LOCK-BASED CACHE COHERENCE PROTOCOL FOR CHIP MULTIPROCESSORS

A Thesis Submitted to The Computer Science Department In partial fulfillment of the requirements for The degree of Master of Science

By

Ihab Hossam Ismail

B.Sc. Systems and Biomedical Engineering

Cairo University

(Under the supervision of Dr. Khaled El-Ayat)

November 2005
LOCK-BASED CACHE COHERENCE PROTOCOL FOR CHIP MULTIPROCESSORS

A Thesis Submitted to
The Department of Computer Science
July 2005
In partial fulfillment of the requirements for
The degree of Master of Science
Has been approved by

Dr. Khaled El-Ayat
Thesis Committee Chair/Adviser
Affiliation

Dr. Ahmed Sameh
Thesis Committee Reader/examiner
Affiliation

Dr. Gamal Aly
Thesis Committee Reader/examiner
Affiliation

Dr. Mohy Mahmoud
Thesis Committee Reader/examiner
Affiliation

Department Chair/Program Director Date
Dean Date
ACKNOWLEDGEMENTS

I wish to thank everyone who participated in making this work come true.

First of all, I would like to thank my wife and my son for bearing with me all the times of frustration during research and for sharing the joy when the research started to fruit its results. I hope my son would understand why I missed a lot of his playing time.

I would like to thank my parents for their continuous support and encouragement until this research came true.

Many thanks to Prof. Mikhail Mikhail (Chairman Computer Science Department) for giving the Graduate Students the chance to work in the laboratory 24/7 times.

Many thanks to the Computer Science Department Engineers and laboratory assistants for helping me in keeping the machine I was working on in good shape.

A lot of thanks to my reviewers for the issues they raised during my proposal presentation which were very important for this work to be completed.

Special thanks to Dr. Mohammed Mudawwar for his ideas that inspired me for this work.

And finally special thanks to Dr. Khaled El-Ayat; my Supervisor who was always there when I needed his support and who always had time for my research helping me overcome all the difficulties I faced.
Chip multiprocessor (CMP) is replacing the superscalar processor due to its huge performance gains in terms of processor speed, scalability, power consumption and economical design. Since the CMP consists of multiple processor cores on a single chip usually with share cache resources, process synchronization is an important issue that needs to be dealt with. Synchronization is usually done by the operating system in case of shared memory multiprocessors (SMP).

This work studies the effect of performing synchronization by the hardware through its integration with the cache coherence protocol. A novel cache coherence protocol, called Lock-based Cache Coherence Protocol (LCCP) was designed and its performance was compared with MESI cache coherence protocol. Experiments were performed by a functional multiprocessor simulator, MP_Simplesim, that was modified to do this work.

A novel interconnection network was also designed and tested in terms of performance against the traditional bus approach by means of simulation.
LIST OF FIGURES

Figure 1-1 Cache hierarchy alternatives for CMP ......................................................... 3

Figure 2-1 State transition of Goodman Protocol ......................................................... 9

Figure 2-2 MSI Cache Coherence Protocol state diagram ............................................. 13

Figure 2.3 MESI Cache Coherence Protocol state diagram ........................................ 14

Figure 2-4 Firefly state transition diagram .................................................................. 16

Figure 2-5 Full-map Directory Cache-Coherence protocol ............................................ 19

Figure 2-6 Limited Directory Cache-Coherence Protocol ............................................ 20

Figure 2-7 Chained Directory Cache-Coherence Protocol .......................................... 21

Figure 3-1 Overview of the Hydra CMP ..................................................................... 23

Figure 3-2 Block diagram of a single-chip Piranha processing node .......................... 25

Figure 3-3 Power4 chip logical view ........................................................................... 30

Figure 3-4 L2 Logical Views ....................................................................................... 34

Figure 3-5 On-chip interconnects in Raw ................................................................. 40

Figure 3-6 Raptor Block Diagram ................................................................................. 43

Figure 4-1 Proposed CMP architecture ....................................................................... 46

Figure 4-2 Implementation of LL/SC instruction pair ................................................. 48
Figure 4-3 State machine diagram for the Lock-based cache coherence protocol .....50

Figure 4-4 Structure of the lock-release wait-table..............................................55

