expected, because on Raw, a store that misses is resolved as a load of the entire cache line followed by a store to local cache that causes a clean-to-dirty event. The event will not occur before the load, and this is physically impossible. If it happens, however (by manually forging an MDN message), this is not fatal. The states proceed as though it were a reader that requested exclusive access, and all of the invalidates are sent. In

fact, the requestor is never verified against the list of readers, since the routine works roughly as "mask out the requesting tile from the list of readers, and see if the list is empty. If not, read lock is needed. Otherwise, proceed straight to exclusive". A zero bit may be masked to a zero bit in this unusual case.

- R7 (misc. impossible) - not fatal. The directory does not know the difference between an invalidate acknowledgement sent by request, and one sent by a superfluous tile. A bit that happened to already be 0 is set to 0 and things proceed. This is impossible unless the user forges the interrupt handler.

4.4 Deadlock Avoidance

In this section, I prove that there is precisely one kind of deadlock that may occur in the system. First, I show this deadlock condition. Then, I demonstrate the correctness of its resolution. Finally, I show that no other deadlock is possible.

4.4.1 Dependencies Among Components

A deadlock is, by definition, a cyclic dependency. This section therefore must find all the dependencies in the system, and show that they resolve without circularity. These dependencies occur either in the form of a stall (wait until some condition occurs), or a bypass (do not execute a certain piece of code until some condition occurs).

A dependency graph of the system is shown in Figure 4-1. They are divided into four major groups. "Cache line sharing" involves the writing back of exclusive lines to main memory and other tiles, including assembly of patches by the directory. "Messages" is the protocol by which a directory communicates with a user tile. "System tile functions" are high-level functionalities on the system tile that deal with various network components. "Routing network" is the physical routing - the MDN is broken into the to-the-east (messages to the directory and to the FPGA) and the to-the-west components. Further details about each component and each dependency arrow are in Tables 4.3 and 4.4.

Component Name	Description
writes acked	Clean-to-dirty events are acknowledged.
invalidate done on demand	Reading tiles invalidate stale lines.
writeback done on demand	Exclusive words ends up in main RAM and directory.
cache line assembled	Directory correctly assembles cache line components.
cache line sent to user	Readers requesting a cache line receive it.
user is not stalling on miss	Cache miss resolved by received data.
interrupt 3 handler on	Interrupt 3 results in a timely manner.
user-to-int "done"	User notifies controller that it is out of interrupt.
dir-to-user request list	Directory's requests received by user.
user-to-dir ping	User contacts directory and asks for requests.
int-to-user list of dirs	Interrupt controller sends list of dirs to user.
user-to-int reply	User lets controller know that it is in interrupt.
interrupt 3 (external)	Message from interrupt controller to user.
system tile not stalling	Write to network not blocked.
GDN handler called	System tile routine that reads GDN.
GDN handler called	System tile routine that reads MDN.
Static handler called	System tile routine that reads static network.
GDN routed	Messages routed correctly on the GDN in due time.
MDN west routed	Messages on MDN westbound (to users) routed correctly.
MDN east routed	Messages on MDN eastbound (from users) routed correctly.
Static network routed	Messages from FPGA routed to directory.

Table 4.3: Components of the System

Dependent (arrow tail)	Master (arrow head)	Description
cache line assembled	static handler called	assembler routine
cache line assembled	writeback done on demand	components must arrive
cache line sent to user	cache line assembled	send only final product
cache line sent to user	MDN west routed	network needed for data transfer
dir-to-user request list	GDN handler called	list sent by GDN handler
dir-to-user request list	user-to-dir ping	lists sent only on demand
GDN handler called	system tile not stalling	instructions must be issued
GDN routed	GDN handler called	read data, free up buffers
interrupt 3 (external)	interrupt 3 handler on	handler receives interrupt
interrupt 3 (external)	MDN west routed	MDN routes interrupt
interrupt 3 (external)	GDN handler called	GDN handler causes interrupt
interrupt 3 (external)	user-to-int "done"	interrupts strictly serial
interrupt 3 handler on	user is not stalling on miss	no interrupts in stall
int-to-user list of dirs	GDN handler called	list sent by GDN handler
int-to-user list of dirs	user-to-int reply	list sent only on demand
invalidate done on demand	dir-to-user request list	must be demanded
MDN east routed	MDN handler called	read data, free up buffers
MDN east routed	static network routed	FPGA routes from one to other
MDN handler called	GDN handler called	buffer overflow avoidance
MDN handler called	system tile not stalling	instructions must be issued
static handler called	GDN handler called	buffer overflow avoidance
static handler called	MDN handler called	buffer overflow avoidance
static handler called	system tile not stalling	instructions must be issued
static network routed	static handler called	read data, free up buffers
system tile not stalling	GDN routed	data issue into network
system tile not stalling	MDN west routed	interrupt issue into network
user is not stalling on miss	cache line sent to user	resolve the miss
user-to-dir "done"	dir-to-user request list	must start job before finishing
user-to-dir ping	int-to-user list of dirs	must know which dirs to ping
user-to-int "done"	user-to-dir "done"	sequential in protocol
user-to-int reply	interrupt 3 (external)	handler issues reply
writeback done on demand	MDN east routed	writeback must reach directory
writeback done on demand	dir-to-user request list	must know what to write
writes acked	invalidate done on demand	ack only when all invalidates done
writes acked	MDN west routed	acks are sent via MDN west

Table 4.4: Dependencies Among Components

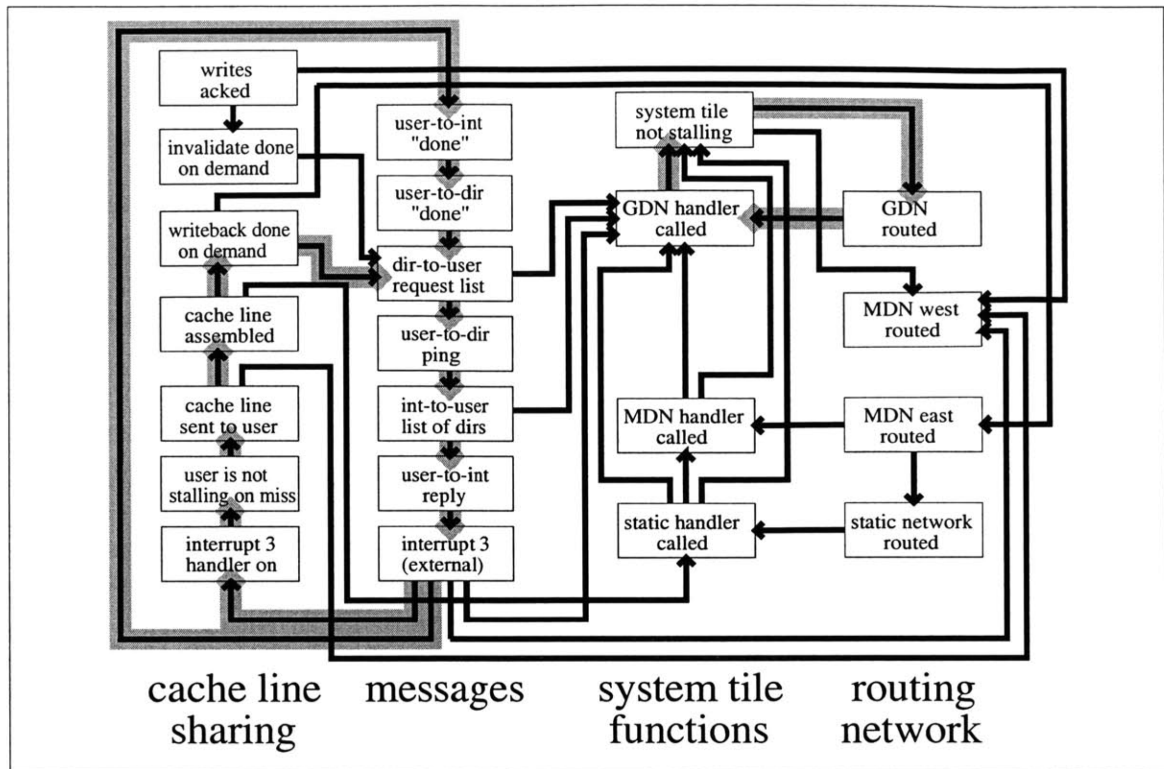


Figure 4-1: All Dependencies in the Shared Memory System

There are three circular dependencies. Two of them share the main sequence of message events, and one is in the GDN. Note that this is a dependency graph, not a causality graph - events in time occur against the flow of the arrows.

4.4.2 Dependencies Involving Cache Line Sharing

The first of the two dependencies involving the message event is not actually circular. The head of the message protocol is connected to the tail to signify that the issue of the *next* Interrupt 3 is dependent on the current one finishing. The current Interrupt 3 does not depend on its own conclusion! Therefore, there is no cycle there.

The second cycle is non-trivial: starting with the issue of Interrupt 3 and working backwards in time (with the arrows), the successful issue depends on the handler being accessible in the user tile, which means that the user must not be stalling on a cache miss. That implies that cache lines must be sent to waiting users, and thus have been properly assembled, so the individual components (resident on *other* must

have been written back. That means that the basic sequence of communications must have gotten to the write-back stage through the various handshaking, and in fact must have been started with an Interrupt 3.

There is no cycle yet, because the requesting and owning tiles are different. But, let us run through the cycle again, reversing their roles. Then, a requesting tile is dependent on its own write-back of a cache line that someone else wants. In space, this means that two tiles are stalling, waiting for cache lines that the other has, but refuses to give up because it is stalling.

I will discuss the active solution to this deadlock in a later section. I now move on to the third cycle in the graph.

4.4.3 The General Dynamic Network is Read Sufficiently Quickly

There is a possible cycle between the blocks "system tile not stalling", "GDN handler called", and "GDN routed". The system tile not stalling depends on the routing of messages out of the processor into the GDN so that the processor does not stall on a write to the network. The GDN router depends on the reading of the GDN by message recipients so that it may route packets forward. Finally, the GDN handler depends on the system tile not stalling so that it may be called in the course of program flow!

In this section, I prove that the GDN is read fast enough as to not backlog and cause a deadlock. First, there is no deadlock possible between user and system tiles on the GDN, as the communications occur completely synchronously, one after another. If one party is stalling, it will not generate a response, and thus the other party cannot possibly stall.

The only possibility, then, is messages between system tiles - the case of a directory communicating with an interrupt controller. The possible deadlock would arise if one tile is stalling due to its output to the network being full, and then the tile who was supposed to read its message also undergoing the same problem.

First, given the previous absence of deadlocks between user and system tiles, any traffic coming from the user tile to the system tile will eventually be read. Due to round-robin scheduling in the routers, messages from the user tiles will effectively alternate with those from the system tiles. Therefore, the message that is "stretching" from one tile's output to another's input will eventually make it into the other tile. Given that a tile is not stalling on output, any message destined to it will reach it. Therefore, the only way to get the deadlock is to *simultaneously* launch a series of messages between system tiles. It is possible to do this, but not often enough to clog up the network. In the worst case, all directories wish to interrupt every user tile, thus necessitating a send of 12 messages per directory - 48 total. Of these, three out of every four go to a physically separate system tile, while one out of four is routed internally. Therefore, 36 messages are placed on the dynamic network. When a message reaches the input port of a processor, the header is stripped, so the message is of size 1; otherwise it is size 2.

There are 16 input slots (4 system tiles, each with 4 words of buffering), so that leaves 20 messages on the network, for 40 words total. This is fit into a total of 40 slots: 16 in the form of four each on the outputs to the network, and 24 as four each, in 3 hops, in 2 directions, for the slots in the routers themselves.

This implies that we can squeak by with overloading the network with a flurry of interrupts without any backlog. This flurry could not take place again until the interrupts are acknowledged with messages from the user tiles to the directories, which by definition must occur when those messages are arriving in the network, and thus there is space for them. Thus, we may be stalling, but we cannot deadlock, as the stalls would all be the result of messages that are from the user to the system tile (or the other way around), which are precisely serialised.

4.4.4 Correctness of the Deadlock Bypass

I now prove the correctness the active deadlock bypass mechanism. This mechanism detects deadlock by waiting a certain number of cycles and then issuing a false cache line that forces the processor out of stall, and then sending a repair message and the

original list of directories that triggered the interrupt in the first place. The processor thus disturbed invalidates the falsely loaded line and backs up the program counter.

The correctness of this mechanism is dependent on two things: it executes correctly when it is needed, and it does not execute at all when it is not.

I first prove that the correction is sent if and only if it is needed. This is done by proving that all non-deadlock events resolve within a certain length of time. The deadlock is detected by the processor not responding to an Interrupt 3 request. Since Interrupt 3 is system level, a lack of response cannot be blamed on the processor handling Interrupt 6 at the time. This leaves only the possibility of interrupts having been disabled, or the processor stalling on a network request.

Interrupt 3 is disabled only by the user. Therefore, any failure to re-enable it in a timely manner is a programmer error. I also assume in this proof that the user has not botched the networks in such a way that he causes a user-level deadlock.

The tile may be stalling, then, on a system-level network request - either inbound or outbound. Inbound requests from the system include cache lines on the MDN, or GDN messages in confirmation of an event-counter event. A GDN reply to an event counter event arrives as soon as the GDN is unclogged. The GDN has been shown to be deadlock-free, and the resolution thereof is at most several hundred cycles. The system tiles read the GDN diligently and the vertical routing paths, which have at most 16 words between the issuing processor and the horizontal path (twelve in 3 hops, 4 in the processor's outgoing buffer), will be read after at most one iteration through the system's main loop (34 cycles in the worst case) and seven additional cycles (eight total cycles, because 16 words are at most 8 messages) through the GDN handler (392 cycles each). This takes at most 2778 cycles.

Therefore, the tile that is in Interrupt 6 will receive the message in a length of time much less than the threshold (20000 cycles) for the deadlock detector.

On the outgoing side, the only system-level message that is possibly being executed in Interrupt 6 on the user tile is an MDN send, which is far faster than the GDN receive, due to the lower amounts of traffic on the MDN. Any other message-based stall must be due to user error.

The only event that can possibly take longer than 20000 cycles to service is the event we are looking for: a cache miss. In any ordinary case, the cache line is assembled after various trips through the GDN, which take a total number of 2778 cycles, and then sent to the receiving tile. In the case of the tile waiting longer than this amount, it must be deadlocked. Therefore, the deadlock detector is triggered if and only if it is needed.

Now, I show that the deadlock detector does the correct thing. The critical aspect of the correctness is to detect the instruction that forced a cache miss and back up to it. Given that the tile was detected to be in deadlock, an Interrupt 3 must have been issued by the controller at some time in the past. Therefore, the exception vector is set, and when the cache line arrives, the interrupt triggers in precisely 2 cycles[48]. Then it is just a matter of backing up two cycles, pulling the address in question and invalidating it. Furthermore, since the tile finally is in the interrupt handler, we have achieved what we have set out to do in the first place, which is to get the writeback. The tile will not leave Interrupt 3 until it receives and services the outstanding requests.

4.4.5 Execution in a Deadlock-Free Environment

Figure 4-2 shows a dependency graph that is free of cycles. The GDN cycle was broken by removing the arrow of implication between "system tile not stalling" and "GDN routed", because I have shown that the GDN cannot be routed so badly as to cause the system tile to stall. Then, I removed the loop-around from "Interrupt 3" to "user-to-int 'done"', as that may appear cyclic in space, but is not so with an additional time dimension. Finally, I broke the mutual-exclusive cycle by removing the arrow between "user is not stalling on miss" and "interrupt 3 handler on", allowing for an external interrupt even in the case of a cache miss.

Darker blocks represent things further down in the chain of dependency. It is interesting to note that there are two things at the top of the chain - namely, things that do not depend on anything else: the MDN is routed west, no matter what (this is a feature of the underlying Raw fabric, which I did not modify in this aspect),

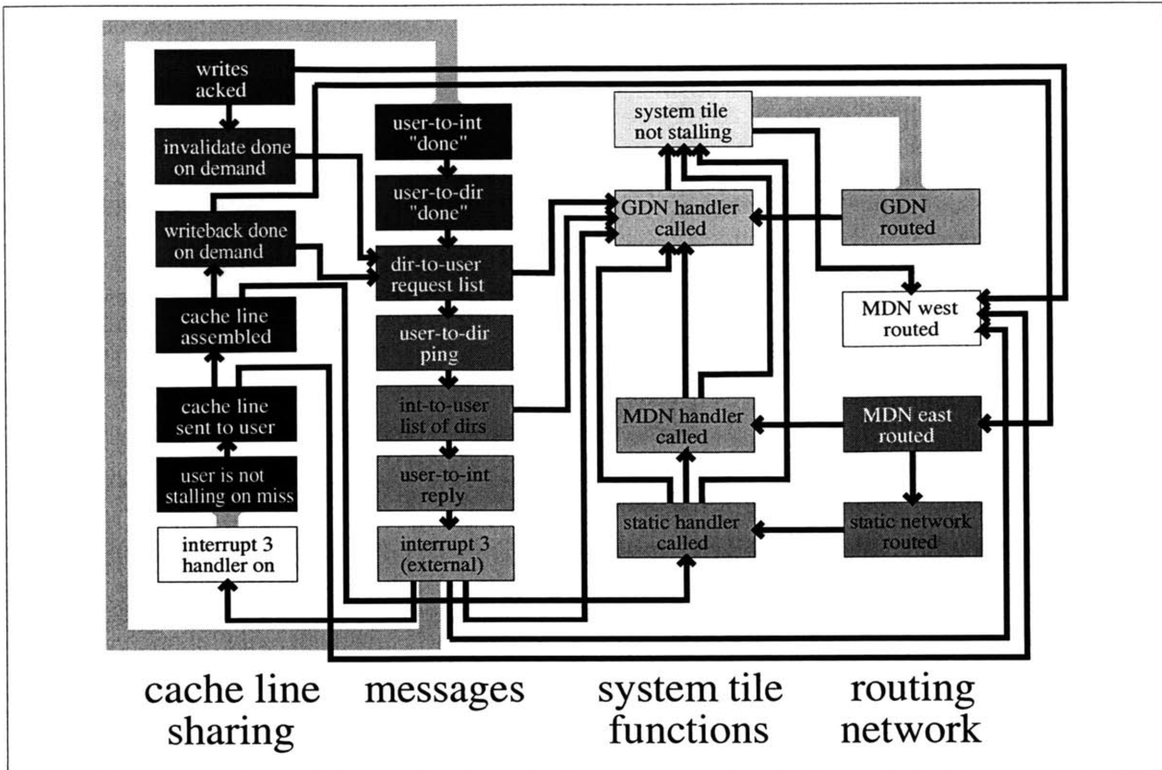


Figure 4-2: Cycle-Free Dependency Graph

and the Interrupt 3 handler is always triggered in a timely manner. This second feature is a non-trivial accomplishment. Slightly further up the chain are "system tile not stalling" and "GDN handler called": important events that should not deadlock, based on the large number of things that are dependent on them. The most dependent item on the list is "user is not stalling on a miss", as it depends on an entire slew of communications to be handled correctly.

4.4.6 Absence of Further Deadlocks

There are no more possible deadlocks, because there are no more dependencies in the graph that could become cyclic. The graph was derived from analysing the code, with instructions checked for dependency based on the criteria of stalls or bypasses. Most redundant connections were removed, as it is correct, but not illustrative, to add a direct link between to events that are also indirectly linked.

4.5 Sequential Consistency

This final section proves sequential consistency. I show the following two things:

- All transactions (loads and stores) issued by a single processor are handled in the order that they are issued.
- All processors see the same ordering of the transactions issued by all processors.

4.5.1 Single-tile Ordering

The proof of single-tile ordering breaks down to several cases. I show that two events are never swapped in order, which implies that full ordering is maintained over all transactions.

From the view of the processor, no two transactions are swapped in order because one transaction completely finishes before the next one begins. In the case of a load, the processor blocks until it receives the line. The MDN request is sent to the FPGA, which bounces it to the directory, not changing the order in which requests are received. Then, the directory processes the first request in the queue at all times, and does not start processing the next request until it has sent the cache line to fulfill the current load.

In the case of a store, the event counter interrupt causes the clean-to-dirty event to stall until acknowledged. The directory tile does not acknowledge until it has updated the state correctly, and therefore does not handle any more stores.

Since the event counter takes six cycles to trigger, the user is forbidden from doing anything that would violate this order - multiple stores stores may be issued within this window, but none of them may be written back to main RAM before the event counter handling them triggers. The necessity for this is shown in the next section.