Figure 4-5 Structure of the Processor ID field......................................................56

Figure 4-6 Architecture of the CMP implementing the Lock-based Cache Coherence Protocol........................................................................................................58

Figure 5-1 RSIM multiprocessor system...............................................................61

Figure 5-2 MP_Simplesim Architecture Model ....................................................63

Figure 5-3 Modified MP_Simplesim Architecture Model.................................65

Figure 6-1 Execution time of Ocean contiguous-partitions benchmark...............72

Figure 6-2 Execution time of Ocean non-contiguous-partitions benchmark........72

Figure 6-3 Execution time of Barnes-Hut benchmark.......................................73

Figure 6-4 Percentage improvement in execution time when using LCCP...........73

Figure 6-5 Execution rate of Ocean contiguous-partitions benchmark..............75

Figure 6-6 Execution rate Ocean non-contiguous-partitions benchmark.........76

Figure 6-7 Execution rate Barnes-Hut benchmark............................................76

Figure 6-8 Number of acknowledged invalidations - Ocean contiguous-partitions....78
Figure 6-9 Number of acknowledged invalidations - Ocean non-contiguous-partitions ................................................................. 78

Figure 6-10 Number of acknowledged invalidations - Barnes-Hut ................................................. 79

Figure 6-11 Percentage improvement in acknowledged invalidations .............................. 79

Figure 6-12 Execution time measured while varying size of L2 cache .............................. 81

Figure 6-13 Acknowledged invalidations varying the size of L2 cache ............................. 82

Figure 6-14 Execution time of ocean contiguous-partitions using interconnection network and bus ........................................................................................................... 83

Figure 6-15 Execution time of ocean non-contiguous-partitions using interconnection network and bus with LCCP .............................................................. 83

Figure 6-16 Execution time of Barnes-hut using interconnection network and bus with LCCP .............................................................. 84

Figure 6-17 Execution rate of ocean contiguous-partitions benchmark using the interconnection and bus with LCCP .............................................................. 85

Figure 6-18 Execution rate of ocean non-contiguous-partitions benchmark using the interconnection and bus with LCCP .............................................................. 85

Figure 6-19 Execution rate of Barnes-hut benchmark using the interconnection and bus with LCCP .............................................................. 86
Figure 6-20 Ack. invalidations of ocean contiguous-partitions using the interconnection and bus with LCCP .......................................................... 86

Figure 6-21 Ack. invalidations of ocean non-contiguous-partitions using the interconnection and bus with LCCP .......................................................... 87

Figure 6-22 Ack. invalidations of Barnes-hut using the interconnection and bus with LCCP .............................................................................. 87
LIST OF TABLES

Table 3-1 Storage hierarchy organization and size ...................................................... 32

Table 4-1 Illustration of synchronization using LCCP ................................................ 59

Table 5-1 Comparing RSIM, Simics and MP_Simplesim simulators ......................... 64

Table 6-1 Benchmark types and data sets .................................................................... 68

Table 6-2 MP_Simplesim verification results ............................................................. 69

Table 6-3 Configurations of L1 and L2 caches used in experiments .......................... 70

Table 6-4 Execution times of ocean contiguous-partitions benchmark ....................... 71

Table 6-5 Execution times of ocean non-contiguous-partitions benchmark ............... 72

Table 6-6 Execution times of Barnes-hut benchmark ................................................ 73
LIST OF ACRONYMS

CMP: Chip Multi-Processor.

ICS: Intra-Chip Switch.

ISA: Instruction Set Architecture.


LL: Load-Linked.

LRU: Least Recently Used.

MC: Memory Controller.

MESI: Modified, Exclusive, Shared, Invalid.

MOESI: Modified, Owned, Exclusive, Shared.

MSI: Modified, Shared, Invalid.

OTP: Online Transaction Processing.

SC: Store-Conditional.

SMT: Simultaneous Multi Threaded.