4.5.2 Sequential Consistency Across Multiple Processors

The second part of the proof is to show that all processors observe the same ordering of events. This statement is equivalent to "all reads must reflect the latest write"[30].

First, a note about causality[16]. There may be a small problem with the word "latest", as events may be unorderable due to their being separated in space so much (or, in time, so little) that one event could not cause the other.

We must, therefore, assure that causality is maintained, and there is a meaning to the word "latest". The effect of every write must be propagated before further writes occur. It is sufficient to require the effect to be acknowledged, as propagation plus acknowledgement by definition takes longer than propagation. Therefore, a writing processor may not be allowed to continue until all the readers have invalidated the line. This takes place in the transition from "read" to "read lock" to "exclusive" (or, possibly from "read lock" to "read lock bypass" to "exclusive pending" - the idea is fundamentally identical). Therefore, the shared memory system does not allow consecutive writes to appear out of order, from the perspective of any user.

Furthermore, all reads reflecting the latest write implies that when the read is resolved (assuming the previous concession to causality, which implies that a reader's cache must miss after a write and before the subsequent read), the latest version of the line is sent. This is guaranteed by assembling the line and accumulating readers, and only sending out the line after it is received and reassembled, and in this interval not allowing any tile to maintain a fresher line exclusively in cache.

The readers are accumulated (as opposed to instantly serviced) in any state except "read", and when the "exclusive-pending" state goes to read, only then is the line sent out[1]. No individual tile has a later version than what is sent out, because upon an attempt to gain exclusivity being received, the changed word is stored for assembly, and the writing tile is sent an immediate invalidate. Therefore, at the time of sending out the line, it is indeed the latest.

Thus, all writes indeed reflect the latest read, and, combined with the ordering of single-processor events, sequential consistency is maintained. The shared memory system is correct.

I now reiterate the programmer's responsibilities in maintaining this correctness.

[1]It is possible to go from "exclusive" to read, but that implies no readers.

4.6 Summary of Programmer Responsibilities

The system is designed to minimise the amount of work that the programmer must do to maintain cache coherence. Most of these requirements arise because of the small hardware restrictions of Raw encountered during the development of the shared memory system. Others are known problems of any interconnection network and any shared memory architecture.

4.6.1 Known Network Hazards

There are two main classifications of hazards that are not specific to Raw. Both of these have solutions in outstanding literature, and are known problems.

The first is the necessity for synchronisation. Any shared memory system, even the abstract ideal discussed in the early chapters, is susceptible to write-after-write hazards. Sequential consistency[30] guarantees that writes made in parallel will have their resolution reflected universally, but it makes no guarantees as to what that resolution actually is. There are no low-level mechanisms for synchronisation on this system, so therefore the user is expected to design locks, semaphores[13], and or other synchronisation mechanisms[12, 30].

The other hazard is one of networking. It is possible to tie up a point-to-point interconnection network, such as Raw's MDN or GDN, through careless programming. While a dimension-ordered network is intrinsically deadlock-free, it is possible to deadlock the processes that are using the network. One possible deadlock exists, and is solved, in the shared memory system. It is entirely possible to come up with more either in user space, or in an interaction between the user and the system. One easy pitfall is to receive an interrupt while the network is being used by the user. I did not explicitly address this problem, because on the ideal shared memory model we were emulating, there is no direct processor-to-processor communication. Therefore, if the user introduces it, he must do it carefully. It is quite possible for the interrupt handler to get confused by a user message arriving when it expects a system message. Therefore, the user must prevent this from happening.

4.6.2 Hazards due to Raw Hardware Restrictions

The other three problems I will discuss are hazards due specifically to the implementation on Raw. One is a hazard that arises on a load[2], and another on a store. The third is a result of how the networks are accessed on Raw.

Load Hazard due to Possible Deadlock Resolution

On a load, the network stalls, and possible deadlock arises. When the deadlock resolution line arrives, the interrupt that triggers it only is handled two cycles later. Therefore, the load, and the two operations after it, must be repeatable. This includes the absence of any branches. Furthermore, the register that holds the address loaded must not be changed in these two cycles, because it is used by the interrupt handler.

4.6.3 Restrictions on Stores due to Event Counter Limitations

There is a delay of at least six cycles between a store and the event counter firing. This delay is known to be six unless the user turns off the interrupts, in which case the interval is longer, but still deterministic. Within this interval, the following requirements must be met:

- Positively no branches must be taken. This would completely foul up the routine that searches for store-word operations *between* two instructions.
- The registers holding the address and the data of the store operation must not be changed. This is because the interrupt handler reads them and forwards them to the directory.
- No store is allowed that would cause a previous store to be flushed to main RAM. Stores in general are allowed, as the interrupt handler will detect and process them, so long as they are guaranteed to not cause cache clash.

[2]This load may of course be one generated by a store missing on a cache line.

- All stores to shared memory must have a "1" as the low bit of the address. This is stripped off by the hardware, but the event counter triggers only on odd addresses (this was implemented on Raw to add selectivity and aid in debugging[48]).

The easiest way to meet the first two restrictions is to simply throw in six cycles of no-op after every store. To increase performance, carefully designed instructions may be attempted. The third requirement is easily met since the store instruction has an immediate field, which may be incremented by one at compile time.

Register-Mapped Network Port Hazards

The last Raw hardware restriction is a direct consequence of the register-mapped network ports. When a value is read from or written to a network port, it disappears and the operation cannot be repeated by the processor. Therefore, no memory instruction can use as an operand a virtual register, because it will be either inspected or repeated by the interrupt handlers.

4.7 Implications of Programmer Responsibility

The necessity of programmer restrictions implies that our system is not as transparent as in the ideal case. However, these restrictions are not overwhelming, and are only a minor performance hit. A compiler should have no difficulty scheduling instructions for repeatability; at worst by strategic placement of no-ops.

The user does not need to explicitly handle memory through the use of cache management instructions. Therefore, the integrity of the design is not compromised.

Chapter 5

Performance of the Shared Memory System

In this chapter, I analyse the performance of the system. First, I give the basic methodology, and then I show the results of both certain unit tests, as well as an application. The application is an edge detection routine[20], that was developed to compare the performance of several system designs on the Raw architecture[21].

5.1 Methodology

The results in this section were gathered through the use of btl[47], the Raw cycle-accurate simulator. I stepped through the programs and noted the cycle count at which various routines started and finished, and from that got the cycle counts shown herein.

Due to some restrictions in the simulator, certain scaffolding elements had to be used. Since event counters are not implemented, I had to manually place the jump after every sw to the interrupt handler[1], after an appropriate delay to simulate the event counter triggering. This does not distort the numbers given, nor change the

[1] In retrospect, this may not be a bad way of doing things, permanently. It saves a few cycles here and there, and simply forcing the programmer to place a jump after every store prevents him from doing damaging things for six cycles. Its main disadvantage is that one has to know at compile-time which stores will dirty a cache line and which will not.

correctness of the system.

Also, due to the MDN not firing under this ersatz "interrupt handler", the GDN was used instead for sending asynchronous messages from user tiles to the system. This does not change the dependencies discussed before.

All of the tests, both unit tests and the application, were written in Raw assembly language. The simulated FPGA memory controller was written in the programming language bC[47], an on-the-fly interpreted multithreading version of C written by Michael Taylor.

The application was run entirely on the simulator. The cache line data for all the addresses used was loaded into the caches of the directory tiles beforehand. In order to simulate event counters, a forced jump was taken after every store, even those that clearly would not have made a cache line go from clean to dirty. The cycle counts may be off by a small amount here and there due to my stepping more than one between observations. Also, the network delays are variable, as are the process switching delays on the system tiles, so all the profiling results of the application should be taken as an approximation to within about 30 cycles.

5.2 Results

In this section, I discuss the performance of the shared memory system. This analysis is based on three levels. First, the low-level routines are analysed and cycle times for various functions are extracted. Then, these functions are combined and the cycle times are derived for various service calls: loads and stores, given a variety of states. Finally, an application[20, 21] is run on the system, and its performance is analysed.

5.2.1 Low-level Results

First, I go through the system, component by component, noting the time taken by a variety of subroutines. These lengths may vary by a few cycles here and there due to branch prediction misses. However, all main RAM accesses by the system are assumed to be in cache. A cache miss results in a penalty of 24 cycles. I start first

Routine	Time
generate first outstanding request	74
add subsequent requests	64
new reader in pending address table	$13 + 5A$
new patch in pending address table	$16 + 5A$
recognise need for writeback request	9
parse a request on the static network	11
send a line from RAM to a reader	49
assemble and store a cache line	$132 + 4A$
broadcast a line to a waiting reader	21
parse an MDN request	16
set read lock state	13
parse a GDN request	24
generate an interrupt	16
send a list of directories to a user	11
send a list of N requests from a directory	$12 + 10N$
process N request acknowledgements	$11 + 12N$
process "done" sent to interrupt controller	5
send a deadlock resolving packet	33

Table 5.1: Cycle Times of Various Routines on the System Tile

with the components of the system tile, and then discuss the interrupt handlers on the user tile. These tables show worst-case latencies. In some cases, the cycles taken may be much fewer, due to the presence of a test that may result in early exit from a particular routine. For example, certain parts of the cache line assembly routine are bypassed entirely if there are no patches on the line.

System Tile

Table 5.1 shows the service times for a variety of routines. The table is divided into five sections: routines common to multiple handlers, routines under the static handler, routines under the MDN handler, routines under the GDN handler, and deadlock resolution routines.

The outstanding request and pending address tables are data structures that are used by several routines. The arrival of a first outstanding request takes slightly longer due to the need to contact an interrupt controller. The pending address table has service times proportional to A, the number of addresses it contains. Finally,

Routine	Time
trigger routine after a store	6
store registers	14
parse and reset event counter	6
parse triggering instruction for address and data	19
send an MDN message and wait for an acknowledgement	$18 + S$
parse acknowledgement and possibly invalidate or flush	16
parse subsequent addresses for address and data	36
restore registers and exit	16

Table 5.2: Cycle Times for the Components of the Interrupt 6 Handler

both the static handler and the GDN handler may note the need for a writeback request on an exclusive line.

The static network handler's assembly routine uses the pending address table, and thus the time taken by that routine is also proportional to A.

Interrupt Handlers on the User Tile

In this section, I discuss the time taken for the Interrupt 3 and Interrupt 6 handlers on the user tile. Each routine is executed in entirety once started, and is not context-swapped out. However, an Interrupt 6 routine may be interrupted by the arrival of Interrupt 3.

Interrupt 6 is triggered by a store instruction dirtying a cache line. The components of the handler are shown in Table 5.2. S represents the total time spent stalling, waiting for an acknowledgement from the directory. At best, this is 10 cycles, due strictly to network latency, but is likely to be much more. After any subsequent addresses (executed between the store that triggered the interrupt, and the interrupt handler invocation) are parsed, those are also sent via the MDN and acknowledged.

Interrupt 3 arrives due to an interrupt controller sending a message to a user tile. Separate stall times indicate that the routine must wait for the next step, either from a directory or an interrupt controller. The routines that talk to directories are invoked as many times as needed, since multiple directories may want to talk to a given user.

Routine	Time
store registers	12
send a ping to interrupt controller and wait	$3 + S$
handle a deadlock resolution mechanism	23
send a ping to a directory and wait	$7 + S$
handle a request (invalidate or flush)	16
send an acknowledgement to a directory	2
send an acknowledgement to the interrupt controller	2
restore registers and exit	14

Table 5.3: Cycle Times for the Components of the Interrupt 6 Handler

State	Load Time	Store Time
read	88	$328N + 174$
other	$328N + 536$	$328N + 696$

Table 5.4: Latencies of Service Calls in Various States

5.2.2 Service Times for Requests

In this section, I discuss the amount of time the system takes to service a variety of requests. From the perspective of the user, there are only two requests to be serviced: load and store. These may occur when the line is in a variety of states, and their total service time is equal to the sum of the times required by the various handlers, and also the network latencies.

Table 5.4 shows the amount of clock cycles taken to service these requests, provided that the system tile is not busy doing anything else. The value N in the table is the number of readers, which all must be sent invalidates in the case of a write.

For the purposes of this discussion, I divide the states into "read" and "other". All the non-read states will, upon receiving a load, return to "read" state, and upon a store, to either "read" or "exclusive". This means that the "other" state is at worst "read-lock". Those numbers are given. If the system is in any other state, the transaction will take less time, because some of the responsibilities will have already been handled.

These latency numbers are in the absence of other traffic, and multiple requests at the same time will take as much as the sum total of the individual request latencies. For example, if in the read state two processors issue a read request, then one processor

is serviced first: that one is expected to get the cache line within 88 clocks, but the other one will take 176.

However, due to certain requests being packed into single transmissions, the service times may be decreased in the presence of traffic compared to the worst-case sum total values.

The variance of these requests is based on both network latency variances (given the physical location of the requestor), and also the time required to load a particular handler on a system tile. Since the system tile loops through polling each of the networks, it may just miss an arriving call and get it only on the next loop. One network loop is 34 clocks long, so the expected time to get to a handler is 17 ± 17 clocks. Therefore, the total variance is equal to the sum of the variances of each network request and each different handler that must be invoked.

I now explain how each of those numbers is derived.

5.2.3 Read Requests

The simplest request is one of a new reading tile while in read mode. This boils down to fetching the line from RAM, forwarding it, and setting the state. This service time is independent of the number of tiles already reading, and is 88 cycles.

In the case of the line not being in read mode, then the worst case is when the read request arrives just as the line has been locked. In this case, the old readers must all be invalidated (328 cycles per reader), and then the writer must flush the line, which is then assembled and sent to any new readers that arrived in the interval between read-lock being set and assembled line, complete with patches, being sent back to main RAM. The flushing and assembly take 536 cycles.

5.2.4 Write Requests

In the case of a write request arriving in read mode, every reader must be sent an invalidate before the writer is acknowledged. After acknowledgement, the routine exits within four cycles. The length of an invalidation is the sum total of generating

the request, bouncing it to the tile via the interrupt controller, and receiving an acknowledgement. This is expected to take 328 cycles per reader.

If the write arrives in another mode, then it implies that two tiles have attempted to write so close to each other in time, that the first write was not fully processed, causing the second tile to request a read first. In this case, the second tile sees the non-read state and the writeback results in the second tile getting a negative-acknowledgement after 519 cycles. This number is large due to the presence of additional traffic on the network: this state transition is not possible unless multiple tiles write at once.

At this point, the first tile is asked to flush, and then the resulting cache line is assembled. This assembly is only allowed to take place once all readers have invalidated, so the total routine takes an extra $177 + 328N$ cycles, for a total latency of $328N + 696$ as shown in the table.

5.3 Application Results

Here, I describe an edge-detection application that was run to gather performance numbers on the shared memory system. The algorithm itself is Sobel's method[20], and there are actually three different implementations on Raw[21]: one using low-level routines that directly access the routing networks, another coded in C using the MPI message-passing library[40], and finally one using the shared memory system, hand-coded in assembly by me. I will first overview my approach to the problem, and then give the results.

5.3.1 Application Overview

Several design decisions were made in the context of the work done by the Raw research group as a whole. We decided on an edge-detection application because it may have some later use as a demo. Therefore, our specific implementations each take streaming data from off the chip, in the form of video frames, and then process it in one of several ways, sending the processed frames back off the chip.

The geographical layout of my implementation is shown in Figure 5-1. The input

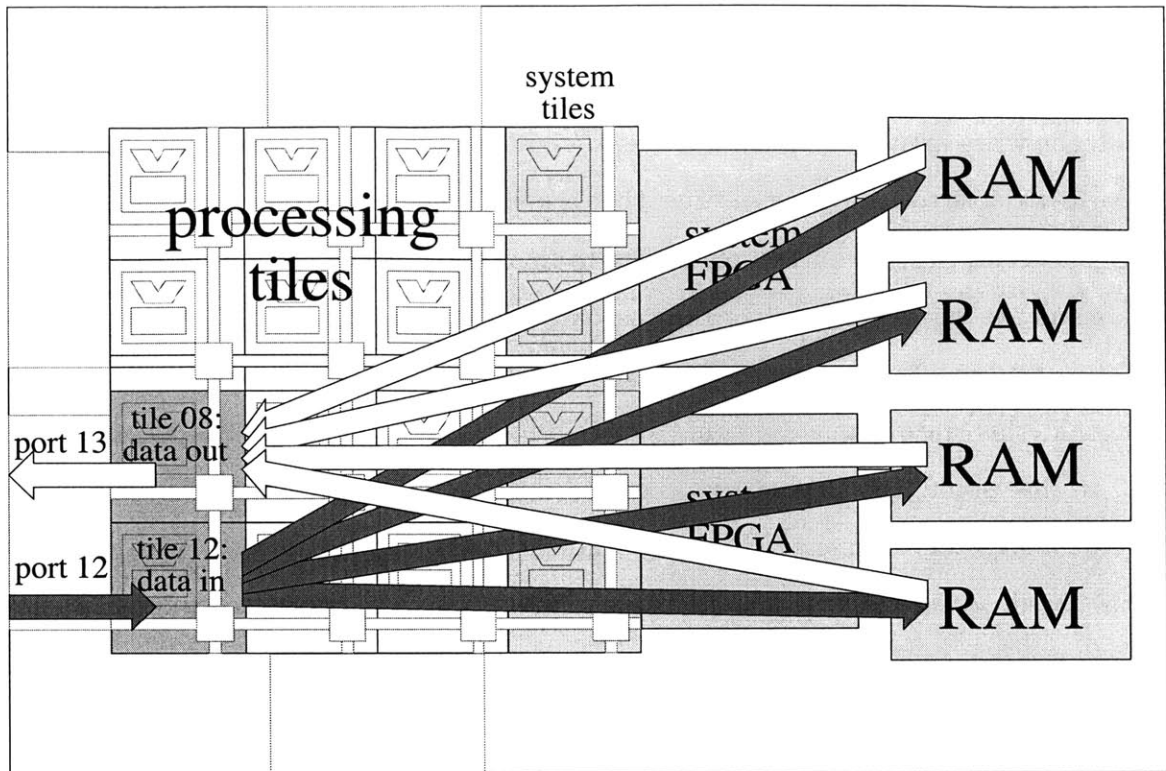


Figure 5-1: An Edge-Detection Application on the Shared Memory System

port is 12 and the output port is 13: frames arrive from port 12 and are immediately written to shared memory by tile 12, the *distribution tile.* Each frame is 320 pixels by 240, and each pixel is one word (24 bits of colour, and 8 unused). Each row of 320 pixels is fit into (for purposes of fast addressing) 512 words of main RAM. In the extra space, control words are placed as semaphores, which signal when a certain portion of each line is written, and thus may be processed.

The processing is done by ten user tiles. The processing consists of, for each pixel, running the Sobel[20] algorithm, which consists of two convolution matrices that, for all intents and purposes, take the derivative of the image in two dimensions. The magnitude of these gradients is then calculated, using an approximate square root[2]. This calculation is repeated for all three channels (red, green, and blue) and the resulting values[3] comprise the new pixel. The new pixel is stored to another

[2]The approximation to $\sqrt{x^2 + y^2}$ is actually $|x| + |y|$. We chose this path because it was simple, and while it would distort the calculations, it would do so uniformly across all the testing platforms.

[3]The resulting value is actually shifted right by three, because the total gradient is at most eight times the maximum pixel intensity. A factor of four results from the convolution matrix, and another

frame buffer in shared memory. When a processor is done with its 32 pixels of input, it clears the semaphore, letting the distribution tile know that the memory may be reused when the next frame comes along.

Each user tile is responsible for thirty-two pixels in each row. When it finishes its task, it sets a semaphore, which is monitored by tile 08, the *reassembly tile*. This tile reads the completed frame out of shared memory and streams it out off the Raw chip. The reassembly tile clears the semaphore when it has streamed out the data.

The entire application code, for all tiles, is in Appendix A. The comments in the code reveal all of the low-level details, including some encoding hacks to minimise collisions across cache lines.

I now analyse this program's performance.

5.4 Analysis of Results

The program did not run particularly quickly, due to certain weaknesses in the implementation. Given that one tile was reading from the static network, and another writing to it, there was a severe bottleneck, where these reading and writing tiles were constantly busy, but the actual processing tiles for the most part were not. The performance would have been very similar had there been only one processing tile, and not ten.

I now give the exact numerical results. First, I profile the amount of time that the tiles spend in a variety of states, showing that a significant portion of computation is relegated to either waiting on semaphores, or on doing system-level routines. I then I calculate an overall frame rate if the application were to be run on an actual Raw processor, and then analyse why the application is so slow.

All of the calculations are done in the steady-state loop of the program, in rows that are neither the first or the last. This comprises 238 of the 240 rows.

---

factor of two from the "square root" combination.

Activity	Cycles Spent
Processing	2528
Load word stalls	2006
Interrupt 3 handler	3925
Spinning on semaphores	752
Interrupt 6 handler	16670
Total	25881

Table 5.5: Time Spent on Various Activities

5.4.1 Profiling of User Tile Activity

As noted before, the program is bottlenecked by the distributor and the reassembler tiles. In one period of execution, the distributor tile writes one section of one row of pixels, intended for precisely one processing tile. In this same period, one of the processing tiles does some calculation and writes the result back to an output frame in memory, while the remaining tiles sit idle. Also in this period, the reassembler tile reads one section of pixels from memory and sends them back out of the system. These three operations overlap and at times are executed in parallel, but at other times, two processes are waiting on a system tile, which is servicing the third process.

I take as a typical example a processor tile, which is doing both loads and stores. The steady-state activity is shown in Figure 5.5. The total period is 17253 clock cycles, of which only 2528 (14.7 per cent) is doing useful work: namely, running the edge detection on data that is in registers. The rest of the time is spent either getting the data into the registers, or performing system tasks. It is important to note the large amount of time spent in the event counter interrupt. Due to the simulator restriction, all stores were followed by a jump, so there are 34 event counter handler calls in every period (32 data words, and two semaphores). It would have been possible to reduce this number to 6 separate cache lines (four data, two semaphore) using a more careful design. In that case, the period would be reduced by 28 fast "already exclusive" calls to the system tile, each of which is expected to take around 174 cycles. The period would be reduced to 12381 cycles, and the efficiency would increase to 20.4 per cent.

The distributor tile has a similar cycle time, except more time is spent waiting on

semaphores, due to the simplicity of the useful work. The reassembler does not store any words to the shared RAM, so therefore it spends even more time spinning on an interrupt controller. The efficiencies of those tiles are very low (on the order of 0.13 per cent), given that all they are doing is communicating with the static network, and there are only 32 words to be read or written (one clock cycle each) per period.

5.4.2 Overall Frame Rate

The total frame rate can be calculated as follows. There are ten periods per row, of which the first and last are slightly different due to border conditions, but the total difference is a handful of cycles. Then, there are 240 rows, of which 238 are as shown above, and the remaining two fundamentally similar, especially after the first frame. (The very first row of the very first frame goes slightly faster, due to the absence of certain data exclusively in remote caches: it is expected to take only about 8000[4] cycles per process period, as opposed to 17253.) The product of 240 times 10 times 17253 is 41 million cycles. If a Raw processor runs at 287 megahertz[48], then it can be expected to execute just about seven frames per second.

5.4.3 Why the Application is Slow

Several observations may be made at this point. A well-designed application, hand-coded for speed (for example, Patrick Griffin's implementation of the edge detection routine[21]) will be bottlenecked by the network streaming into Raw, as opposed to the processor itself. The network can support around 40 frames per second[25].

The application is slow for three reasons: it is not an ideal shared memory application, the system tiles are a bottleneck, and the user tiles spend far too much time processing overhead.

This application is not ideal for shared memory, as each data word is read by precisely two processes. There are two canonical producer-consumer pairs in the system: the input frame is produced by the distributor, and consumed by the processors. The

[4]The exact number is something like 7962, plus or minus a network delay.

output frame is produced by the processors, and consumed by the reassembler. It would be far simpler to use a streaming or a message-passing model, in which the data is simply sent from one point to the other. An application which involved much more all-to-all communication patterns, and need for broadcasts, would be better served by the shared memory model.

The system tiles are a bottleneck, because at least one is always running. Since all the memory is shared, the processor cannot do anything without relying on a system tile. I attempted to distribute the data between the four system tiles evenly, but it was not really possible to do so: two tiles were always busy, and two less so. One was handling the input frame, and one the output. A better distribution of addresses would have helped. It may be possible to gain better performance, at least in the context of this application, through the use of more system tiles. Perhaps four separate directories and four separate interrupt controllers (or maybe even eight directories, and four interrupt controllers in this case) would have helped matters. As a general solution, however, taking away that much user-level resources is probably not ideal.

Finally, the user tiles do not spend enough time running the user application. This is a disadvantage of a software-based memory controller: the software must coexist with the actual application, and the two naturally end up competing for resources; in this case time.

Chapter 6

Further Development

In this section, I address the possible further development of this shared memory system. I focus on three areas of research: design changes in the Raw architecture, implementation changes of the shared memory model when the topology changes from a 4x4 grid, and support for software development, including the standard OpenMP[38] interface.

6.1 Hardware Changes in Raw to Facilitate Shared Memory

This first section deals with the possibility of modifying the Raw architecture to make the system software more straightforward. Both reads and writes require a bit of cleverness to maintain correctness, and these are mostly due to hardware restrictions. I discuss three of these restrictions, and possible solutions.

6.1.1 Handling Cache Misses in Software

Currently, a cache miss is handled by an explicit hardware mechanism that generates an MDN message that is routed to the FPGA. In the shared memory implementation, it is necessary that messages be routed to the directory, and currently this is done by a workaround - the FPGA bounces the message to the directory via the static

network. The design would be cleaner if the hardware were programmable to allow cache miss messages to be directed to a particular location.

A more ambitious task would be to throw an exception upon a cache miss and allow for software to resolve the problem. This possibility was discussed by Michael Taylor[48], but the current Raw architecture does not have this interrupt possibility. The corresponding handler is a fundamentally difficult task, but there may be some good research to be had in its development. It would clear the problem of the network stalling hard (so much so that external interrupts are masked) on a cache miss, resulting in a deadlock.

6.1.2 Improved Awareness of Writes

The second complication deals with the functionality of the event counter. The event counter was never designed to support what it has been asked to handle by this system. It is merely a debugging hack. A possible improvement on it would be a mechanism that quickly fires an interrupt (within two cycles, not six) upon a store resulting in a dirty cache line. On one level, this violates Raw's design principles, since it is dedicated hardware, but on the other hand I cannot think of a way to detect a cache miss in software with no hardware support whatsoever. It may be possible to throw an interrupt after *every* store. In fact an alternate design - in which a store is immediately followed by a jump-and-link to a procedure - would not need any modification to the Raw hardware, and result in a very quick and atomic recognition of the event. This of course adds user responsibility.

6.1.3 Identification of Interrupt Sources

Currently, an external interrupt on Raw is an MDN message of length zero. Since headers are not forwarded to a reading processor, and are instead eaten by the final step of the network routing, the interrupt message itself is never seen by the processor. Instead, a bit is set in an interrupt vector. Unfortunately, this means that all interrupts carry one bit of information, leaving no room for the source, and thus

the shared memory system needs separate interrupt controller tiles. Things would be made a lot simpler if an external interrupt would be identifiable by source. One possibility is to have a special register be written to, with the source of the interrupt (as encoded into the MDN message), whenever the interrupt bit is set. This would allow the handler to read that special register, identify the source of the interrupt, and not have to worry about the intermediate steps in the protocol by which an interrupt controller forwards directory identity information.

The problem with this implementation is that messages arriving from a tile are indistinguishable from those arriving from outside the fabric, neighbouring the tile. There are no "initial routing" bits, as a processor and its off-chip neighbours are serviced by a single router. This is an implementation difficulty, and probably not fatal. Surely it is known at compile time where interrupts are expected to arrive from, and they may be distinguished given the design of a system.

6.2 Expanding to Larger Raw Processors

Raw is not exclusively a 4x4 tile with FPGAs surrounding it. That is merely the first implementation of the architecture, and other configurations are coming soon, including 8x8 and 32x32 tiles. It remains to be seen exactly how a memory interface will be placed into these implementations. I will discuss the ramifications of larger scales on the correctness of currently designed operations, as well as offer some ideas for how to scale the system effectively to maintain performance.

6.2.1 Necessity of More System Tiles

It is clear that if there are more user tiles, and more ports to memory, such as shown on the possible 64-tile grid in Figure 6-1, then more system tiles are needed. How many more, and what their functionality is, remains to be seen. I discuss these scaling issues for n by n grids, with a total of n^2 processors.

One may assume that traffic will increase with the number of processors, and also that the 3:1 ratio of user to system tiles is ideal. In this case, on a 64-tile grid, the

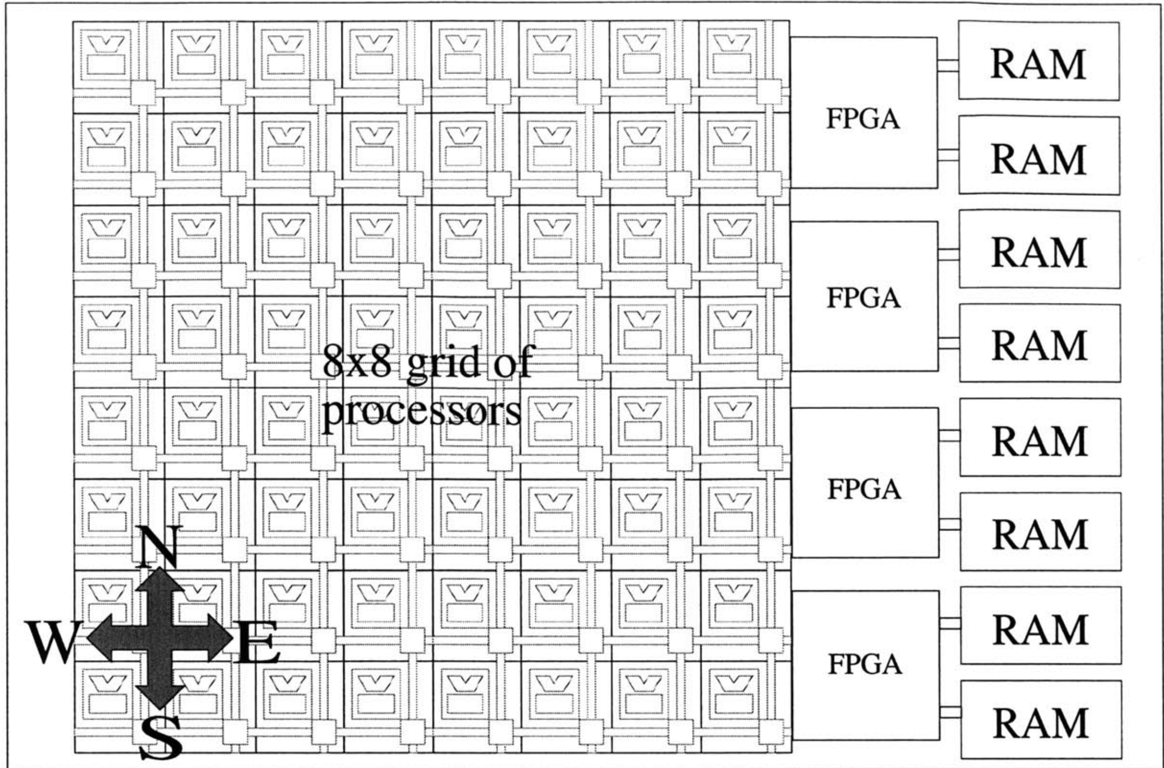


Figure 6-1: A Possible Design for an 8x8 Raw Processor

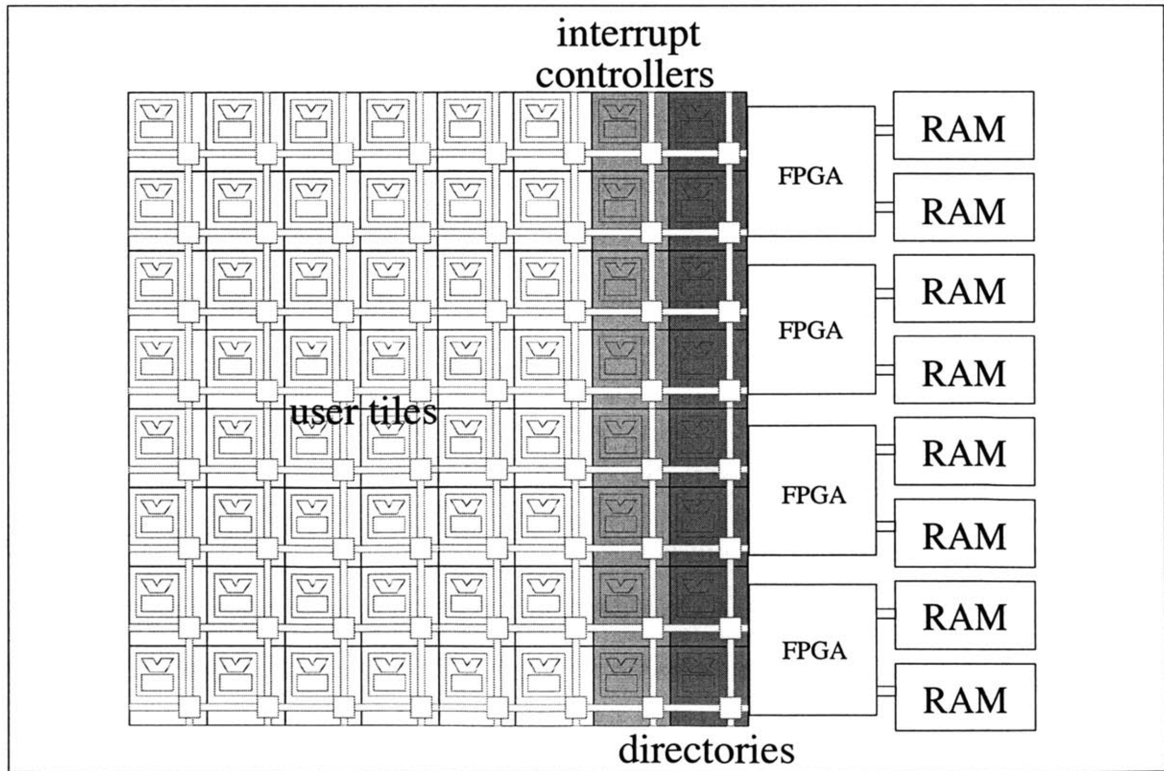


Figure 6-2: Division of Labour among System Tiles on an 8x8 Shared Memory System

rightmost two rows are system tiles. It may be reasonable to break the system tile into its two physical components, and have one column be all directories, and one all interrupt controllers. This possibility is shown in Figure 6-2. I will discuss in the next section why this must be done to maintain correctness for large grids.

Alternately, the traffic may increase with the number of possible processor-to-processor links, which implies that the number of system tiles is proportional to $O(n^2)$. A certain equilibrium is reached for large number of tiles: a constant number of columns are user tiles, and the remaining $O(n)$ columns are system tiles! This of course is a horrendous waste of resources, so a better solution must be devised at some point.

This problem is already a topic of research, both in the Raw group and elsewhere. Hoffmann *et. al.* [23] discuss parallel streaming algorithms on architectures in which the amount of RAM (and RAM management) is proportional not to $O(n^2)$, but rather to $O(n)$. Raw is an example of this form of architecture. Other architectures, for example Stanford's Smart Memories[34], implement RAM in equal proportion to processing power.

It remains to be seen how the shared memory system scales with larger grids. The correct answer probably does not have significant $O(n^2)$ components, given that the overall RAM bandwidth is $O(n)$. Therefore, having $O(n)$ tiles, probably along the edge of the processor grid, be dedicated to RAM management, is the way to go.

6.2.2 Routines that Fail for More than 16 Tiles

The low-level implementation details of the current system are very dependent on a 16-tile architecture. For example, the readers of a particular cache line are stored as a bit vector. Given at most 12 readers, the entire state of a cache line may be encoded in one 32-bit word[1]. However, as the number of bits increases, the algorithms for manipulating the state must be subtly redesigned to handle multi-word state vectors. Also, a lot of state is stored currently in registers that would need to be swapped out

[1] in fact, 18 bits are used at most: in the read-lock state, one needs two state bits, twelve old reader bits, and four bits to identify a future writer

to cache, decreasing the efficiency of the system.

Furthermore, the correctness of the system is heavily dependent on the number of system tiles being sufficiently small. Upon a flurry of directory-to-interrupt controller messages, the GDN does not back up simply because the maximum number of message words is less than the space in the network buffers. If more tiles are added, and the maximum size of communication increases as $O(n^2)$, while the buffer space only as $O(n)$, at a certain point the proof fails.

It is possible to remove the feedback problem if no tile can physically be both a directory and an interrupt controller: therefore, separating the two functionalities for larger grid sizes is a matter of correctness.

6.3 Support for Application Development

So far I have not explicitly mentioned the interface between the shared memory system and the software that is to run on it. It is not likely that the user will write the assembly language by hand, so therefore a compiler will be needed. I have discussed several restrictions on programs that must be run, and here I expand on that, discussing how shared memory management may look to a higher level language. First I overview some basics of shared memory from the application's perspective, describing some basic routines that would be part of a library. Then, I discuss one specific library implementation, OpenMP, showing that it will run on the system I have designed.

6.3.1 Compiler Support for Shared Memory

The keys for a compiler, above and beyond the standard low-level limitation avoidance discussed before, is to facilitate the correct use of shared memory, by providing certain primitives. These primitives include routines for memory allocation and process synchronisation.

Process Synchronisation

Even in the most ideal shared memory model, there is a need for synchronisation. This is done via having control objects in shared memory that are accessed in proper ways, thus acting as locks, semaphores, or other primitives. This is a known problem, with many known solutions. For example, the System V IPC specification[41] allows for many of these primitives to be run on a single-processor multiprocess system. It is easy to expand this design to multiprocessors.

Consistent Allocation of Shared Memory

Clearly it is not sufficient for a process that desires shared memory to simply call an allocation routine. Two parallel processes, issuing two calls to `malloc()` or the like, will likely get back two entirely different pointers. Therefore, one process must initiate the allocation, and the other must only request its address. This boils down to a problem of synchronisation.

A related problem is the generation of addresses that correctly reflect the nature of memory. On the Raw system, currently memory is identified as shared or private based only on certain address bits. This allows for very easy decoding of requests, but is inflexible as it is now: the system starts out with a predefined amount of shared RAM, and a predefined amount of private RAM. It may be possible to allow the FPGA to be reprogrammed dynamically to allow for a more flexible scheme. An easier, but somewhat more dangerous, solution may be to use the sub-word bits, already being used by the event counter, to identify shared memory. All requests for shared memory are already tagged, it just remains for the user not to forge a tag and self-destruct. Subword objects, such as bytes, would have to be handled very carefully. The compiler can be given several rules on how to implement a routine such as `malloc()`, and how to handle various data types.

6.3.2 Support for the OpenMP Library

The OpenMP interface[38] is a specification for shared memory support for C/C++ programs. It is important to note what OpenMP is not: it is not a compiler, and thus has no idea about the underlying system that it is implemented on. It can be viewed as a language extension: a compiler must be able to parse OpenMP primitives and turn them into correct machine code.

OpenMP allows for the possibility of mechanisms that resolve deadlocks, write-after-write hazards, and other conflicts. However, it does not automatically check for these problems. That remains the user's responsibility.

In this section, I do not describe explicitly how to build an OpenMP compliant compiler for the Raw architecture with my shared memory system on top of it, but I do show that it is mathematically possible. The proof has already been given in the chapter on correctness; it would be sufficient to say that since OpenMP works on the abstract shared memory system that has sequential consistency, it will work on mine.

I will demonstrate this in more detail, mirroring the OpenMP specification[38] and linking, where relevant, OpenMP mechanisms to their resolution on the shared memory system. These mechanisms take three forms: directives, library functions, and environment variables.

Directives

Directives are instructions to the compiler. They can be broken into three basic groups: access control, task assignment and task coordination. Access control calls, for example `threadprivate`, allow for private and shared memory to be identified correctly. So long as there is a possibility for both of these types, an OpenMP-compliant compiler will assign objects as is needed. For more fine-grain access control needs (for example, data being shared only among a certain subset of processors), it is possible to implement this functionality in software, as is done in Unix[41] for example.

Task assignment calls such as `parallel`, `for`, `sections`, and `single`, result in

the compiler generating code on specific subsets of processors. This code is more than likely generated statically and then spawned dynamically through the passing of appropriate branch conditions. The implementation is left up to the compiler, and the only requirement for success is that processes be allowed to communicate.

Already running code is coordinated through other directives, such as `critical`, `barrier`, and `atomic`. These routines spawn code that acts as a low-level primitive, such as a semaphore or a lock. These primitives are also available as library functions, so I will discuss them later.