TLB: Table Look-aside Buffer.
TABLE OF CONTENTS

1 INTRODUCTION .......................................................................................... 1

1.1 Chip Interconnects .................................................................................. 2

1.2 Cache Hierarchy Alternatives for CMP ................................................... 2

1.3 Motivation ............................................................................................... 4

1.4 Thesis Objective ..................................................................................... 4

1.5 Thesis Roadmap ..................................................................................... 5

2 CACHE COHERENCE IN SHARED MEMORY MULTIPROCESSORS ...... 7

2.1 Snoopy Cache Coherence Protocols ..................................................... 8

2.1.1 Write Invalidate Protocols ................................................................ 8

2.1.1.1 Goodman Protocol .................................................................... 8

2.1.1.2 Synapse Protocol: ................................................................... 11

2.1.1.3 Berkeley Protocol .................................................................... 11

2.1.1.4 MSI Protocol ........................................................................... 11

2.1.1.5 MESI (Illinois) Protocol ............................................................ 13

2.1.1.6 MOESI Protocol ...................................................................... 15

2.1.2 Write Update Protocols ..................................................................... 15
2.1.2.1 Firefly Protocol ................................................................. 15

2.2 Directory Cache Coherence Protocols ........................................... 17

2.2.1 Full-map Directory Cache-Coherence Protocol ............................. 18

2.2.2 Limited Directories Cache-Coherence Protocol ............................. 19

2.2.3 Chained Directories Cache-Coherence Protocol .......................... 20

3 CACHE COHERENCE IN CMP ......................................................... 22

3.1 The Stanford Hydra CMP ............................................................... 22

3.1.1 Hydra CMP Composition ............................................................ 22

3.1.2 Cache Coherence in Hydra CMP ................................................ 24

3.2 The Compaq Piranha CMP ............................................................... 24

3.2.1 Piranha CMP Architecture .......................................................... 25

3.2.2 Cache Coherence in Piranha CMP ................................................. 26

3.2.2.1 Inter-chip coherence protocol .................................................. 27

3.2.2.2 Intra-chip coherence protocol .................................................. 27

3.3 IBM POWER4 CMP ................................................................. 28

3.3.1 Design Principles ................................................................. 28

3.3.2 POWER4 Chip Composition ...................................................... 30
3.3.3 Cache Hierarchy and Coherence in POWER4 ........................................ 32

3.4 IBM POWER5 CMP .............................................................................. 38

3.4.1 Cache Implementation.......................................................................... 38

3.5 The Raw Microprocessor ..................................................................... 39

3.6 RAPTOR Single Chip Multiprocessor................................................... 43

3.6.1 RAPTOR Hardware Architecture ....................................................... 43

3.6.2 RAPTOR Cache Coherence.................................................................. 44

4 LOCK-BASED CACHE COHERENCE PROTOCOL FOR CMP ............... 45

4.1 Synchronization ................................................................................... 46

4.2 Finite State-Machine of LCCP............................................................... 50

4.3 Cache Coherence Details ..................................................................... 53

4.3.1 Read Miss (no other copy is found) ................................................... 53

4.3.2 Read Miss (one or more copies are found)........................................ 54

4.3.3 Write hit ............................................................................................. 54

4.3.4 Write miss .......................................................................................... 54

4.3.5 Read-Modify-Write miss ..................................................................... 55

4.4 Architecture of the CMP with LCCP .................................................... 56
4.5 Synchronization using LCCP ............................................................... 58

5. EXPERIMENTAL METHODOLOGY ......................................................... 60

5.1 Multiprocessor Simulators ............................................................... 60

5.1.1 RSIM ..................................................................................................... 60

5.1.2 Simics .................................................................................................. 61

5.1.3. MP_Simplesim ..................................................................................... 62

5.2 MP_Simplesim Modifications ............................................................... 64

5.3 The Benchmarks .................................................................................. 66

5.3.1 Ocean ................................................................................................... 66

5.3.2 Barnes-Hut ........................................................................................... 67

5.4 Workflow .............................................................................................. 67

6 RESULTS AND ANALYSIS ....................................................................... 68

6.1 MP_Simplesim Verification ............................................................... 69

6.2 Experimental Setup .............................................................................. 70

6.3 Execution time ..................................................................................... 71

6.4 Number of instructions executed per second ....................................... 75

6.5 Number of acknowledged invalidations ................................................ 77
1 INTRODUCTION

With the decrease in CMOS technology attempting to reach ultra-small 35 nanometer devices, new processors have been designed to use these transistors effectively to improve performance [Ham00]. It is expected that Chip Multi-Processor (CMP) will be the design choice for future servers for four reasons.