One interesting task coordination directive is `flush`, which requires that a set of objects be flushed, such that all processes have a consistent view of it. OpenMP therefore facilitates the interface on shared memory systems that require user-level hooks[45, 11, 27, 52] for correct operation.

Library Functions

Library functions can be broken into two large categories: lock functions and task assignment helper functions[2].

Assignment helper functions are analogous to assignment directives. A routine such as `set_num_threads()` probably does not get past the compiler, and therefore needs only basic communication primitives in order to work correctly.

The other group of functions implements a variety of locks. These locks are all based on a simple lock primitive[13], which is possible given sequential consistency.

Environment Variables

Environment variables do not go beyond the level of task assignment primitives, and therefore do not need any further discussion. They are just a slightly different way of handling task assignment.

I have now shown that the OpenMP interface is supported by the shared memory model described in this thesis. Future work may include the implementation of this

[2]There are also some timing functions, but they are not required to be consistent across processors, so therefore I will not discuss them.

model, including a compiler that translates OpenMP-extended C or C++ to Raw shared memory code.

Chapter 7

Conclusion

A software-based memory controller is an ambitious task. This attempt at one is not fully implemented in system-level software, but it is still substantially different from most of the other schemes, with the only similarity being Alewife[9, 8].

In Figure 7-1, I add one more data point to the design space of cache coherence schemes. The design is within a margin-of-error approximation of the ideal all-system-software design, with the hardware hooks being simply some small FPGA modifications and event detectors, and the user-level software being only a small amount of restrictions that the programmer must keep an eye on. As a first attempt, on a platform that was not ever specifically designed with this goal in mind, the design is a success.

However, on an implementation level, the system shows many of the effects of migrating traditional hardware functionality to software. It is inherently not nearly as fast as dedicated hardware, with basic routine latencies measurable in the hundreds of cycles. Furthermore, these routines are not entirely parallelisable away from the user, and some of the time spent executing them is time taken away from the application.

A more refined implementation would attempt to place more responsibility into parallel components - in this case, the system tiles. This may entail reserving more tiles for system use, even in the 4x4 processor, or somehow getting better use out of the ones already there. This could be accomplished by several small revisions in the architecture itself, to make it less clumsy in supporting mechanisms that were not

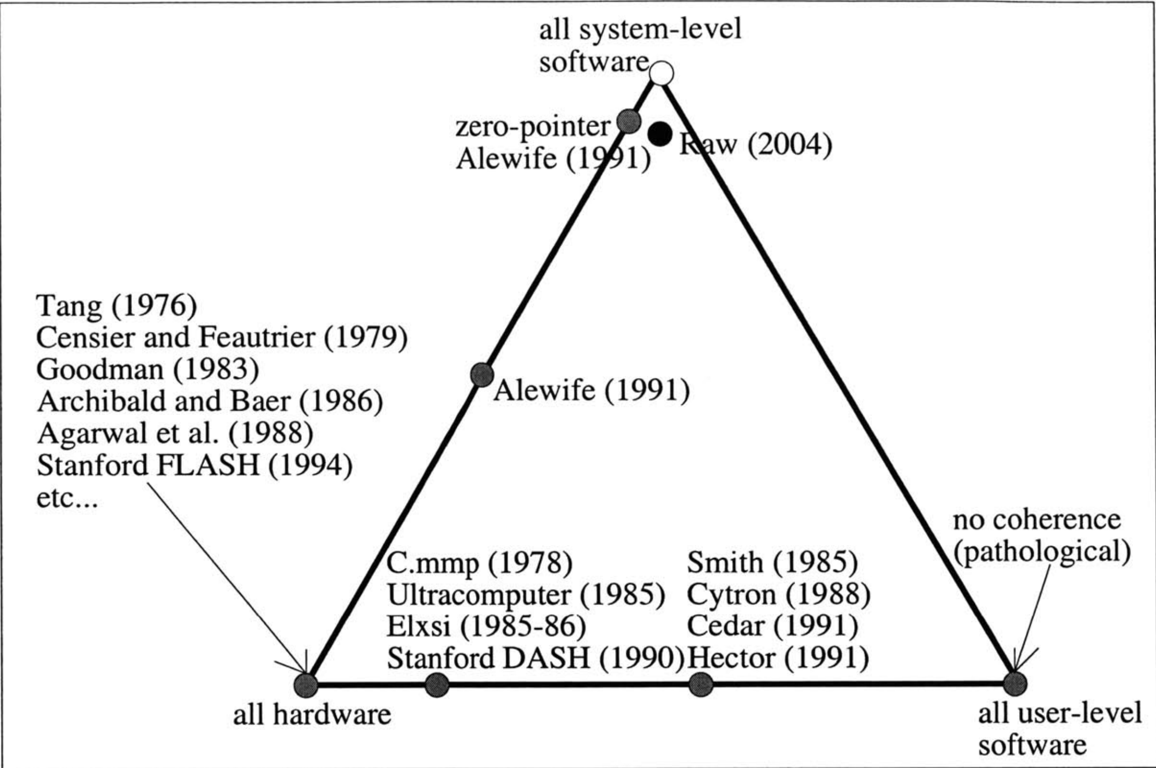


Figure 7-1: A Software-based Cache Coherence Scheme compared to Other Schemes

originally thought of when it was initially designed.

This thesis has shown the feasibility of the concept of software-based shared memory management on an exposed architecture. Comparing these results to other work on the Raw architecture, for example a software-based message-passing system[40] on the exact same hardware, demonstrates the flexibility that is a primary design goal of exposed architectures.

Appendix A

Raw Code

Here is the code that implements a shared-memory subsystem.

A.1 System Tile Code

This is the entire code that is loaded onto a system tile. It contains the necessary routines for managing both directory and interrupt responsibilities. Tile 3 is shown. Tiles 7, 11, and 15 are identical, with the exception of the first several lines of code (up to the first comment), which are dynamically generated constants unique to each tile identifying its location and RAM it uses to store state.

```
#define TILE_NUMBER 3
#define TILE_ROW 0
#define TILE_ONEHOT 1
#define BASE_ADDRESS 0x18000000
// directory tile.  Part of a shared-memory implementation.
// Levente Jakab
// 5/11/04 18.40

// number of cycles between an interrupt and a response at which point
// we declare deadlock.  Times four, because the cycle counts are always
// stored starting at bit 2.
#define DEADLOCK_TRIGGER_CYCLES -80000

#define DI_LIST_OP 0x300
```

```
// states:
// 0 - read (readers in bits 8-31)
// 2 - read lock (readers in 8-31, writer in 4-7)
// 3 - read lock bypass (readers in 8-31, writer in 4-7)
// 2 - exclusive (writer in 4-7)
// 3 - exclusive pending (writer in 4-7)

// read lock and exclusive are the same, except for the presence of
// reading tiles needing to be negotiated

#include "module_test.h"

.text
.align 2

.global begin
.ent begin

// register convention (above and beyond hardware restrictions)
// $2 is swap space... usually a tile number or an address
// $3 is swap space... an address in state space
// $4 is swap space
// $5 is swap space... a test condition
// $6 is swap space... an iterator
// $7 is swap space... a tile number
// $8 is swap space
// $9 is swap space... a tile number iterator

// $12 contains the constant 0xFF
// $13 is the static network stall condition
// $14 is one-hot directory number
// $15 contains the number "256"
// $16 contains a U-D map.  Which D-tile wants to speak to which U-tile.
// $17 contains an interrupt header constant
// $18 contains the offset of ocreq_table
// $20 contains the number "1"
// $21 contains the number "-1"
// $22 contains the BASE_ADDRESS
// $23 contains the offset of pending_address_table
// $28 contains the offset of pending_request_table
// $29 contains the number of pending addresses remaining

// $30 is a linkage pointer to one-away routines.  It is also swap space.
// Be careful!
```

```
// $31 is the link register.  Must be careful with this one.

// init routine.  Called exactly once per execution, so we don't care
// what regs we use.
init_begin:

// set up some things in RAM.  Address bit 31 is always zero.
// Address bits 30-27 identify the owner of the RAM.  Bits
// 26 and 25 define it as shared or not.

// if bits 26 and 25 of an address are both 1, then the RAM is
// shared.  Therefore, pages 6, 7, 14, 15.. 126, 127 (each page
// is 2^16 bytes here) are all filled with state space.  We jump
// from big table to big table (6/7, 14/15, ... 126/127),
// clearing out all 2^17 bytes of data stored there.

li $2, (8 << 16) // the big jump from pageset to
// pageset

li $3, BASE_ADDRESS // the beginning.

li $6, (6 << 16) // $6 has the address of the
addu $6, $3, $6 // first pageset (6/7)

li $4, (126 << 16) // $4 now has the address of the
addu $4, $3, $4 // final pageset (126/127)

reset_state_big_loop:
li $5, (1 << 17) // 2^17 addresses need to be
// cleared
addu $5, $4, $5 // starting with the current big
// jump start
reset_state_small_loop:

addiu $5, $5, -4 // subtract a word

sw $0, 0($5) // and store a zero

// if we just cleared out what is in address $4 (the beginning
// of the current big table) then we must fetch a new big
// table.
bne $5, $4, reset_state_small_loop

subu $4, $4, $2 // subtract big jump
```

```
// if this was the first pageset, we are done... else, we
// are not.
bne $6, $4, reset_state_big_loop

// write into pending address table the first free entry
la $2, pending_address_table
addiu $2, $2, 40

swsw $2, %lo(pending_address_table)($0)

// reset static network stall
or $13, $0, $0

// reset U-D map
or $16, $0, $0

// load constants
li $12, 0xFF
li $14, TILE_ONEHOT
li $15, 256
li $17, (0x00F00000 | (TILE_ROW << 3))
// constant meaning "interrupt"
// and the tile row, since each
// I-tile tracks only columns

li $20, 1
subu $21, $0, $20

// load addresses
la $18, ocreq_table
la $22, BASE_ADDRESS
la $23, pending_address_table
la $28, pending_request_table

li $29, 51 // 64, off-by-one, minus
// number of user tiles is the
// max number of pending
// addresses we can handle

j init_done



// static network routine.  Destroys every swap-space register.
```

```
static_network_begin:

// if we do not have any outstanding OCREQ space, then we must
// not attempt a static-network read.
bne $13, $0, static_network_done

// check to see if there is anything on static network
        mfsr $4, SW_BUF1 // pending incoming data

// note that we do not knock off any stray bits on the static
// network since everything on the static network is coming in
// from the east, so we save an instruction here.  Also, we may
// reduce latency by checking for an early message (one that
// just hit the switch), and by the time the BEQ is passed, it's
// viable.  Nifty.

        beq $4, $0, static_network_done // no data?  oh well, goodbye

or $10, $csti, $0 // read in a static word (header)

or $2, $csti, $0 // read in the address
srl $3, $2, 5
sll $2, $3, 5 // truncate low 5 bits to zero
sll $3, $3, 2 // for comparison purposes - $3
// has the short version, also
// needed.

addu $3, $22, $3 // cache line metadata address

lw $4, 0($3) // load the status of the line

// now, make the lw-sw distinction.
andi $6, $10, 0x8000

bne $0, $6, static_network_sw

// looks like this is a lw, so go forth and continue the lw

andi $6, $4, 3 // load the state bits

beq $6, $0, static_network_lw_read
// zero implies none or read, so therefore nonzero is one of the
// other states.

// we must be in some state that is not read.  In this case, the
```

```
// protocol is to request a writeback, if we have not done so
// already.  (This is signified by the low bit of the state.)

andi $6, $4, 1 // low bit is on?
bne $6, $0, static_network_lw_writeback_set

// we must request a writeback and also update the state as
// either going from read-lock to read-lock-bypass, or from
// exclusive to exclusive-pending.  This is done by setting the
// low bit.

ori $4, $4, 1 // set that bit
sw $4, 0($3) // and store back.

// if we are in the read-lock state, then the writeback will
// occur later, and we do not need to force it.

srl $6, $4, 8 // any readers implies read-lock
bne $6, $0, static_network_lw_writeback_set

// Set up a writeback request.  $2 contains the address.  $9
// must contain the tile number, which (given that we are not
// in "read") is in bits 4-7 of the state vector.

rrm $9, $4, 4, 0xF // tile number
li $11, 1 // request number 1 - coughup

or $30, $0, $31 // store the old linkage pointer
jal ocreq_begin // make function call
or $31, $0, $30 // and restore LP

static_network_lw_writeback_set:

// now, deal with the requestor.  Convert his number into some
// civilised form.

rrm $8, $10, 3, 0xC // $8 has yy00 of requestor
rrmi $8, $10, 0, 0x3 // $8 has yyxx (tile number)

or $6, $23, $0 // load the beginning of
// the pending-address table

// at this point
// $2 has the address in question
// $3 is its address in state space
```

```
// $4 is its state
// $6 has the beginning of the pending address table
// $8 has the requesting tile's number
// $5, $6, $7, $10, $11 are swap space

// here, what we do is check all the addresses in the pending
// address table.  If we have a match, then we go to that one.
// Otherwise, we dribble off the bottom and go to the next one.

swlw $9, 0($6) // this is the address of the
// first free entry in the table.

static_network_lw_address_loop:
addiu $6, $6, 40 // add 40 to this entry
swlw $7, 4($6) // load the address there

// exact match, bail out
beq $7, $2, static_network_lw_address_exact

bne $6, $9, static_network_lw_address_loop

// must be a new entry.  Where we are now, plus 40, is the next
// free entry.
addiu $7, $6, 40
swsw $7, 0($23) // store this new past-end
// pointer

swsw $2, 4($6) // and our current block must
// have the new address

swsw $0, 0($6) // clear out any possible
// detritus control bits.

static_network_lw_address_exact:

// at this point, $6 must contain a correct slot.

sllv $7, $15, $8 // requestor bit mask
swlw $5, 0($6)
or $7, $7, $5 // note new requestor

swsw $7, 0($6) // header spot = 1<<(requestor+8)

j static_network_done
```

```
static_network_lw_read:

// we are in the "read" state, and someone attempted an "lw", so
// all we really do is add him to the list of readers and give
// him the cache line.  $2 contains the address, $10 the header
// with the U-tile number.  $3 contains the cache-line metadata
// corresponding to this address, and $4 the data at $3.

// first, forge an MDN header so that we get the data from RAM.

// send to ourselves, but route funny to the east
li $cmno, CONSTRUCT_DYNAMIC_HEADER(4, 1, 0, (TILE_ROW >> 2), 3, (TILE_ROW >> 2)

or $cmno, $2, $0 // send the address

// and now, the MDN will send the line.  We must then forward
// it along.  First, prepare an MDN header.

or $cmno, $10, $0
// and now read in the data words, and pass them back out
.rept 8
or $cmno, $cmni, $0
.endr

// we have now sent the data to the new reader.  We have to
// add it to the list of readers.
rrm $8, $10, 3, 0xC // $8 has yy00 of requestor
rrmi $8, $10, 0, 0x3 // $8 has yyxx (tile number)

sllv $8, $15, $8 // shift "256" over by tile
// number
or $4, $4, $8 // set the bit
sw $4, 0($3) // and write it back

// and goodbye
j static_network_done

static_network_sw:

// this is what happens when we've gotten a store-word.  This
// could be in response to a coughup, or not.  In all possible
// cases, the data being stored is the freshest line around, so
// we should send it to any tiles stalling out waiting for the
// read.  Also, we should write it back to main RAM.
```

```
andi $8, $4, 1 // grab the low bit of the state
// to differentiate between
// pending and regular

// if not pending, then that means there was no possibility of
// extra writers, and no need to worry about patching.  Also,
// there are no outstanding readers, so no need to worry about
// that either.

beq $8, $0, static_network_sw_nopatch_noreq

// we must have either readers or patches.


or $6, $23, $0 // load the beginning of
// the pending-address table

// $2 contains the current transaction address
// $3 the address of that cache lines state
// $4 contains the entire state line
// $6 has the beginning of the pending address table
// $10 contains a MDN header corresponding to the writing tile
// $5, $6, $7, $8, $9, $11 are swap space

// We must find the entry in the pending address table
// corresponding to our address.

swlw $9, 0($6) // this is the address of the
// first free entry in the table.

static_network_sw_address_loop:
addiu $6, $6, 40 // add 40 to this entry
swlw $7, 4($6) // load the address there

// exact match, bail out
bne $7, $2, static_network_sw_address_loop

// note that the loop does not bail out unless it finds an
// exact match, because the match must be in the table, given
// the state.

// at this point, $6 must contain a correct slot.

swlw $4, 0($6) // address match, load header
```

```
andi $5, $4, 255 // load patch condition

addiu $9, $6, 8 // beginning of patch data
addiu $6, $6, 40 // point past end of patch data

// at this point, we have found the patch and reader data in
// the pending address table.
// $2 contains the address of all this trouble.
// $3 contains the cache line metadata's address
// $4 contains the patch/reader bits
// $5 contains only the patch bits
// $6 contains the end of the section in the patch table
// $9 contains the first patch line
// $10 contains a MDN header corresponding to the writing tile
// $7, $8, $11 swap space

static_network_sw_patchloop:

or $7, $csti, $0 // load the current data word

andi $8, $5, 1 // check current patch bit
srl $5, $5, 1 // and shift down

bne $8, $0, static_network_sw_patchloop_patched

// this word is not patched, so the latest version is coming
// off the static network.  Store it to memory.
swsw $7, 0($9)

static_network_sw_patchloop_patched:

addiu $9, $9, 4 // next patch point
bne $6, $9, static_network_sw_patchloop

addiu $6, $6, -40 // $6 has a little before the
// first point in cache line.

// at this point, the correct cache line is stored in the
// directory.  We must write it to its correct location in
// RAM.

// The actual address is in $2.

andi $5, $4, 255 // load patch condition
// we only monitor addresses that are patched, because there
```

```
// is a finite number of readers, so if this line was free of
// patches it was not accounted for, and thus does not need to
// be freed explicitly.

beq $5, $0, static_network_sw_notpatched

// But if it was, free it.
addiu $29, $29, 1 // one more free address

static_network_sw_notpatched:

// send an MDN message to the FPGA with the new cache line.

// send to ourselves, but route funny to the east
li $cmno, CONSTRUCT_DYNAMIC_HEADER(4, 9, 4, (TILE_ROW >> 2), 3, (TILE_ROW >> 2),

or $cmno, $2, $0 // send the address

swlw $cmno, 8($6) // amd write the data
swlw $cmno, 12($6)
swlw $cmno, 16($6)
swlw $cmno, 20($6)
swlw $cmno, 24($6)
swlw $cmno, 28($6)
swlw $cmno, 32($6)
swlw $cmno, 36($6)

// here, we have taken care of sending the patched data back
// to main RAM.  Now, we must send it to any possible
// recipients.
// $2 contains the address of all this trouble
// $3 contains the cache line metadata's address
// $4 contains the patch/reader bits
// $6 contains the beginning of the section in the address table
// $10 contains a MDN header corresponding to the writing tile
// $5, $7, $8, $9, $11 swap space

srl $5, $4, 8 // waiting tiles

// if no tiles, then exit
beq $5, $0, static_network_sw_senddone

sll $9, $20, 27 // size = 8

addiu $7, $0, -29 // current tile.  -29 because
```

```
// we will add 1, and then 28
// more as the first part of
// the iterator.
static_network_sw_sendloop:

addiu $7, $7, 1 // increment tile number
andi $11, $7, 3 // tile multiple of 4?

bne $11, $0, static_network_sw_noadjust

addiu $7, $7, 28 // 0, 1, 2, 3, 32, 33, 34, ...

static_network_sw_noadjust:

andi $8, $5, 1 // is this tile waiting?
srl $5, $5, 1
beq $8, $0, static_network_sw_sendloop

// looks like this tile is waiting on data... send it along.
// the header is the current tile number plus "size 8".
addu $cmno, $7, $9

swlw $cmno, 8($6) // and write the data
swlw $cmno, 12($6)
swlw $cmno, 16($6)
swlw $cmno, 20($6)
swlw $cmno, 24($6)
swlw $cmno, 28($6)
swlw $cmno, 32($6)
swlw $cmno, 36($6)

// more tiles requesting?
bne $5, $0, static_network_sw_sendloop

static_network_sw_senddone:

// here, we are done with this address.  So what we must do
// is clear it.  If it is the last address, then we simply
// let it fall off the bottom.  If not, we take the last entry,
// and move it into the entry being cleared.  In all cases,
// we decrement the next-empty-spot pointer.

swlw $8, 0($23) // find the last address
addiu $8, $8, -40 // subtract 40 because there
// is one fewer address.
```

```
swsw $8, 0($23) // store it back

beq $8, $6, static_network_sw_last

// must not have been the last entry.  So therefore, take what
// is the last entry (*$8) and move it to here (*$6).  Now
// the last entry is truly blank and the pointer $8 is correct.

// yes, this code is a brute, but I invite you to come up with
// a better memcpy call!

swlw $5, 0($8)
swsw $5, 0($6)
swlw $5, 4($8)
swsw $5, 4($6)
swlw $5, 8($8)
swsw $5, 8($6)
swlw $5, 12($8)
swsw $5, 12($6)
swlw $5, 16($8)
swsw $5, 16($6)
swlw $5, 20($8)
swsw $5, 20($6)
swlw $5, 24($8)
swsw $5, 24($6)
swlw $5, 28($8)
swsw $5, 28($6)
swlw $5, 32($8)
swsw $5, 32($6)
swlw $5, 36($8)
swsw $5, 36($6)

static_network_sw_last:

rrm $8, $10, 3, 0xC // $8 has yy00 of writer
rrmi $8, $10, 0, 0x3 // $8 has yyxx (tile number)

// now, write the new state, which is "read" on all the tiles
// we just sent data to (list is in $4), and also the writing
// tile.

sllv $6, $15, $8 // 1 << (writingtile + 8)

andi $5, $4, 255 // patch bits
```

```
subu $4, $4, $5 // all but patch bits

or $6, $4, $6 // new state

sw $6, 0($3) // store new state in state space

j static_network_done

static_network_sw_nopatch_noreq:

// this is a bypass in which we simply write the line back
// to main RAM, and set the reader as the old writer, and
// go to the read state.

// $2 contains the address of all this trouble.
// $3 contains the cache line metadata's address
// $4, $5, $6, $7, $8, $9, $10, $11 swap space

// send to ourselves, but route funny to the east
li $cmno, CONSTRUCT_DYNAMIC_HEADER(4, 9, 8, (TILE_ROW >> 2), 3, (TILE_ROW >> 2),

or $cmno, $2, $0 // send the address

or $cmno, $0, $csti // forward 8 data words
or $cmno, $0, $csti
or $cmno, $0, $csti
or $cmno, $0, $csti
or $cmno, $0, $csti
or $cmno, $0, $csti
or $cmno, $0, $csti
or $cmno, $0, $csti

// now, store the new state (read on this tile)
rrm $8, $10, 3, 0xC // $8 has yy00 of writer
rrmi $8, $10, 0, 0x3 // $8 has yyxx (tile number)

sllv $4, $15, $8 // 256 << this number
sw $4, 0($3) // store this state, which is
// read on this one tile
// only.

static_network_done:
jr $31
```

```
// pending IRQ handler.  Destroys all swap space.
pending_irq_begin:

or $3, $16, $0 // load the U-D buffer
or $8, $0, $0 // current U-tile number (times
// four)

pending_irq_u_loop:

andi $4, $3, 0xFF // current U-tile requests

// if there is nothing for this U-tile, then skip attempting to
// process it, and just go to the next one.
beq $4, $0, pending_irq_next_u

ilw $6, %lo(int_status_0)($8) // load the tile state from
// instruction memory.

andi $6, $6, 3 // get only the state bits

bne $6, $0, pending_irq_next_u // tile busy.  Go away.  Either
// we just interrupted it, or we
// are communicating with it via
// MDN, so things are happening.

pending_irq_do_interrupt:

// tile may be interrupted, so lets interrupt it.
mfsr $6, CYCLE_LO // grab cycle count
sll $6, $6, 2 // shift over
addiu $6, $6, 2 // and add state bits (state=2 is
// "interrupt pending")

isw $6, %lo(int_status_0)($8) // store new state

srl $4, $8, 2 // shift tile-number into place

// $17 is a pre-made interrupt header, waiting for a tile number.
or $cmno, $17, $4 // instant interrupt, booya

pending_irq_next_u:
addiu $8, $8, 4 // increment tile number being
// handled

srl $3, $3, 8 // next U-tile is pending
```

```
bne $3, $0, pending_irq_u_loop // loop if there are more
// requests

pending_irq_done:
jr $31


// gdn message handler.  This takes requests from user tiles to either
// D-tile or I-tile functionality of the directory tile.
// Destroys all swap space
gdn_handler_begin:

// check to see if there is anything on the GDN
        mfsr $4, GDN_BUF // pending incoming data
rrm $4, $4, 5, 0x7 // knock off "out" and other crap

        beq $4, $0, gdn_handler_done // no data?  oh well, goodbye

or $4, $cgni, $0 // read in a memory word

andi $6, $4, 0xF00 // get the request type
andi $7, $4, 0x7F // get the tile number (in
// somewhat wrong GDN form)
// into $7 (this is kept a while)

// $$$ make request number equal jump length and do a jr.

// request types are:
// 0x000 - "okay, done" from U-tile to I-tile
// 0x100 - "yes?" from U-tile to I-tile
// 0x200 - "yes?" from U-tile to D-tile ("hello")
// 0x300 - list from D-tile to I-tile ("call this tile")
// 0x400 - user to directory acknowledgement

beq $6, $0, gdn_handler_ui_done

addiu $6, $6, -0x100 // 0x100 equals zero now
beq $6, $0, gdn_handler_ui_call

addiu $6, $6, -0x100 // 0x200 equals zero now
beq $6, $0, gdn_handler_ud_call

addiu $6, $6, -0x100 // 0x300 equals zero now
beq $6, $0, gdn_handler_di_list
```

```
// default, must be a user-to-directory acknowledgement.  In
// this case, all of the addresses that needed to be invalidated
// were invalidated, and we can clear some bits from their
// state.

// from the tile number in YY000XX form, get a pending request
// begin address, which is YYXX0000000.
rlm $6, $7, 4, 0x600 // YY000000000
rlmi $6, $7, 7, 0x1FF // YYXX0000000

// generate a mask that has all bits set, except the bit
// corresponding to our tile number, so that we can OR it with
// a state and mask out acked readers.
srl $7, $6, 7 // this is the requesting
// tile number

sllv $7, $15, $7 // 1 << (this number + 8)
xor $7, $21, $7 // all bits but this one

addu $5, $6, $28 // load beginning of pending
// requests

gdn_handler_ud_nextaddress:
swlw $2, 0($5) // load address
addiu $5, $5, 4 // and increment pointer

// if this is the null-terminator then we have no more actual
// addresses.
beq $2, $0, gdn_handler_done

// otherwise, $2 contains an address.

srl $3, $2, 3 // address shifted right
addu $3, $22, $3 // cache line metadata address

lw $4, 0($3) // load the status of the line
and $4, $4, $7 // mask out the acknowledging
// tile
sw $4, 0($3) // store it back

srl $8, $4, 8 // get only a list of readers
bne $8, $0, gdn_handler_ud_nextaddress

// looks like that was the last reader.  Meaning, we are in
// a new state (exclusive or excl-pending) and thus must
```

```
// send an ack to the writing tile via the GDN.  First, generate
// an MDN header to send back to the writing tile.  Its address
// is in the form YYXX0000 in the state and we must convert it
// to 0YY000XX.

rrm $9, $4, 1, 0x60 // 0YY00000
rrmi $9, $4, 4, 0x3 // 0YY000XX

sll $11, $20, 24 // $11 = 1 << 24
addu $cmno, $11, $9 // header... message is of
// length 1

// now, send the ack - which is either a plain ack (0) or an
// ack-and-flush (1).  This corresponds precisely (I am so
// clever) with the low bit of the state.

andi $cmno, $4, 1

j gdn_handler_ud_nextaddress

gdn_handler_ui_done:

rlm $4, $7, 2, 0xC // tile number's horizontal
// offset, times four

// this is the correct offset in the interrupt status table.

isw $0, %lo(int_status_0)($4) // store a zero, state of "not
// in interrupt" because it just
// finished talking.
j gdn_handler_done

gdn_handler_ui_call:

// U-I communication ("who was that???")

sll $4, $20, 24 // $4 = 1 << 24
addu $cgno, $4, $7 // header... message is of
// length 1

or $cgno, $16, $0 // send along the big list of
// requests (let user process
// it!)
// this includes the possible
// "you've been spoofed!" bit.
```

```
rlm $4, $7, 2, 0xC // find tile numbers horiz offset
// times four.  We now have the
// correct address in interrupt
// status table land.

sll $8, $4, 1 // tile number times 8 (needed
// later).

isw $20, %lo(int_status_0)($4) // store a "1" to the location,
// meaning that a phone-call was
// successful so the tile is now
// in interrupt.

sllv $3, $12, $8 // shift over constant 0xFF
// ($12) appropriately

xor $2, $3, $21 // generate mask
// ($21 = 0xFFFFFFFF save 2
// instrs)

and $16, $2, $16 // and zero out just-sent
// requests

j gdn_handler_done

gdn_handler_ud_call:

// U just pinged D (the important phone call), so send along a
// list of requests.  For each of these requests, if it is an
// invalidate, write it to the pending (already sent) request
// table.

// from the tile number in YY000XX form, get an OCREQ begin
// address, which is YYXX0000000.
rlm $6, $7, 4, 0x600 // YY000000000
rlmi $6, $7, 7, 0x1FF // YYXX0000000

srl $2, $6, 7 // tile number we are dealing
// with (needed later)

addu $4, $6, $18 // load beginning of OCREQs for
// this particular tile

addu $5, $6, $28 // and list of pending requests
```

```
// which we will be storing
// into.

swlw $3, 0($4) // number of requests for
// this tile.

srl $3, $3, 2 // shift right to get rid of
// off-by-four
swsw $0, 0($4) // and reset that counter

addiu $9, $3, 1 // message length is the number
// of requests plus one, so
// shift that
sll $9, $9, 24 // over 24 slots so it is in the
// right place in the header.

or $cgno, $9, $7 // add tile number to message
// length, voila header
or $cgno, $3, 0 // send along number of requests
// coming

gdn_handler_ud_loop:

addiu $4, $4, 4 // next slot

addiu $3, $3, -1 // decrement number of requests
// remaining
swlw $9, 0($4) // load the actual request

or $cgno, $9, $0 // and send it along

// here, check to see if the request is an invalidate or a
// flush.  If it's an invalidate (ends in 2, as opposed to in 1)
// then we must store it in the outstanding buffer space.

andi $10, $9, 1 // low bit
bne $10, $0, gdn_handler_ud_flush

// must be an invalidate.
swsw $9, 0($5) // store the address

addiu $5, $5, 4 // next slot in pending
// request table

gdn_handler_ud_flush:
```

```
bne $3, $0, gdn_handler_ud_loop // more UD OCREQs to deal with?

// null-terminate the list of pending addresses.  Since 0 is
// not a possible shared address, it will never exist
// legitimately in this table, and thus works as a terminator.
swsw $0, 0($5)

// is there a static-network stall due to OCREQ backlog?  Well,
// we just sent some OCREQs so maybe we can clear it.
beq $13, $0, gdn_handler_done

// $2 is the tile we just processed
// $9 is a blank slot

sllv $9, $20, $2 // 1 << tile number
xor $9, $21, $9 // invert (all "1"s except bit
// at tile# is a zero)
and $13, $13, $9 // knock off a bit

j gdn_handler_done

gdn_handler_di_list:

// a list from a D-tile to an I-tile that the D-tile expects can
// help it.  The message contains the D-tile's one-hot index and
// the U-tile it wishes to communicate with.

or $4, $cgni, $0 // read in a GDN word (UUDDDD)

rrm $2, $4, 1, 0x18 // shift by 1 (UUDDD) and get the
// column (UU000).
andi $3, $4, 0xF // get request numbers of dirtile
// (00DDDD)
sllv $2, $3, $2 // shift dirtile and header
// correctly.
// (00DDDD << UU000)
or $16, $2, $16 // mask in the new request

gdn_handler_done:

jr $31


// mdn message handler.  This takes unexpected event counter events from
```

```
// U-tile to D-tile.
// Destroys all swap space
mdn_handler_begin:

// if we do not have any outstanding OCREQ space, then we must
// not attempt an MDN read
bne $13, $0, mdn_handler_done

// same thing with no outstanding address space
bne $29, $0, mdn_handler_done

// check to see if there is anything on the MDN
        mfsr $4, MDN_BUF // pending incoming data
rrm $4, $4, 5, 0x7 // knock off "out" and other crap

        beq $4, $0, mdn_handler_done // no data?  oh well, goodbye

or $7, $cmni, $0 // read in a memory word
// this is the tile number in
// YY000XX form (and it also
// has a MDN return handler -
// self addressed stamped
// envelope!)

// an event counter event.  This implies that a cache line went
// exclusive.
or $2, $cmni, $0 // read in the address
or $10, $cmni, $0 // and the data

rrm $8, $7, 3, 0xC // $8 has yy00 of writer
rrmi $8, $7, 0, 0x3 // $8 has yyxx (tile number)

// $2 is an address.
// $7 has an MDN header
// $8 is an offending tile (YYXX)
// $10 is a data word
// $3, $4, $5, $6, $9, $11 free

srl $3, $2, 3 // address shifted right
addu $3, $22, $3 // cache line metadata address

andi $5, $2, 0x1C // low bits of address (which
// part of the cache line is
// getting modified)
```

```
srl $4, $2, 5 // address, high bits only
sll $2, $4, 5

lw $4, 0($3) // load the status of the line

andi $11, $4, 7 // grab state number

// here, if the state is "read" then we do one thing, and if
// it isn't, another.

beq $11, $0, mdn_handler_read

// must not be read.  Check to see if the writer is the owner.

rrm $11, $4, 4, 0xF // tile number only (YYXX)

// if the writer is the owner, then we cannot send it an
// invalidate because then it would lose precious data!
bne $11, $8, mdn_handler_noclash

or $cmno, $0, $7 // send the ack.
or $cmno, $0, $0

j mdn_handler_done

mdn_handler_noclash:

// In this case, the state is not read, and the writer is not
// the owner, so we must record its data in the patch table,
// and send it an invalidate request.

or $cmno, $0, $7 // send the ack.
ori $cmno, $0, 2 // message is "invalidate"

// $2 contains the address
// $3 contains its state address
// $4 the actual state
// $5 the word of the cache line
// $10 is a data word
// $6, $7, $9, $11 free

// First, update the state and get a writeback if needed.

andi $6, $4, 1 // low bit is on?
bne $6, $0, mdn_handler_writeback_set
```

```
// we must request a writeback and also update the state as
// either going from read-lock to read-lock-bypass, or from
// exclusive to exclusive-pending.  This is done by setting the
// low bit.

ori $4, $4, 1 // set that bit
sw $4, 0($3) // and store back.

// if we are in the read-lock state, then the writeback will
// occur later, and we do not need to force it.

sll $6, $4, 8 // any readers implies read-lock
bne $6, $0, mdn_handler_writeback_set

// Set up a writeback request.  $2 contains the address.  $9
// must contain the tile number, which (given that we are not
// in "read") is in bits 4-7 of the state vector.

rrm $9, $4, 4, 0xF // tile number
li $11, 1 // request number 1 - coughup

or $30, $0, $31 // store the old linkage pointer
jal ocreq_begin // make function call
or $31, $0, $30 // and restore LP

mdn_handler_writeback_set:

// Now that that is out of the way, we record the patch.
// $2 and $5 have the full address, $10 the data.

or $6, $23, $0 // load the beginning of
// the pending-address table

// here, what we do is check all the addresses in the pending
// address table.  If we have a match, then we go to that one.
// Otherwise, we dribble off the bottom and go to the next one.

swlw $9, 0($6) // this is the address of the
// first free entry in the table.

mdn_handler_address_loop:
addiu $6, $6, 40 // add 40 to this entry
swlw $7, 4($6) // load the address there
```

```
// exact match, bail out
beq $7, $2, mdn_handler_address_exact

bne $6, $9, mdn_handler_address_loop

// must be a new entry.  Where we are now, plus 40, is the next
// free entry.
addiu $7, $6, 40
swsw $7, 0($23) // store this new past-end
// pointer

swsw $2, 4($6) // and our current block must
// have the new address

swsw $0, 0($6) // clear out any possible
// detritus control bits.

addiu $29, $29, -1 // one more patch slot used

mdn_handler_address_exact:

// at this point...
// $2 contains the transactions address
// $6 an offset in the pending address table
// $5 an offset to a word in the cache line
// $10 the data word
// $3, $4, $7, $8, $9 free

addu $9, $6, $5 // add the cache subpart to
// get the correct offset

swsw $10, 8($9) // and store the data word
// so kindly provided by the
// writing tile.  "8" is because
// the first two words in the
// block are header.

srl $8, $8, 2 // number of subpart
sllv $8, $20, $8 // one-hot encoding

swlw $9, 0($6) // grab header
or $9, $8, $9 // mask in bit

swsw $9, 0($6) // and store new header
```

```
j mdn_handler_done

mdn_handler_read:

// $4 contains the state.  In order to check for extra readers
// (defined as those that are not requesting exclusive access),
// the state word is manipulated to pull out the requesting
// tile, and this word is compared to zero.

// If there are no other readers, then our life is simple, and
// we set the tile to be exclusive ($9 has a tile number) and
// send the ack.  Otherwise, we go into read lock.

// $3 has the address of the state
// $4 has the state
// $7 has a return envelope
// $8 has the offending tile

sllv $6, $15, $8 // 1 << offender
xor $6, $21, $6 // zero in that position, ones
// everywhere else
and $4, $4, $6 // modified state line

bne $4, $0, mdn_handler_read_lock

// looks like the requestor was the only reader.

// set state to exclusive.
sll $9, $8, 4 // move to proper position
ori $6, $9, 2 // state is exclusive on the tile

sw $6, 0($3)

// $7 has the first word sent by the tile to the directory, and
// it is a return-addressed MDN header with its number as the
// recipient, and the length 1.

or $cmno, $7, $0 // send back the ack
or $cmno, $0, $0 // which is zero for "okay"

// that's all, folks.
j mdn_handler_done

mdn_handler_read_lock:
```

```
// this is the more complicated case.  We have readers.
// Therefore, we have to set the state to read lock, on the
// extant readers.  We do not set it on the new writer, because
// we cannot send the new writer an invalidate (it's got the
// fresh data!).

sll $9, $8, 4 // put in the correct position
ori $9, $9, 2 // set the read-lock state
// bit.

or $4, $9, $4 // and OR in the writer and
// lock bit

sw $4, 0($3) // store new state

// We must send a bunch of tiles an invalidate message, as a
// newer version of the cache line was just found.

srl $3, $4, 8 // list of readers

or $9, $0, $0 // current potential tile number
// (start at zero)
li $11, 2 // request 2 - invalidate

mdn_handler_read_loop:

// this is the writing tile.  It has the latest version of
// the cache line.  Can't send it an invalidate or we lose
// data!

beq $9, $8, mdn_handler_read_done

andi $6, $3, 1 // low-bit mask

// this tile is not on the list... bypass the call
beq $6, $0, mdn_handler_read_done

// make up an invalidate request for this tile, since it is on
// the list.  $2 contains the address to be invalidated, $11 has
// constant "2" because we are asking for an invalidate, and $9
// is the tile number.  (three params to OCREQ call)

or $30, $0, $31 // store the old linkage pointer
jal ocreq_begin // make function call
or $31, $0, $30 // and restore LP
```

```
mdn_handler_read_done:
srl $3, $3, 1 // knock off a tile
addiu $9, $9, 1 // keep two counters synced

// once the list is empty, it will be all zeroes so we must
// continue if $3 is not zero, meaning more tiles need an
// invalidate
bne $3, $0, mdn_handler_read_loop

mdn_handler_done:

jr $31


// deadlock detector routine

deadlock_detector_begin:

// here, we detect possible deadlock by seeing if a tile that
// got pinged N cycles ago still has not responded.  If this is
// the case, we set up a special force condition, and also send
// along a fake cache line.

li $8, 8 // load this constant; it is
// the tile number we are
// servicing (times four).  An
// iterator.

deadlock_detector_loop:

ilw $6, %lo(int_status_0)($8) // load the tile state

andi $4, $6, 2 // get only the high state bit,
// which is set only when state
// is 2 (there is no state 3)

// if not in state 2 (pinged but no response yet), we are not
// interested.
beq $4, $0, deadlock_detector_next_u

// check to make sure we haven't already sent the forcer, as
// we want to do this precisely once!

sll $4, $8, 1 // tile number (times 8)
```

```
li $2, 0x10 // bit 4
sllv $2, $4, $2 // shift over by 8 times the
// tile number - this is a
// forcer bit.

and $11, $2, $16 // check this forcer bit.  If
// $5 is one, then we've already
// sent a forcer.
bne $11, $0, deadlock_detector_next_u

// check to see how long ago we've pinged.  This is in $6 (with a
// footer of "10" but oh well)

mfsr $7, CYCLE_LO // grab cycle count
sll $7, $7, 2 // shift over

subu $7, $6 // subtract new time minus old
// time, giving the number of
// elapsed cycles since the last
// ping

// subtract off the threshold
li $11, DEADLOCK_TRIGGER_CYCLES
subu $7, $7, $11

// if this number is less than zero, then we have not exceeded
// the number of cycles and all is well.
bltz $7, deadlock_detector_next_u

// oops, looks like we've exceeded our timing.
sll $4, $8, 1 // tile number (times 8)

li $2, 0x10 // bit 4
sllv $2, $4, $2 // shift over by 8 times the
// tile number - this is a
// forcer bit.

or $16, $2, $16 // set this bit.  The interrupt
// forcer is set.  This bit will
// be received by the tile and
// it knows to deal with the
// force.

srl $11, $8, 2 // number of tile (times 1)
```

```
sll $10, $20, 27 // message of length 8
addiu $10, $10, (TILE_ROW << 3) // correct route

// Here comes the forcer.
or $cmno, $11, $10 // header going out (size 8)

.rept 8
ori $cmno, $0, 666 // and 8 bogus words
.endr

deadlock_detector_next_u:
addiu $8, $8, -4 // iterate
bne $8, $0, deadlock_detector_loop

deadlock_detector_done:
jr $31


// OCREQ handler subroutine.

// three args are passed in

// $2 has the address we are requesting
// $9 has the owner tile that we are doing the requesting for
// $11 has the request number (1 for cough-up, 0 for invalidate)

// $13 is the OCREQ overflow vector (global variable)
// $14 is the d-tile one-hot number (global constant)
// $18 is the OCREQ table begin address (global constant)
// $20 is "1" (global constant)

// $3, $5, $9 and $10 must not be written, as a caller needs those.
// $30 is a linkage pointer one stack-frame away, so must not be killed

// $4, $6, $7, $8 are destroyed

ocreq_begin:
// the OCREQ consists of a message to an I-tile, asking the
// I-tile to interrupt a U-tile.  Locally, we store what we want
// to tell the owner when the owner calls back.  The OCREQ table
// is first collated by tile #, and each tile gets 2^7
// consecutive bytes (32 words).  Each of these words is a
// particular OCREQ.

// since tiles 3, 7, 11, 15 are actually D-tiles, the OCREQ tile
```

```
// space corresponding to them does not contain actual OCREQs.
// instead, they contain the number of outstanding requests
// contained in the table for the preceding 3 tiles.  Thus, the
// table is set up like so:

// 0x000 [OCREQs to tile 0] - 32 words
// 0x080 [OCREQs to tile 1] - 32 words
// 0x100 [OCREQs to tile 2] - 32 words
// 0x180 - number of requests to tile 0
// 0x184 - number of requests to tile 1
// 0x188 - number of requests to tile 2
// 0x18C - 18C through 1FC are blank (29 words)

// 0x200.... tile 4, 5, 6... etc

sll $6, $9, 7 // shift target tile# left by 7
// to get an OCREQ begin address

addu $4, $6, $18 // add the master OCREQ offset to
// get the beginning of OCREQs
// for this particular tile

swlw $8, 0($4) // number of requests is the
// first word in the list.
bne $8, $0, ocreq_not1st

// number of requests increases from zero to 1.  We must contact
// the interrupt tile when this happens.  If the # of requests
// increases, say, 11 to 12, then what that means is that the
// phone-call has been made, and we do not need to make another,
// its just that multiple requests will be told to the U-tile
// when the U-tile calls back.

rlm $7, $9, 3, 0x60 // shift column over by 3 to get
// it into proper coord form
ori $7, $7, 3 // this is the I-tile handling
// this particular U-tile (same
// row, col=3)

// if the I-tile is the same, physically, as the D-tile, then do
// not send a message.  This is to avoid a deadlock in the GDN
// caused by $cgno being dependent on $cgni!

li $6, TILE_NUMBER
bne $7, $6, ocreq_send_request
```

```
rlm $7, $9, 3, 0x18 // shift up the U-tile number
// and mask only the column,
// times eight
sllv $7, $14, $7 // shift dirtile's number by
// this amount

or $16, $7, $16 // and mask in the new request

j ocreq_not1st

ocreq_send_request:


// the request to the I-tile is two words; one a GDN header, and
// a data word.  The GDN header says "1 byte long, sending to the
// I-tile".  The data word is the current D-tile.  What the GDN
// message says is "D wishes to speak to U."  The message will
// be of the form UUDDDD, where UU is a two-bit indicator of the
// U-tiles column, and DDDD is the one-hot address of the
// current D-tile this code is running on.

sll $6, $20, 25 // $6 = 2 << 24
or $cgno, $7, $6 // dynamic message to I-tile,
// size 2

addiu $cgno, $0, DI_LIST_OP // 0x300 is the opcode

sll $7, $9, 4 // shift up the U-tile number
or $cgno, $14, $7 // and swap in the dirtiles
// number, which is stored in
// $14... request sent
ocreq_not1st:

// at this point, $4 still contains the beginning offset of the
// U-tiles OCREQ.  We must add to that the correct number of
// words so that we end up at the first blank spot.  The number
// of requests (times four) is stored in $8.

addiu $8, $8, 4 // increase number-of-reqs by
// four to save some instrs in
// word addressing land.
swsw $8, 0($4) // and write this number back to
// main RAM.
```

```
addu $4, $4, $8 // first open OCREQ slot
addu $7, $2, $11 // address and request - note
// that since the address ends
// in 1, the request is now 3
// for an invalidate and 2 for
// a flush
swsw $7, 0($4) // store request.

// here, we do bounds checking.
li $6, 120 // max number of requests, times
// four.

bne+ $8, $6, ocreq_done // this branch is almost always
// taken, even if the compiler
// doesn't think so

// we just wrote in the last request into the table.  Oh dear.
// Set the stall condition.  If any OCREQ set (for any one tile)
// is filled, then we no longer take OCREQs for ANY tile.  This
// is a bit overcautious, but given that OCREQs are ideally
// passed off kinda quickly, this should be reasonable ($$$ test
// this performance)

// $13 contains one bit for each tile that has its OCREQ table
// full.  This one-hot address scheme is used because some
// OCREQs communications (not this one, but say an invalidate
// flurry!) could possibly set a lot of tiles to go from 30->31.

sllv $8, $20, $9 // 1 << stalling tile number
or $13, $8, $13 // set that bit

ocreq_done:
jr $31



// Main routine begins here.

begin:

        mtsri   SW_FREEZE, 1
        la      $4, swcode
        mtsr    SW_PC, $4
        mtsri   SW_FREEZE, 0
```

```
j init_begin
init_done:

PASS(56)

mainloop:
jal static_network_begin

jal mdn_handler_begin

jal gdn_handler_begin

// if there are pending IRQs, handle them
jnel $16, $0, pending_irq_begin

jal deadlock_detector_begin

j mainloop

DONE(1)

.end begin

int_status_0:
.word 0

int_status_1:
.word 0

int_status_2:
.word 0

        .swtext
        .align 3

swcode:

nop route $cEi->$csti
nop route $cEi->$csti
nop route $cEi->$csti
        j swcode                route $cEi->$csti

ocreq_table:
.rept 512
.word 0
```

```
.endr

pending_request_table:
.rept 512
.word 0
.endr

pending_address_table:
.rept 1024
.word 0
.endr
```