First, the superscalar paradigm is reaching diminishing returns particularly as clock scaling will soon slow sharply [HBK01].

Second, wire delays will limit the area of the chip that is useful for a single conventional processing core. The International Technology Roadmap for Semiconductors (ITRS) [ITRS] projects that multi-billion chips will be designed by the end of this decade with feature size around 50 nm and clock frequencies around 10 GHz. Benini and De Micheli [BD02] showed that delays on wires will dominate. They calculated the delay on wires that span a significant fraction of the chip and proved that the propagation delay will exceed the clock cycle. Although large delays can be managed by pipelining techniques, timing uncertainty will be a problem for designers.

A third problem is design time since SuperScalar processors are difficult to design. CMP reduces design time because it allows a single processor to be replicated multiple times over a die. Each processor core on a CMP can be smaller than a competitive uniprocessor and that minimizes the core design time.

The fourth problem in high frequency superscalar design is very high power consumption.
1.1 Chip Interconnects

Processor cores in CMP architecture can communicate together either through shared on-chip caches, through an on-chip bus in case of an on-chip shared L2 cache or through an interconnection network in case of private L1 and L2 caches.

The on-chip network structures the global wiring so that their electrical parameters such as cross-talk is low and predictable and power dissipation is reduced by a factor of ten and propagation velocity is increased by three times. [PA02]

Current chip multiprocessors (CMP) such as the Stanford Hydra, the Compaq Piranha, IBM POWER4 and IBM POWER5 use different architectures to interconnect the processing cores. Stanford Hydra uses 2 buses, READ and WRITE. Compaq Piranha, IBM POWER4 and IBM POWER5 use internal switches. The Raw CMP utilized a mesh interconnection network to connect its 64 cores however the rest of the design was complex and was specially tailored in some of its parts to deal with specific problems.

1.2 Cache Hierarchy Alternatives for CMP

Cache coherence is a major factor in the design of CMP. There are several alternatives for building the cache architecture for a CMP. The simplest one is to use a single multi-ported cache shared by multiple processors. There are two advantages of such architecture; constructive interference can reduce overall miss-rate and inter-processor communication is easy to implement. The major disadvantage of this approach is that it is not scalable.
The second alternative is to allow each processor to have its own private Level 1 (L1) cache while all processors share a Level 2 (L2) cache. The main advantages of this approach are low access latency and good scalability. The main drawback of this approach is cache coherence resulting from duplicate copies in different caches. Cache coherence protocols are needed to maintain data consistency between the individual caches. Figure 1-1 shows the different cache design alternatives for CMPs.

**Figure 1-1** Cache hierarchy alternatives for CMP
1.3 Motivation

Existing chip multiprocessors did not tackle the problem of implementing atomic synchronization primitives in the cache coherence protocols they used. The support of the load-Linked and store-Conditional (LL/SC) synchronization primitives was implemented in the Stanford Hydra CMP processor.

The LL/SC synchronization primitive is implemented in the processor by loading the synchronization variable from its memory location or cache block into a register raises a *lock flag* and places the address of the block in a *lock address* register. The lock flag can be reset if the cache block is replaced or in case of a context switch. The store-conditional checks the lock flag and if set, a write to the synchronization variable occurs, otherwise it fails.

MESI cache coherence protocol supports synchronization with the help of a spin-waiting algorithm implemented by the programmer. This spin-waiting algorithm increases the traffic on the internal bus of the CMP.

The traffic generated from the spin-waiting algorithm and subsequent re-loading of the cache block containing the synchronization lock variable in case of a store-Conditional (SC) failure motivated this research work.

1.4 Thesis Objective

The thesis objective is to remove the spinning effect of the MESI cache coherence protocol when waiting on a synchronization variable. This is done by introducing a specially designed cache coherence protocol called Lock-based Cache Coherence Protocol (LCCP). The performance of the newly introduced protocol will be
compared to MESI protocol supported by spin-waiting protocol. The newly designed LCCP will be implemented and its performance will be measured.