A.2 User Tile Code

This code is overhead that is loaded, along with actual user code, onto every U-tile. The insertion point for actual user code is noted. All of the user tiles have identical overhead code, except for the first several lines of dynamically generated constants.

```
#define TILE_NUM 0
#define TILE_ROTATE 0
#define INT_NUM 3
// define some features common to all the tiles for the edge detection
// application.  These are the shared addresses that we will be using.

// first, the input buffers.  4 lines of 512 words (64 cache lines)
// each.  The addresses are staggered as to not cause any cache misses.

#define IN_BUFFER_0 0x06000000
#define IN_BUFFER_1 0x26000800
#define IN_BUFFER_2 0x46001000
#define IN_BUFFER_3 0x66001800

// now, the output buffers.  Same general idea.  These are staggered
// so that different dirtiles are asked to handle requests dealing
// with corresponding in-out pairs.

#define OUT_BUFFER_0 0x46002000
#define OUT_BUFFER_1 0x66002800
#define OUT_BUFFER_2 0x06003000
```

```
#define OUT_BUFFER_3 0x26003800
// user tile.  Part of a shared-memory implementation.
// Levente Jakab
// 5/11/04 18.51

#define OP_UI_DONE 0x000
#define OP_UI_CALL 0x100
#define OP_UD_CALL 0x200
#define OP_UD_DONE 0x400
#define OP_EC 0x600

#include "module_test.h"

.text
.align 2

.global begin
.ent begin


// interrupt 3 handler
mdn_interrupt:
j mdn_code
mdn_code:

// an external interrupt signals a long chain of transactions,
// initiated by a directory tile.  First, a D-tile sends the
// U-tile an interrupt, and the U-tile ends up here.  It needs
// to figure out just who did that, so it sends an MDN message
// to the I-tile in charge.  The I-tile sends back a list of
// dirtiles that wish to talk to it, and also a possible spoof
// flag, meaning that a data word just sent to the tile was
// false, and further action must be taken.

// At this point, the U-tile talks to each of the D-tiles via
// the MDN, getting a list of addresses that must either be
// flushed or invalidated.  It does so.

// Finally, the U-tile sends to the I-tile a message saying that
// its task is complete.  The I-tile now knows it is free to
// interrupt the U-tile again.

// store some regs

isw $2, %lo(reg3_2)($0)
```

```
isw $3, %lo(reg3_3)($0)
isw $4, %lo(reg3_4)($0)
isw $5, %lo(reg3_5)($0)
isw $6, %lo(reg3_6)($0)
isw $31, %lo(reg3_31)($0)

// first, send a message to the interrupt controller.  The
// message will be of length 1.
li $cgno, ((1 << 24) | INT_NUM)

// send along the tile's number and an opcode
li $cgno, (OP_UI_CALL | TILE_NUM)

// now, wait for the I-tile to send back the list of
// requests.  When it arrives, grab the requests relevant only
// to this tile, by rotating and masking appropriately.

rrm $2, $cgni, TILE_ROTATE, 0xFF

// see if there was a spoof that needs to be taken care of.
// this is dealt with in bit 4.
andi $3, $2, 0x10 // mask all but bit 4
beq+ $3, $0, mdn_no_spoof

// a spoof has occurred.  The last lw instruction was patently
// false.  Therefore, invalidate the cache line in question
// and back up the program counter to the lw.  We may use any
// register except $2.

// First, we must grab the program counter at which the interrupt
// occurred.
mfsr $3, EX_PC

// now, we know that the interrupt occurred precisely 2 cycles
// after the offending instruction.  So back up the program
// counter by one.  We'll back it up another in a sec.
addiu $3, $3, -4

mdn_not_memop:

// back up one instruction.
addiu $3, $3, -4
ilw $4, 0($3) // load the instruction there

// now, we must make sure that the instruction is either a lw or
```

```
// a sw.

// bits 31-29 are either 010 or 001.  Thus, on 1xx, 000, or 011,
// forget about it.
rrm $5, $4, 31, 0x1 // grab the high bit
bne $5, $0, mdn_not_memop // exit on 1xx

// high bit must be a zero.
rrm $5, $4, 29, 0x3 // grab bits 30-29
beq $5, $0, mdn_not_memop // exit on 000
xori $5, $5, 0x3 // flip low bits
beq $5, $0, mdn_not_memop // exit on 011

// high bits must be either 010 or 001.  Next bits must be either
// 000, 010, or 100.  Not 110, and certainly not xx1.
rrm $5, $4, 26, 0x1 // check for xx1
bne $5, $0, mdn_not_memop // exit on xx1

// must be xx0
rrm $5, $4, 27, 0x3
xori $5, $5, 0x3 // check for 110
beq $5, $0, mdn_not_memop // exit on 110

// so what we really have here is an honest to goodness memory op
// in $4.  Extract the address, thrashing regs $5, $6,and $7.
// (and $31)
jal grab_address_and_data

// we have the address, so invalidate the cache line.
ainv $6, 0

// and set the program counter back to the offending lw/sw,
// where the cache will miss and the tile will go back to being
// in stall, once we return from handling the interrupt.
mtsr EX_PC, $3

addiu $2, $2, -10 // knock off the bit-4 mask.

mdn_no_spoof:

li $3, ((1 << 24) | 3) // load the current dirtile
// to deal with.  Start at tile
// 3.  Add on a header meaning
// an MDN message of length 1.
mdn_ud_outer_loop:
```

```
// see if we actually need to talk to this dirtile by masking
// in the lowest bit.  If not, then bypass.
andi $4, $2, 1
beq $4, $0, mdn_ud_ocreq_noneed

or $cgno, $3, $0 // send the header

// and send the data word
li $cgno, (OP_UD_CALL | TILE_NUM)

// now wait for the dirtile's response.
or $4, $cgni, $0 // $4 now contains the number
// of requests we are expecting.
// This is our main iterator.
mdn_ud_ocreq_loop:

or $5, $cgni, $0 // load an OCREQ.  This contains
// in bits 0 an opcode, and in
// bits 31-2 an address.

andi $6, $5, 0x1 // grab the opcode.  The opcode
// is 10 for invalidate and 01
// for flush.

// $2 has the list of dirtiles we need to talk to
// $3 has the current dirtile we are talking to
// $4 has the number of the request for the current dirtile
// $5 has the actual request
// $6 has the requests's opcode

// if the opcode is 10, we want an invalidate, and if it is
// 01 we want a flush.

beq $6, $0, mdn_ud_ocreq_invalidate

// must be a flush.  We can flush with extra address low-bits
// as the cache does not care.  This op coughs up the line and
// writes it to main RAM.  It also invalidates it because
// sometimes the line must be patched up.
aflinv $5, 0

// we have handled the flush.
j mdn_ud_ocreq_handled
```

```
mdn_ud_ocreq_invalidate:

// must be an invalidate.  Therefore, we invalidate the cache
// line pointed to.
ainv $5, 0

mdn_ud_ocreq_handled:

// we've taken care of the current OCREQ, whether it be
// invalidate or flush.

addiu $4, $4, -1 // subtract one more request
// and loop if there are more
// coming.
bne $4, $0, mdn_ud_ocreq_loop

mdn_ud_ocreq_noneed:

// acknowledge, to the dir tile, what we have just accomplished
or $cgno, $3, $0 // send the header

li $cgno, (OP_UD_DONE | TILE_NUM)

srl $2, $2, 1 // shift right the list of
// dirtiles needing to talk.
addiu $3, $3, 32 // and increment the current
// dirtile number.


// if there are more dirtiles wanting to talk, then loop
bne $2, $0, mdn_ud_outer_loop

// we are done, so send along a final ack to the I-tile handling
// us, letting it know that we are about to exit interrupt mode.

// The message will be of length 1.
li $cgno, ((1 << 24) | INT_NUM)

// send along the tile's number and an opcode
li $cgno, (OP_UI_DONE | TILE_NUM)

// restore regs
ilw $2, %lo(reg3_2)($0)
ilw $3, %lo(reg3_3)($0)
ilw $4, %lo(reg3_4)($0)
```

```
ilw $5, %lo(reg3_5)($0)
ilw $6, %lo(reg3_6)($0)
ilw $31, %lo(reg3_31)($0)

inton

eret

// interrupt 6 handler
event_counter_interrupt:
j event_counter_code
event_counter_code:

// an event-counter event records the cache line going from
// 'clean' to 'dirty', meaning that an sw has caused main memory
// to go out of date.  Since the event counter takes 6 cycles to
// interrupt, and there are no consecutive sw calls, we must
// check for a total of three sw ops in the instructions
// following the one that trapped.

// squirrel away registers $2-6, 31 as we will be destroying
// them.
isw $2, %lo(reg6_2)($0)
isw $3, %lo(reg6_3)($0)
isw $4, %lo(reg6_4)($0)
isw $5, %lo(reg6_5)($0)
isw $6, %lo(reg6_6)($0)
isw $7, %lo(reg6_7)($0)
isw $31, %lo(reg6_31)($0)

// grab the event counter
mfec $3, EC_WRITE_OVER_READ

srl $2, $3, 14 // program counter of triggering
// instruction.  Two bits are an
// off-by-word-versus-byte.

mtec EC_WRITE_OVER_READ, $0 // reset event counter

addiu $3, $3, 1 // add one to number of events
// to get number of extra sw's
andi $3, $3, 0xFFFF // we look for...
// 0, -1 or -2 (negative
// because event counter counts
// backwards)
```

```
// the event that triggered (PC=$4) must be an sw, by definition

// grab the instruction from imem - $3 contains the number of
// sw's detected, $2 the address of the triggering pc, $4 is the
// instruction found there.  We're really trying to minimise
// register use because we are in interrupt, so we have to store
// and restore them, which is overhead.
ilw $4, 0($2)

// now, $4 has the instruction.  This subroutine will return in
// $6 the address and in $5 the data.  $4 and $7 are destroyed,
// as is $31.
jal grab_address_and_data

// now we have the address in $6 and the data in $5.

// let's make an MDN header to the appropriate directory.  Bits
// 30-29 of the address are the row of the correct dirtile, and
// the column is "11".  The bits are put into slots 6-5.
rrm $4, $6, 24, 0x60 // shift by 24
ori $4, $4, 0x3 // and mask in

li $7, (3 << 24) // message length is 3.

or $cmno, $7, $4 // header length and target,
// sent!

// send the tile number (self addressed stamped envelope, an
// MDN header for a return of length 1)

li $7, (1 << 24)
ori $cmno, $7, TILE_NUM

// and now send the address and data
or $cmno, $6, $0
or $cmno, $5, $0

event_counter_wait_first:

// check to see if there is anything on the MDN.  This is a
// non-blocking wait that allows interrupts to fire.
        mfsr $4, MDN_BUF // pending incoming data
rrm $4, $4, 5, 0x7 // knock off "out" and other crap
```

```
        beq $4, $0, event_counter_wait_first

or $4, $cmni, $0

// wait for an acknowledgement, and possibly invalidate or
// flush.  Message 0 is none, 1 is flush, 2 is invalidate.
beq $4, $0, event_counter_none_first

andi $4, $4, 1 // grab low bit

beq $4, $0, event_counter_inv_first

afl $6, 0

j event_counter_none_first

event_counter_inv_first:

ainv $6, 0

event_counter_none_first:

// hey, that might have been it.
beq $3, $0, event_counter_done

// there must have been multiple stores... start rooting around
// the instructions immediately following the program counter.

// First, grab the LAST possible instruction (the one that was
// interrupted) - we need to check only what come before it.
mfsr $3, EX_UPC

// immediately following THIS store, cannot be another store, so
// discount that instruction and move on.
addiu $2, $2, 4

event_counter_more_addresses:
addiu $2, $2, 4 // next instruction

ilw $4, 0($2) // grab it.

// an SW looks like the following.  In bits 31-29 are 001.  In
// bits 28-26 are 000, 010, or 100.  But not 110.  Do a set of
// tests.
rrm $5, $4, 29, 0x7 // grab high 3 bits.
```

```
xori $5, $5, 0x1 // compare with "001"
bne $5, $0, event_counter_more_addresses

// looks like the top 3 bits are "001".  Now, is bit 26 a zero?
rrm $5, $4, 26, 0x1 // 1 or 0
bne $5, $0, event_counter_more_addresses

// okay, 26 is a zero.  Good.  Now are 28 and 27 both 1, because
// that is bad.
rrm $5, $4, 27, 0x3 // bits 27 and 26
xori $5, $5, 0x3 // compare with "11"
beq $5, $0, event_counter_more_addresses

// $4 is now known to be a store.  Do exactly as with the
// first word.
jal grab_address_and_data

// now we have the address in $6 and the data in $5.

// let's make an MDN header to the appropriate directory.  Bits
// 30-29 of the address are the row of the correct dirtile, and
// the column is "11".  The bits are put into slots 6-5.
rrm $4, $6, 24, 0x60 // shift by 24
ori $4, $4, 0x3 // and mask in

li $7, (3 << 24) // message length is 3.

or $cmno, $7, $4 // header length and target,
// sent!

// send the tile number (self addressed stamped envelope, an
// MDN header for a return of length 1) and the address

li $7, (1 << 24)
ori $cmno, $7, TILE_NUM

// and now send the address and data

or $cmno, $6, $0
or $cmno, $5, $0

event_counter_wait_next:

// check to see if there is anything on the MDN.  This is a
// non-blocking wait that allows interrupts to fire.
```

```
        mfsr $4, MDN_BUF // pending incoming data
rrm $4, $4, 5, 0x7 // knock off "out" and other crap

        beq $4, $0, event_counter_wait_next

or $4, $cmni, $0

// wait for an acknowledgement, and possibly invalidate or
// flush.  Message 0 is none, 1 is flush, 2 is invalidate.
beq $4, $0, event_counter_none_next

andi $4, $4, 1 // grab low bit

beq $4, $0, event_counter_inv_next

afl $6, 0

j event_counter_none_next

event_counter_inv_next:

ainv $6, 0

event_counter_none_next:

// and since we just found an SW, the next one cannot be one...
// we want to skip that exhaustive test (8 instructions,
// including some nasty branches!) as much as possible.
addiu $2, $2, 4

// see if we're done yet (if we have hit the instruction that
// got interrupted, we must be done)
bne $2, $3, event_counter_more_addresses

event_counter_done:

// Restore the regs we so consciencio usly put away.
ilw $2, %lo(reg6_2)($0)
ilw $3, %lo(reg6_3)($0)
ilw $4, %lo(reg6_4)($0)
ilw $5, %lo(reg6_5)($0)
ilw $6, %lo(reg6_6)($0)
ilw $7, %lo(reg6_7)($0)
ilw $31, %lo(reg6_31)($0)
```

```
// and we're done.
eret

// this routine takes an instruction that is an sw and extracts an
// address from it.  The instruction is in register $4.  Registers $5
// and $6 are used as temporaries.  $4 is destroyed as well.

grab_address_and_data:

// First, extract the register number from $4.  The instruction
// bits are as follows: OOOOOOAAAAADDDDDXXXXXXXXXXXXXXXX
// O = opcode.  Bits 31-26.  Already verified as store.
// A = address.  Bits 25-21.  This is the reg number we seek.
// D = data.  Bits 20-16.  What is written.  Data reg.
// X = offset.  Bits 15-0.  Dealt with later.

// rotate right by 18 bits and mask by 0b11111000, leaving the
// reg number multiplied by 8.  rrm is hella nifty.

rrm $5, $4, 18, 0xF8

// what we do here is grab the address of this big table.  We
// add to that eight times the register number in the
// instruction.  We jump to that address!  (Eight because each
// part of the switch is two instructions, thus 8 bytes.)  This
// gets the value stored in the register... either it is still
// stored in the reg, or it is fetched from instruction memory.
// Very fast.

// The result goes into $5.  Note that the four ports ($24-27)
// are blanked out - this is because attempting to re-read a
// port would be disastrous at best.  Thus, things break but at
// least don't hang.  Note to user: don't sw directly off a
// network.  Also, $1 is reserved for assembler, so that also
// may break, though it is supported, as at best it returns
// trash.

la $7, grab_address_regtable // address.
addu $7, $5, $7 // add in the reg's offset.

jr $7

grab_address_regtable:
or $5, $0, $0 // $0
j grab_address_regdone
```

```
.set noat
or $5, $0, $1 // $1 - reserved
.set at
j grab_address_regdone
ilw $5, %lo(reg6_2)($0) // $2 - stored
j grab_address_regdone
ilw $5, %lo(reg6_3)($0) // $3 - stored
j grab_address_regdone
ilw $5, %lo(reg6_4)($0) // $4 - stored
j grab_address_regdone
ilw $5, %lo(reg6_5)($0) // $5 - stored
j grab_address_regdone
ilw $5, %lo(reg6_6)($0) // $6 - stored
j grab_address_regdone
ilw $5, %lo(reg6_7)($0) // $7 - stored
j grab_address_regdone
or $5, $0, $8 // $8
j grab_address_regdone
or $5, $0, $9 // $9
j grab_address_regdone
or $5, $0, $10 // $10
j grab_address_regdone
or $5, $0, $11 // $11
j grab_address_regdone
or $5, $0, $12 // $12
j grab_address_regdone
or $5, $0, $13 // $13
j grab_address_regdone
or $5, $0, $14 // $14
j grab_address_regdone
or $5, $0, $15 // $15
j grab_address_regdone
or $5, $0, $16 // $16
j grab_address_regdone
or $5, $0, $17 // $17
j grab_address_regdone
or $5, $0, $18 // $18
j grab_address_regdone
or $5, $0, $19 // $19
j grab_address_regdone
or $5, $0, $20 // $20
j grab_address_regdone
or $5, $0, $21 // $21
j grab_address_regdone
or $5, $0, $22 // $22
```

```
j grab_address_regdone
or $5, $0, $23 // $23
j grab_address_regdone
or $5, $0, $0 // $24 - csti
j grab_address_regdone
or $5, $0, $0 // $25 - cgni
j grab_address_regdone
or $5, $0, $0 // $26 - csti2
j grab_address_regdone
or $5, $0, $0 // $27 - cmni
j grab_address_regdone
or $5, $0, $28 // $28
j grab_address_regdone
or $5, $0, $29 // $29
j grab_address_regdone
or $5, $0, $30 // $30
j grab_address_regdone
ilw $5, %lo(reg6_31)($0) // $31 - stored
j grab_address_regdone

grab_address_regdone:

// add offset, which sits in $4 and needs to be sign extended.
// $$$
// is there a more efficient way of doing this?
sll $6, $4, 16
sra $6, $6, 16 // sign extended
addu $6, $5, $6 // now $6 has the full address.

// same thing, except grab the data now.  Rotate by 13, and
// data ends up in $5

rrm $5, $4, 13, 0xF8

// what we do here is grab the address of this big table.  We
// add to that eight times the register number in the
// instruction.  We jump to that address!  (Eight because each
// part of the switch is two instructions, thus 8 bytes.)  This
// gets the value stored in the register... either it is still
// stored in the reg, or it is fetched from instruction memory.
// Very fast.

// The result goes into $5.  Note that the four ports ($24-27)
// are blanked out - this is because attempting to re-read a
// port would be disastrous at best.  Thus, things break but at
```

```
// least don't hang.  Note to user: don't sw directly off a
// network.  Also, $1 is reserved for assembler, so that also
// may break, though it is supported, as at best it returns
// trash.

la $7, grab_data_regtable // address.
addu $7, $5, $7 // add in the reg's offset.

jr $7

grab_data_regtable:
or $5, $0, $0 // $0
j grab_data_regdone
.set noat
or $5, $0, $1 // $1 - reserved
.set at
j grab_data_regdone
ilw $5, %lo(reg6_2)($0) // $2 - stored
j grab_data_regdone
ilw $5, %lo(reg6_3)($0) // $3 - stored
j grab_data_regdone
ilw $5, %lo(reg6_4)($0) // $4 - stored
j grab_data_regdone
ilw $5, %lo(reg6_5)($0) // $5 - stored
j grab_data_regdone
ilw $5, %lo(reg6_6)($0) // $6 - stored
j grab_data_regdone
ilw $5, %lo(reg6_7)($0) // $7 - stored
j grab_data_regdone
or $5, $0, $8 // $8
j grab_data_regdone
or $5, $0, $9 // $9
j grab_data_regdone
or $5, $0, $10 // $10
j grab_data_regdone
or $5, $0, $11 // $11
j grab_data_regdone
or $5, $0, $12 // $12
j grab_data_regdone
or $5, $0, $13 // $13
j grab_data_regdone
or $5, $0, $14 // $14
j grab_data_regdone
or $5, $0, $15 // $15
j grab_data_regdone
```

```
or $5, $0, $16 // $16
j grab_data_regdone
or $5, $0, $17 // $17
j grab_data_regdone
or $5, $0, $18 // $18
j grab_data_regdone
or $5, $0, $19 // $19
j grab_data_regdone
or $5, $0, $20 // $20
j grab_data_regdone
or $5, $0, $21 // $21
j grab_data_regdone
or $5, $0, $22 // $22
j grab_data_regdone
or $5, $0, $23 // $23
j grab_data_regdone
or $5, $0, $0 // $24 - csti
j grab_data_regdone
or $5, $0, $0 // $25 - cgni
j grab_data_regdone
or $5, $0, $0 // $26 - csti2
j grab_data_regdone
or $5, $0, $0 // $27 - cmni
j grab_data_regdone
or $5, $0, $28 // $28
j grab_data_regdone
or $5, $0, $29 // $29
j grab_data_regdone
or $5, $0, $30 // $30
j grab_data_regdone
ilw $5, %lo(reg6_31)($0) // $31 - stored
j grab_data_regdone

grab_data_regdone:

jr $31


// main routine.
begin:

// set up the event counter event to trigger interrupt 6

// load the constant 0x10000 (bit 16 is on)
aui $6, $0, (1 << kEC_WRITE_OVER_READ)
```

```
mtsr EVENT_CFG, $6
li $6, kEVENT_CFG2_WRITE_OVER_READ

mtsr EVENT_CFG2, $6
mtec EC_WRITE_OVER_READ, $0

// load vector of interrupt 3 handler and store it in instruction
// memory.
ilw $3, %lo(mdn_interrupt)($0)
isw $3, (3 << 4)($0)

// and the interrupt 6 handler...
ilw $3, %lo(event_counter_interrupt)($0)
isw $3, (6 << 4)($0)

// enable int 3 (MDN) and int 6 (event counter)
mtsri EX_MASK, ((1 << 3) + (1 << 6))

// interrupts on
inton

// HERE is where user-level code is inserted.

        j .
// end user level code


.end begin

// storage space

reg3_2:
.word 0

reg3_3:
.word 0

reg3_4:
.word 0

reg3_5:
.word 0

reg3_6:
```

```
.word 0

reg3_31:
.word 0

reg6_2:
.word 0

reg6_3:
.word 0

reg6_4:
.word 0

reg6_5:
.word 0

reg6_6:
.word 0

reg6_7:
.word 0

reg6_31:
.word 0

reg6_eret:
.word 0
```