Another objective will be to design a novel interconnection network and use it to test the performance of LCCP and compare it to the original bus design.

1.5 Thesis Roadmap

Chapter 2 gives an extensive literature review for cache coherence protocols in multiprocessors. It discusses the design of bus-based (snoopy) cache coherence protocols as well directory based cache coherence protocols.

Chapter 3 reviews existing chip multiprocessors such as Compaq Piranha, Stanford Hydra, IBM POWER4, IBM POWER5, RAW and RAPTOR processors. The internal designs of such processors are explored with a focus on the cache structure and the cache coherence protocol used.

Chapter 4 introduces the *Lock-based Cache coherence protocol* (LCCP). It discusses the load-linked / store-conditional (LL/SC) instruction and how it is implemented in hardware. The finite-state-machine (FSM) diagram of the protocol is shown and details of cache coherence are explained.

Chapter 5 gives an overview of the simulators used to simulate chip multiprocessors (CMP) and the benchmarks used to simulate this work. Simulator modifications to accommodate LCCP is explained.

Chapter 6 shows the results of the simulation experiments performed on the design. It also introduces the design of a novel interconnection network to be used in the CMP.
instead of the bus. The design of the interconnection network is explored with the aid of the simulator and results are compared with those of bus-based CMP.

Finally, chapter 7 concludes the thesis work and suggests some future work.
2 CACHE COHERENCE IN SHARED MEMORY
MULTIPROCESSORS

Cache memories have served as an important way to reduce the average memory access time in uni-processors. Due to the locality of memory referencing over time (temporal locality) and space (spatial locality) more than 95 percent of all memory requests are fulfilled by the cache and only 5 percent are fulfilled by main memory itself.

Shared memory multiprocessors have the advantage of sharing code and data structures among the processors comprising the parallel application. This sharing can result in several copies of a shared block in one or more caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is the cache coherence problem.

Cache coherence schemes include protocols that maintain coherence in hardware, and software policies that prevent the existence of copies of shared, writable data in more than one cache at the same time.

Hardware coherent protocols include snoopy cache protocols, directory cache protocols and cache-coherent network architectures [Ste90].
2.1 Snoopy Cache Coherence Protocols

Snoopy cache coherence protocols are good-match for bus-based, shared memory multiprocessors; because they take advantage of the broadcast capability of the single interconnect [Zah03] [TM93].

Two main protocol categories are available in the snoopy cache coherence protocols; Write Invalidate and Write Update.

2.1.1 Write Invalidate Protocols

In this type of protocols the processor that modifies its block invalidates all other cached copies of shared data and can then update its own without further bus operations. There are four protocols under this category:

2.1.1.1 Goodman Protocol

This protocol was the first write-invalidate protocol. It was proposed by Goodman and was called write-once protocol.

The write-once protocol associates a state with each cached copy of a block. Possible states for a copy are

- **INVALID**: The copy is inconsistent.
- **VALID**: There exists a valid copy consistent with the memory copy.
- **RESERVED**: Data has been written only once and the copy is consistent with the memory copy; which is the only other copy.
• **DIRTY:** Data has been modified more than once and the copy is the only one in the system.

The protocol uses a *copy-back* memory update policy. To maintain consistency the protocol supports besides the normal *memory read* and *memory write* the following commands:

- Write-Inv: Invalidates all other copies of the block.
- Read-Inv: Reads the block and invalidates all other copies.

The four states of the protocol are shown in Figure 2-1. The *solid lines* mark processor-initiated actions and *dashed lines* mark consistency actions initiated by other caches and sent over the bus.

![Figure 2-1 State transition of Goodman Protocol](image-url)
Coherence details:

a. Read miss

If there is no DIRTY copy in one cache, then the memory has a consistent copy and supplies a copy to the requesting cache. This copy will be in a VALID state. If a DIRTY copy exists then the corresponding cache inhibits the memory and sends a copy to the requesting cache. Both caches will be changed to VALID because the memory is updated also.

b. Write hit

If the copy is in DIRTY or RESERVED state then the write can be executed locally and the state is changed to DIRTY. If the state is VALID then an invalidate signal is broadcast to all caches to invalidate their copies, then the memory copy is updated and the new state will be RESERVED.

c. Write miss

When there is a write miss, the requesting cache will either get a copy from another cache that has a DIRTY copy or from memory itself. This will result in a Read-Inv command that invalidates all cached copies. If there is a cache that has a DIRTY copy, it will provide this copy to the requesting processor and will update the memory before it invalidates its copy. The copy is updated locally and the state is changed to DIRTY.
2.1.1.2 Synapse Protocol

This approach was used in the Synapse N+1 multiprocessor for fault tolerant transaction processing [AB86]. The N+1 differs from the other shared bus designs in that it has two system buses. The added bandwidth of the extra bus allows the system to be expanded to a maximum of 28 processors. Also a single bit is included with each cache block in main memory. This bit indicates whether main memory is to respond to a miss on that block. If a cache has a modified copy of the block, the bit tells the memory that it need not respond. This prevents a possible race condition if a cache does not respond quickly enough to inhibit main memory from responding.

2.1.1.3 Berkeley Protocol

Berkeley scheme is similar to the Synapse approach with two major differences:

- It uses direct cache-to-cache transfers in the case of shared blocks
- Dirty blocks are not written back to memory when they become shared requiring additional state called SHARED-DIRTY.

2.1.1.4 MSI Protocol

The protocol uses the following three states in order to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty) [CS99].

- **MODIFIED (M):** Only this cache has a valid copy of the block.
- **SHARED (S):** The block is in an unmodified state in this cache, main memory is up-to-date and zero or more other caches may also have an up-to-date (shared) copy.
• **INVALID (I):** Data present in the cache is not valid.

The processor issues two types of requests: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. If the block is not in the cache, another block currently in the cache will have to be replaced with the newly requested block and if the existing block is in the modified state, its contents will have to be written back to memory.

The bus allows the following transactions:

- **Bus Read (BusRd):** A PrRd that misses in the cache generates this transaction and the processor expects a data response as a result. The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system (possibly another cache) supplies the data.

- **Bus Read Exclusive (BusRdX):** This transaction is generated by a PrWr to a block that whether is not in the cache or is in the cache but not in the modified state. The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system (or another cache) supplies the data. All other caches are invalidated. Once the cache obtains the exclusive copy, the write can be performed in the cache. The processor may require an acknowledgement as a result of this transaction.

- **Bus Write Back (BusWB):** A cache controller generates this transaction on a write back; the processor does not know anything about it and does not expect a response. The cache controller puts the address and the contents of the memory block on the bus. The main memory is updated with the latest contents.
Figure 2-2 shows the MSI cache coherence protocol state diagram

![MSI Cache Coherence Protocol state diagram](image)

**Figure 2-2** MSI Cache Coherence Protocol state diagram [CS99]

2.1.1.5 MESI (Illinois) Protocol

A concern arises when using the MSI protocol with a sequential program running on a multiprocessor. When the process reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in S state and the second is a BusRdX that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and by loading the block in this
state, we can save the later transaction since the state indicates that no other processor is caching the block.

The new state is:

**EXCLUSIVE (E):** Only one cache (this cache) has a copy of the block and it has not been modified (memory is up-to-date). Figure 2-3 shows MESI protocol.

![MESI Cache Coherence Protocol state diagram](CS99)
2.1.1.6 MOESI Protocol

A fifth OWNED state may be added which indicated that even though other shared copies of the block may exist, this cache (instead of main memory) is responsible for supplying the data when it observes a relevant bus transaction. This led to a five-state MOESI protocol [CS99] [SS86].

2.1.2 Write Update Protocols

In the write-update protocols the processor broadcasts updates of shared data to other caches, so that all copies remain identical. An example of a write-update protocol is the *Firefly protocol*. 

2.1.2.1 Firefly Protocol

This protocol got its name because it was implemented in the Firefly multiprocessor workstation from Digital Equipment [AB86]. It uses a state diagram that consists of three states as shown in figure 2-4.
Figure 2-4 Firefly state transition diagram [AB86]

- **Valid-Exclusive**: the only copy in caches and not modified.
- **Shared**: The cache copy is not modified but other caches have a copy
- **Dirty**: The cache is modified and it is the only copy in caches.