A.3 Application Code

This section is the code for the edge-detection application. I divide the code into three subsections, corresponding to the distributor (tile 12), the reassembler (tile 08), and the code that is common to all the processors.

A.3.1 Distributor Code

```
PASS(59)

// code for distributor tile for edge detection application.

// This tile takes in a stream of pixels, in row-first order (an entire
// row is scanned, before moving on to the next one).  Each row
// comprises 512 words.  The actual pixel data (320 words) is written
// as ten blocks of 40 words each, one block for each processing tile.
// 40 is chosen as that is exactly five cache lines, and thus there is
// no collision.

// 32 is insufficient, despite it being the actual number of pixels,
// because the convolution requires near-neighbour pixels.  Thus, for a
// stripe of 32 result pixels, 34 input pixels are needed, with the
// leftmost and rightmost pixels needed by two tiles.  Therefore, the
// distributor tile replicates these edge pixels, so that each processor
// has a unique copy and does not have to fight with

// Since the convolution matrix requires neighbours, each set of 32
// consecutive pixels that must be done is translated into 34 pixels,
// with the edge pixels duplicated.  Normally, 32 would be sufficient,
// except for the fact that this requires a smaller number of
// semaphores.  Each tile is now assigned, uniquely, a semaphore to
// correspond to the readiness of its task, and does not have to
// monitor that of its neighbours to make sure its edge pixels are
// correct.

// Then, the next 80 words of the row (400-479) are reserved for these
// semaphores - one per cache line (8 words) per processing tile, so
// again there is no sub-line collision.

// To start, all semaphores are zero.  The distributor tile checks for
// the presence of "1", and stalls until the user clears it.  When this
// is the case, the user is done with the address space (the total
// address space for incoming rows is only four rows' worth, so it is
// cycled through quickly), and new data may be written to it.
// When the 34 words are written (in the case of an absolute edge, one
// is repeated here, so therefore the distributor takes care of all
// left-right edge problems), the corresponding semaphore is set to "1"
// and this is the signal to the processor that it may process those 32
// words.
```

```
// The processor requires three lines in order to process one.  Thus
// the processing of line N only can start when line N+1 is written,
// and line N-1 is only freed after line N is processed.  Thus, the need
// for at least three lines of address space.  Four are used, actually:
// one per physical RAM bank.  This allows for maximal distribution of
// system tile resources.

// the write-line loop is four-way unrolled to cycle through the
// addresses correctly.

row_loop_begin:

li $2, IN_BUFFER_0
jal write_row

li $2, IN_BUFFER_1
jal write_row

li $2, IN_BUFFER_2
jal write_row

li $2, IN_BUFFER_3
jal write_row

j row_loop_begin

write_row:

// here, we do the actual writing of a row.  $2 contains the
// row's starting offset.

// Load the address of the first semaphore.
addiu $9, $2, 1600

// the first write is a bit different from the rest, because
// of the edge condition.  We read in 32 pixels, and write out
// 0, 0, 1, 2, ... 30, 31, 31.  We remember 30 and 31 for the
// next tile.

write_row_first_spin:
lw $6, 0($9)
nop
nop
```

```
bne $6, $0, write_row_first_spin

// semaphore is now tested and set, so we are ready to write.
or $10, $csti, $0 // get a static word

sw $10, 0($2) // repeat first pixel.
nop
nop
nop
nop
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
nop
nop

addiu $6, $2, 120 // exit clause

// we now grab words, two at a time, and write them in.
// When we reach spot 120, we know we have just written
// pixels 29 and 30, so we exit.
write_row_first_ss:
addiu $2, $2, 8 // increment pointer

or $11, $csti, $0 // grab two static words
or $10, $csti, $0

sw $11, 0($2)
nop
nop
nop
nop
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
```

```
nop
nop

bne $2, $6, write_row_first_ss

// $2 is now 120, so write the last pixel at the edge of the
// frame, and a new one, one past it.

or $11, $csti, $0

sw $11, 8($2)
nop
nop
nop
nop
nop
nop

or $10, $csti, $0

sw $10, 12($2)
nop
nop
nop
nop
nop
nop

// set the semaphore.  Data is valid and present.
sw $9, 0($9)
nop
nop
nop
nop
nop
nop

addiu $9, $9, 32

// align $2 with the next block's start
addiu $2, $2, 40

// here, $10 and $11 contain the last two static network reads.
// important, because they are the first two pixels to be
// written to the NEXT block.
```

```
// this loop is done 8 times, for the middle blocks.

addiu $30, $2, 1280 // exit clause
write_row_ss_loop:

write_row_ss_spin:
lw $6, 0($9)
nop
nop

bne $6, $0, write_row_ss_spin

sw $11, 0($2) // store the first two pixels
nop
nop
nop
nop
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
nop
nop

addiu $6, $2, 120 // exit clause

// we now grab words, two at a time, and write them in.
// When we reach spot 120, we know we have just written
// pixels 29 and 30, so we exit.
write_row_ss_ss:
addiu $2, $2, 8 // increment pointer

or $11, $csti, $0 // grab two static words
or $10, $csti, $0

sw $11, 0($2)
nop
nop
nop
nop
```

```
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
nop
nop

bne $2, $6, write_row_ss_ss

// $6 is now 120, so write the last pixel at the edge of the
// frame, and a new one, one past it.

or $11, $csti, $0

sw $11, 8($2)
nop
nop
nop
nop
nop
nop

or $10, $csti, $0

sw $10, 12($2)
nop
nop
nop
nop
nop
nop

// set the semaphore.  Data is valid and present.
sw $9, 0($9)
nop
nop
nop
nop
nop
nop
```

```
addiu $9, $9, 32

// align $2 with the next block's start
addiu $2, $2, 40

// steady state block is done.  If this is the block at
// 1440, then it is block 9, the last one
bne $2, $30, write_row_ss_loop

write_row_last_spin:
lw $6, 0($9)
nop
nop

bne $6, $0, write_row_last_spin

sw $11, 0($2) // store the first two pixels
nop
nop
nop
nop
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
nop
nop

addiu $6, $2, 120 // exit clause

// we now grab words, two at a time, and write them in.
// When we reach spot 120, we know we have just written
// pixels 29 and 30, so we exit.
write_row_last_ss:
addiu $2, $2, 8 // increment pointer

or $11, $csti, $0 // grab two static words
or $10, $csti, $0

sw $11, 0($2)
nop
```

```
nop
nop
nop
nop
nop

sw $10, 4($2)
nop
nop
nop
nop
nop
nop

bne $2, $6, write_row_last_ss

// $6 is now 120, so write the last pixel at the edge of the
// frame twice.

or $11, $csti, $0

sw $11, 8($2)
nop
nop
nop
nop
nop
nop

sw $11, 12($2)
nop
nop
nop
nop
nop
nop

// set the semaphore.  Data is valid and present.
sw $9, 0($9)
nop
nop
nop
nop
nop
nop
```

```
// all done.  320 reads, 340 writes.
jr $31
```

A.3.2 Reassembler Code

```
PASS(59)

// code for reassembler tile for edge detection application.

// This tile reassembles lines that are processed by the processing
// tiles.  It only looks at the finished frame address space (512 words
// per row, four rows).  These 2048 words are distributed, one row per
// system tile.

// Each row consists of 320 data words, packed tight into slots 0 to 319.
// (There is no need for expansion because there is no problem with the
// convolution requiring neighbours, as in the inbound frame address
// space.)  Slots 320-399, 80 words, correspond to 10 semaphores that
// signal when data is readable.  Each cache line (8 words) contains one
// semaphore, that is a 0 when the data is not there, and 1 when it is.

// The corresponding processor tile sets the semaphore to nonzero.  The
// assembler tile spins on a zero, and when it receives a 1, it reads
// the data from shared memory, and streams it out through the static
// port.  It reads the memory in correct order, and when the data is
// read, the semaphore is set to zero.  This signals the processor tile
// that it may write to the address again.  There are four lines, so
// when line N is processed, it is overwriting line N-4, which is
// actually long gone, streamed out.

// the read-line loop is four-way unrolled to cycle through the
// addresses correctly.

row_loop_begin:


li $2, OUT_BUFFER_0
jal read_row
```

```
li $2, OUT_BUFFER_1
jal read_row

li $2, OUT_BUFFER_2
jal read_row

li $2, OUT_BUFFER_3
jal read_row

j row_loop_begin

read_row:

// here, we do the actual reading of a row.  $2 contains the
// row's starting offset.

// Load the address of the first semaphore.
addiu $9, $2, 1600

addiu $30, $2, 1280 // one past the last data offset

read_row_loop:

// first, spin on the data-ready semaphore.  If it is zero,
// then data is not ready to be read.
read_row_spin:
lw $6, 0($9)
nop
nop

beq $6, $0, read_row_spin

// data must be ready.
addiu $6, $2, 128 // escape clause

read_row_inner_loop:

lw $csto, 0($2) // read the data and send it
// out the static network
nop
nop

addiu $2, $2, 4 // next pixel

// repeat if we have not reached the end
```

```
bne $2, $6, read_row_inner_loop

sw $0, 0($9) // clear the semaphore, we are
// done with the data.
nop
nop
nop
nop
nop
nop

addiu $9, $9, 32 // next semaphore

// next?
bne $2, $30, read_row_loop

// if there is no next, then we are done with the row.
jr $31
```

A.3.3 Processor Code

```
#define LOCAL_OFFSET 0


PASS(59)

// code for a user tile.

// this is tile 0.  All tiles are fundamentally similar, except for one
// constant, LOCAL_OFFSET.  The ten processing tiles each have a
// different local offset (0 to 9), and from that they may figure out
// which stripe of pixels they are working on.

frame_begin:

// beginning of frame.  Here, we are on line 0, and there is no "line -1"
// so we only need to spin on the presence of the semaphores
// corresponding to lines 0 and 1.  The processor takes care of
// top/bottom edge problems (the distributor, of left-right).

li $2, (IN_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
```

```
init_spin_0:
lw $3, 0($2) // load the semaphore
nop
nop

beq $3, $0, init_spin_0 // if it's zero, then data is
// not ready.

li $2, (IN_BUFFER_1 + (LOCAL_OFFSET * 32) + 1600)
init_spin_1:
lw $3, 0($2) // same with row 1's semaphore
nop
nop

beq $3, $0, init_spin_1

// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
init_spin_0out:
lw $3, 0($2)
nop
nop

bne $3, $0, init_spin_0out

// now we know lines 0 and 1 are ready.  Process 32 pixels.  These
// are located in slots 4, 8, 12, ... 120, 124, 128.  The neighbours are
// where they are expected to be.

// first pixel of row 0
li $2, (IN_BUFFER_0 + (LOCAL_OFFSET * 160) + 4)

// first pixel of row 1
li $3, (IN_BUFFER_1 + (LOCAL_OFFSET * 160) + 4)

// $4 reserved for steady state

// one past last pixel of row 0 (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_0 + (LOCAL_OFFSET * 128))

// the initialiser is different from the steady state, since
// first we must load 6 new values, and later we must only
```

```
// load the 2 on the leading edge of the window.

// Three extra ops are done.  This is for code cleanness (to
// call a procedure that has pre-defined places to look).  It
// affects the performance trivially.  This way, $10-$18
// reflect the exact matrix terms, even though they are
// generated from only six pixel values.

lw $10, -4($2) // -1, -1
lw $11, -4($2) // -1, 0
lw $12, -4($3) // -1, 1
lw $13, 0($2) // -1, 0
lw $14, 0($2) // 0, 0
lw $15, 0($3) // 0, 1
lw $16, 4($2) // -1, 1
lw $17, 4($2) // 1, 0
lw $18, 4($3) // 1, 1
nop
nop

// now we have our "nine" pixel values.  One cycle is wasted
// here in the jal, but screw it, it makes the code look not
// godawful.  The result will be in register 9.  Regs 19-23
// are killed, as is reg 31.

jal convolve_and_sum

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// done with one pixel.  Now, move onto steady state.

init_process_loop:
addiu $2, $2, 4 // next input, row 0
addiu $3, $3, 4 // next input, row 1
addiu $7, $7, 4 // next output

or $10, $13, $0 // shift previously
or $11, $14, $0 // loaded values
or $12, $15, $0
```

```
or $13, $16, $0
or $14, $17, $0
or $15, $18, $0
lw $16, 4($2) // and load new ones
lw $17, 4($2)
lw $18, 4($3)
nop
nop

jal convolve_and_sum // do the math

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// if not done, loop
bne $2, $6, init_process_loop

// looks like we are done with our row.  At this point, no
// semaphore needs to be cleared, since we'll be needing both
// rows 0 and 1 in the steady state.  But we do need to set a
// semaphore on the other side.

li $2, (OUT_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop

// we need to go through the steady state 59 and one half
// times, since it is a 4-way unrolling, and there are 240
// rows.  One is at the beginning, one at the end, so 238 in
// the middle.  This means that we go through 59 complete
// times unequivocally, decrementing at the end.  Then, we
// compare with zero after stage 2 of the unrollment, thereby
// getting "59 and a half".

li $5, 59
```

```
steady_state:

// In the steady state, the loop is unrolled four ways to
// deal with four separate addressings.  Every time we return
// here, we're at row 4N+1.  Row 4N is in cache, 4N+2 not yet.

// spin on the semaphore for row 4N+2.
li $2, (IN_BUFFER_2 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_2:
lw $3, 0($2)
nop
nop

beq $3, $0, ss_spin_2

// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_1 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_1out:
lw $3, 0($2)
nop
nop

bne $3, $0, ss_spin_1out

// first pixel of rows 4N, 4N+1, 4N+2
li $2, (IN_BUFFER_0 + (LOCAL_OFFSET * 160) + 4)
li $3, (IN_BUFFER_1 + (LOCAL_OFFSET * 160) + 4)
li $4, (IN_BUFFER_2 + (LOCAL_OFFSET * 160) + 4)

// one past last pixel of top row (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_1 + (LOCAL_OFFSET * 128))

// at this point, $2, $3, $4 contain inputs pointers, $6 a
// bound, $7 an output address.  Jump to a processing subroutine.

jal ss_process

// done with row 4N+1.  Clear the semaphore for row 4N.
li $2, (IN_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
sw $0, 0($2)
nop
```

```
nop
nop
nop
nop
nop

// set the output semaphore
li $2, (OUT_BUFFER_1 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop

// spin on the semaphore for row 4N+3.
li $2, (IN_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_3:
lw $3, 0($2)
nop
nop

beq $3, $0, ss_spin_3

// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_2 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_2out:
lw $3, 0($2)
nop
nop

bne $3, $0, ss_spin_2out

// first pixel of rows 4N+1, 4N+2, 4N+3
li $2, (IN_BUFFER_1 + (LOCAL_OFFSET * 160) + 4)
li $3, (IN_BUFFER_2 + (LOCAL_OFFSET * 160) + 4)
li $4, (IN_BUFFER_3 + (LOCAL_OFFSET * 160) + 4)

// one past last pixel of top row (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_2 + (LOCAL_OFFSET * 128))
```

```
// at this point, $2, $3, $4 contain inputs pointers, $6 a
// bound, $7 an output address.  Jump to a processing subroutine.

jal ss_process

// done with row 4N+2.  Clear the semaphore for row 4N+1.
li $2, (IN_BUFFER_1 + (LOCAL_OFFSET * 32) + 1600)
sw $0, 0($2)
nop
nop
nop
nop
nop
nop

// set the output semaphore
li $2, (OUT_BUFFER_2 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop

// escape clause after 59.5 cycles
beq $5, $0, ss_done

// spin on the semaphore for row 4N.
li $2, (IN_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_0:
lw $3, 0($2)
nop
nop

beq $3, $0, ss_spin_0

// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_3out:
lw $3, 0($2)
nop
nop

bne $3, $0, ss_spin_3out
```

```
// first pixel of rows 4N+2, 4N+3, 4N+4
li $2, (IN_BUFFER_2 + (LOCAL_OFFSET * 160) + 4)
li $3, (IN_BUFFER_3 + (LOCAL_OFFSET * 160) + 4)
li $4, (IN_BUFFER_0 + (LOCAL_OFFSET * 160) + 4)

// one past last pixel of top row (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_3 + (LOCAL_OFFSET * 128))

// at this point, $2, $3, $4 contain inputs pointers, $6 a
// bound, $7 an output address.  Jump to a processing subroutine.

jal ss_process

// done with row 4N+3.  Clear the semaphore for row 4N+2.
li $2, (IN_BUFFER_2 + (LOCAL_OFFSET * 32) + 1600)
sw $0, 0($2)
nop
nop
nop
nop
nop
nop

// set the output semaphore
li $2, (OUT_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop

// spin on the semaphore for row 4N+1.
li $2, (IN_BUFFER_1 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_1:
lw $3, 0($2)
nop
nop

beq $3, $0, ss_spin_1
```

```
// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
ss_spin_0out:
lw $3, 0($2)
nop
nop

bne $3, $0, ss_spin_0out

// first pixel of rows 4N+3, 4N+4, 4N+5
li $2, (IN_BUFFER_3 + (LOCAL_OFFSET * 160) + 4)
li $3, (IN_BUFFER_0 + (LOCAL_OFFSET * 160) + 4)
li $4, (IN_BUFFER_1 + (LOCAL_OFFSET * 160) + 4)

// one past last pixel of top row (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_0 + (LOCAL_OFFSET * 128))

// at this point, $2, $3, $4 contain inputs pointers, $6 a
// bound, $7 an output address.  Jump to a processing subroutine.

jal ss_process

// done with row 4N+4.  Clear the semaphore for row 4N+3.
li $2, (IN_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
sw $0, 0($2)
nop
nop
nop
nop
nop
nop

// set the output semaphore
li $2, (OUT_BUFFER_0 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop
```

```
addiu $5, $5, -1 // one more loop gone
j steady_state


// this is where we break out to after 59.5 cycles.  At this
// point, we are at the bottom row, which is of the form 4N+3.
// We handle similarly to the initial row.
ss_done:

// make sure that the line we are writing to has been cleared.
li $2, (OUT_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
output_spin_3out:
lw $3, 0($2)
nop
nop

bne $3, $0, output_spin_3out

// first pixel of row 238
li $2, (IN_BUFFER_2 + (LOCAL_OFFSET * 160) + 4)

// first pixel of row 239
li $3, (IN_BUFFER_3 + (LOCAL_OFFSET * 160) + 4)

// one past last pixel of row 0 (bounds check)
addiu $6, $2, 128

// first pixel of output spot
li $7, (OUT_BUFFER_3 + (LOCAL_OFFSET * 128))

// note reuse of $3 since there is no row 240.
lw $10, -4($2) // -1, -1
lw $11, -4($3) // -1, 0
lw $12, -4($3) // -1, 1
lw $13, 0($2) // -1, 0
lw $14, 0($3) // 0, 0
lw $15, 0($3) // 0, 1
lw $16, 4($2) // -1, 1
lw $17, 4($3) // 1, 0
lw $18, 4($3) // 1, 1
nop
nop
```

```
jal convolve_and_sum

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// done with one pixel.  Now, move onto steady state.

final_process_loop:
addiu $2, $2, 4 // next input, row 0
addiu $3, $3, 4 // next input, row 1
addiu $7, $7, 4 // next output

or $10, $13, $0 // shift previously
or $11, $14, $0 // loaded values
or $12, $15, $0
or $13, $16, $0
or $14, $17, $0
or $15, $18, $0
lw $16, 4($2) // and load new ones
lw $17, 4($3) // note 2,3,3 not 2,2,3 as
lw $18, 4($3) // in the init!
nop
nop

jal convolve_and_sum // do the math

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// if not done, loop
bne $2, $6, final_process_loop

// We are done with rows 238 and 239, so clear both
// semaphores.
li $2, (IN_BUFFER_2 + (LOCAL_OFFSET * 32) + 1600)
```

```
sw $0, 0($2)
nop
nop
nop
nop
nop
nop

li $2, (IN_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
sw $0, 0($2)
nop
nop
nop
nop
nop
nop

// set the final output semaphore
li $2, (OUT_BUFFER_3 + (LOCAL_OFFSET * 32) + 1600)
sw $2, 0($2)
nop
nop
nop
nop
nop
nop

// start another frame
j frame_begin


convolve_and_sum:

// regs $10-$18 contain nine pixel values that need to be
// manipulated into one result, which will be in register 9.

// Deal with red first.

// $19-23 are accumulators.  I could, if I really felt motivated,
// shave a few cycles here and there.  Not motivated ;-)

rrm $19, $16, 0, 0xFF // times one
rlm $20, $17, 1, 0x1FE // times two
addu $21, $20, $19 // partial sum
```

```
rrm $19, $18, 0, 0xFF // times one
addu $21, $21, $19 // accumulate - positive done

rrm $19, $10, 0, 0xFF // times one
rlm $20, $11, 1, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $12, 0, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $21, $21, $22 // final answer

// now, same thing except the other matrix

rrm $19, $10, 0, 0xFF // times one
rlm $20, $13, 1, 0x1FE // times two
addu $23, $20, $19 // partial sum

rrm $19, $16, 0, 0xFF // times one
addu $23, $23, $19 // accumulate - positive done

rrm $19, $12, 0, 0xFF // times one
rlm $20, $15, 1, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $18, 0, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $23, $23, $22 // final answer

// here we have red's GX and GY.  Need to find abs(gx) + abs(gy)
// cheesy method: forget off-by-one, knock off the high bits.
// This method is actually correct because we're going to
// divide the result by eight, to let it fit into 0-255 (right
// now, each addend is between 0 and 1020, so the max. sum is
// 2040).
rrm $21, $21, 2, 0xFF // adjusted x
rrm $23, $23, 2, 0xFF // adjusted y
addu $21, $23, $21 // sum (0 to 510)
rrm $9, $21, 1, 0xFF // correct sum

// we now have a red value in reg $9.  Move on to green.

rrm $19, $16, 8, 0xFF // times one
rrm $20, $17, 7, 0x1FE // times two
```

```
addu $21, $20, $19 // partial sum

rrm $19, $18, 8, 0xFF // times one
addu $21, $21, $19 // accumulate - positive done

rrm $19, $10, 8, 0xFF // times one
rrm $20, $11, 7, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $12, 8, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $21, $21, $22 // final answer

// now, same thing except the other matrix

rrm $19, $10, 8, 0xFF // times one
rrm $20, $13, 7, 0x1FE // times two
addu $23, $20, $19 // partial sum

rrm $19, $16, 8, 0xFF // times one
addu $23, $23, $19 // accumulate - positive done

rrm $19, $12, 8, 0xFF // times one
rrm $20, $15, 7, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $18, 8, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $23, $23, $22 // final answer

// got green.
rrm $21, $21, 2, 0xFF // adjusted x
rrm $23, $23, 2, 0xFF // adjusted y
addu $21, $23, $21 // sum (0 to 510)
rlmi $9, $21, 7, 0xFF00 // $9 now has red and green


// here's blue
rrm $19, $16, 16, 0xFF // times one
rrm $20, $17, 15, 0x1FE // times two
addu $21, $20, $19 // partial sum

rrm $19, $18, 16, 0xFF // times one
```

```
addu $21, $21, $19 // accumulate - positive done

rrm $19, $10, 16, 0xFF // times one
rrm $20, $11, 15, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $12, 16, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $21, $21, $22 // final answer

// now, same thing except the other matrix

rrm $19, $10, 16, 0xFF // times one
rrm $20, $13, 15, 0x1FE // times two
addu $23, $20, $19 // partial sum

rrm $19, $16, 16, 0xFF // times one
addu $23, $23, $19 // accumulate - positive done

rrm $19, $12, 16, 0xFF // times one
rrm $20, $15, 15, 0x1FE // times two
addu $22, $20, $19 // partial sum

rrm $19, $18, 16, 0xFF // times one
addu $22, $22, $19 // accumulate - negative done

subu $23, $23, $22 // final answer

// got blue's GX and GY
rrm $21, $21, 2, 0xFF // adjusted x
rrm $23, $23, 2, 0xFF // adjusted y
addu $21, $23, $21 // sum (0 to 510)
rlmi $9, $21, 15, 0xFF // $9 has red, green, and blue
// packed correctly

jr $31



ss_process:

// regs 2, 3, 4 contain input addresses.  $6 contains a bound
// that we need to reach on $2.  $7 contains an output
```

```
// address.

// Load our starting 9.

lw $10, -4($2) // -1, -1
lw $11, -4($3) // -1, 0
lw $12, -4($4) // -1, 1
lw $13, 0($2) // -1, 0
lw $14, 0($3) // 0, 0
lw $15, 0($4) // 0, 1
lw $16, 4($2) // -1, 1
lw $17, 4($3) // 1, 0
lw $18, 4($4) // 1, 1
nop
nop

// in the steady state, we have nested procedures, so deal with
// multiple returns.
or $30, $31, $0
jal convolve_and_sum
or $31, $30, $0

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// done with one pixel.

ss_process_loop:
addiu $2, $2, 4 // next input, row K-1
addiu $3, $3, 4 // next input, row K
addiu $4, $4, 4 // next input, row K+1
addiu $7, $7, 4 // next output

or $10, $13, $0 // shift previously
or $11, $14, $0 // loaded values
or $12, $15, $0
or $13, $16, $0
or $14, $17, $0
or $15, $18, $0
lw $16, 4($2) // and load new ones
```

```
lw $17, 4($3)
lw $18, 4($4)
nop
nop

// do some math
or $30, $31, $0
jal convolve_and_sum
or $31, $30, $0

sw $9, 0($7)
nop
nop
nop
nop
nop
nop

// if not done, loop
bne $2, $6, ss_process_loop
```

Bibliography

[1] Anant Agarwal. Computational models. 6.846 Lecture Notes 21, April 2004.

[2] Anant Agarwal. Performance analysis of interconnection networks. 6.846 Lecture Notes 19, March 2004.

[3] Anant Agarwal. Shared memory implementation. 6.846 Lecture Notes 34, April 2004.

[4] Anant Agarwal, Richard Simoni, John Hennessey, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, New York, June 1988. IEEE.

[5] 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.

[6] Ian Bratt, Levente Jakab, and David Wentzlaff. Multiple exclusive states. Informal discussion, April 2004.

[7] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[8] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. *International Symposium on Computer Architecture*, April 1994.

[9] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: a scalable cache coherence scheme. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[11] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches. *Proceedings ICPP*, August 1988.

[12] T. Dekker. Algorithm for mutual exclusion between two processes sharing memory. 1962.

[13] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, (9), September 1965.

[14] Edsger W. Dijkstra. Co-operating sequential processes. *Programming Languages*, 1968.

[15] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Willson. Issues related to mimd shared-memory computers: the nyu ultracomputer approach. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, 1985.

[16] Albert Einstein. On the electrodynamics of moving bodies. *Annalen der Physik*, 1905.

[17] S. H. Fuller and S. P. Harbison. The c.mmp multiprocessor. Technical Report, Carnegie Mellon University, October 1978.

[18] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. 10 Ann. Symp. Comp. Arch.*, pages 124–131, June 1983.

[19] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze,

and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *ASPLOS 2002*, October 2002.

[20] Bill Green. Edge detection tutorial. 2002.

[21] Patrick Griffin, Levente Jakab, and James Psota. Fixme performance analysis of various platforms on the raw architecture. 2004.

[22] J. L. Hennessey and D. A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[23] Henry Hoffmann, Volker Strumpen, and Anant Agarwal. Stream algorithms and architecture. *MIT Technical Memo*, March 2003.

[24] *Electronic Data-Processing Machines. Type 704: Manual of Operation.* International Business Machines Corporation, New York, 1955.

[25] Levente Jakab. The usb host interface. May 2004.

[26] Ho-Seop Kim and James E. Smith. An instruction set architecture and microarchitecture for instruction level distributed processing. *International Symposium on Computer Architecture*, June 2002.

[27] J. Konicek, T. Tilton, A. Veidenbaum, C. Q. Zhu, E. S. Davidson, R. Downing, M. Haney, M. Sharma, P. C. Yew, P. Farmwald, D. Kuck, D. Lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, and N. Warter. The organization of the cedar system. *Proceedings of the International Conference on Parallel Processing*, I:49–56, 1991.

[28] Jeffery Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford flash multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[29] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear analysis and optimization of stream programs. *Proceedings of the SIGPLAN '03 Conference on Programming Langauge Design and Implementation*, June 2003.

[30] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[31] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessey. The directory-based cache coherence protocol for the dash multiprocessor. *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[32] Jan Lindheim. The beowulf project at cacr. 2000.

[33] J. S. Liptay. Structural aspects of the system/360 model 85. ii: The cache. *IBM Systems Journal*, 7(1):15–21, January 1968.

[34] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. *International Symposium on Computer Architecture*, June 2000.

[35] Steve McGrogan, Robert Olson, and Neil Toda. Parallelizing large existing programs - methodology and experiences. *Proceedings of Spring COMPCON*, pages 458–466, March 1986.

[36] Jason E. Miller. Storing of data in raw's instruction memory. Private communication with author., May 2004.

[37] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. *34th Annual International Symposium on Microarchitecture*, December 2001.

[38] *OpenMP C and C++ Application Program Interface.* OpenMP Architecture Review Board, 2002. www.openmp.org.

[39] Susan Owicki and Anant Agarwal. Evaluating the performance of software cache coherence. *Proceedings, Architectural Support for Programming Languages and Operating Systems*, March 1989.

[40] James Psota. rmpi: A message passing library for raw. May 2004.

[41] Arnold Robbins. *Unix in a Nutshell: System V Edition.* O'Reilly Media, 1999.

[42] Karthikeyan Sankaralingam, Ramadass Nagarajan, Stephen W. Keckler, and Doug Burger. A technology-scalable architecture for fast clocks and high ilp". *Proceedings of HPCA-7*, January 2001.

[43] Nathan R. Shnidman. Memory controller documentation. Raw Group internal document., August 2001.

[44] David Signoff. untitled document. A tentative description of the Raw shared-memory system., July 2002.

[45] Alan Jay Smith. Cpu cache consistency with software support and using one time identifies. *Proceedings of the Pacific Computer Communications Symposium*, October 1985.

[46] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[47] Michael B. Taylor. Testing and verification for the raw chip. /starsearch/README, December 2003.

[48] Michael Bedford Taylor. Comprehensive specification for the raw processor. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/ieee-micro-2002.pdf, December 2003.

[49] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal.

A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[50] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: a computational fabric for software circuits and general purpose programs. *IEEE Micro*, Mar/Apr 2002.

[51] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. *International Symposium on High Performance Computer Architecture*, February 2003.

[52] Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The hector multiprocessor. *IEEE Computer*, 24(1):12–24, January 1991.

[53] Peter Wolcott and S. E. Goodman. High-speed computers of the soviet union. *IEEE Computer*, pages 32–41, September 1988.

[54] J. Jay Wolf, editor. *Burroughs Corporation Records.* Charles Babbage Institute, Minneapolis, Minnesota, 1990.

[55] *Virtex-II Platform FPGAs: Introduction and Overview.* Xilinx Corporation, 2004.

[56] E. U. Yevreinov and Y. G. Kosarev. High efficiency computing systems. *Engineering Cybernetics*, (4):1–18, 1963.