There is a special bus line used to detect sharing and this is called the *SharedLine*

Coherence details:

a. **Read miss**

If another cache has a block, it supplies it directly to the requesting cache and raises the *SharedLine*. All caches with a copy respond by putting the data on the bus. All caches including the requesting cache set the state to SHARED. If the owning cache had the block in the state DIRTY, the block is written to main memory at the same time. If no other cache has a
copy of the block, it is supplied by main memory and it is loaded in the state VALID-EXCLUSIVE.

b. Write hit

If the block is DIRTY, the write can take place without delay. If the block is in the state VALID-EXCLUSIVE, the write is also performed immediately and the state is changed to DIRTY. If the block is in the state SHARED, the write is delayed until the bus is acquired and a write-word to main memory is initiated. Other caches with the block observe the write-word on the bus and take the new data and overwrite that word in their copy of the block.

c. Write miss

Similar to a read-miss, the block is supplied by other caches, if any other caches have a copy. If the block is coming from memory it is loaded in state DIRTY and written to without overhead. If it came from a cache it is loaded in state SHARED and the requesting cache must write the word to memory. Other caches with a copy of the block will take the new data and overwrite the old block contents with the new one.

2.2 Directory Cache Coherence Protocols

Snoopy cache protocols are perfect choice for multiprocessors using bus-based memory systems because all the processors can observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take such appropriate action as invalidating the local copy.
Scalable multiprocessor systems interconnect using short point-to-point wires in direct or multistage networks. To support such an interconnection a cache-coherence protocol that does not use broadcasts and stores the locations of all cached copies of each block of shared data was developed. Such a protocol is called Directory Cache-Coherence Protocol [Cha90]. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block. Each directory entry contains a dirty-bit to specify whether or not a unique cache has permission to write the associated block of data.

Directory protocols fall under three categories:

2.2.1 Full-map Directory Cache-Coherence Protocol

This protocol uses directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processors cache. If the dirty bit is set, then one and only one processor’s bit is set and that processor has permission to write into the block. A cache maintains two bits of state per block. One bit indicates whether a block is valid; the other bit indicates whether a valid block may be written. The protocol must keep the state bits in the memory directory and those in the caches consistent. Figure 2-5 shows an example of three processors and a shared memory using full-map directory cache-coherence.
In Figure 2-5 processors P1, P2 and P3 request to *read* location x, as a result the memory replies to all of them with the data. The directory entry of location x reflects the locations of those processors that have a copy of the data. The status bit is set to *clean*.

Processor P3 requests to *write* to location x. This time the directory removes previous history for this block, sets the status bit to *dirty* and then gives P3 the grant to change data. Processor P3 will be the only one set in the directory entry.

### 2.2.2 Limited Directories Cache-Coherence Protocol

The full-map protocol provides a useful upper bound for the performance of centralized directory based cache coherence. However it is not scalable with respect to memory overhead. Since the size of directory is proportional to the number of
processors so the space consumed by the directory for \( N \) processors and \( N \) memory blocks will be proportional to \( N^2 \).

\[
\text{Size} = \Theta(N^2)
\]

Eq 2.1

Limited directories solve the directory size problem by limiting the number of simultaneously cached copies of any particular data block. This directory protocol uses the notation Dir\(_i\)X where the suffix \( i \) stand for the number of pointers and \( X \) is either NB for a scheme with No Broadcast and B for one with Broadcast.

Figure 2-6 shows the limited directory cache protocol

![Limited Directory Cache-Coherence Protocol](image)

**Figure 2-6 Limited Directory Cache-Coherence Protocol**

The protocol shown in Figure 2-6 is Dir\(_2\)NB. At the beginning processors P1 and P2 had copies of the block at location \( x \), then processor P3 requests a read for location \( x \).

Here a pointer replacement algorithm is used to replace one of the pointers and the pointer of P3 is placed in the directory.

### 2.2.3 Chained Directories Cache-Coherence Protocol

This protocol is called chained because it keeps track of shared copies of data by maintaining a chain of directory pointers. Figure 2-7 shows such a protocol.
Suppose that there are no shared copies of location X. If processor P1 reads location X the memory sends a copy together with a chain termination (CT) and keeps a pointer to P1. Subsequently, when processor P2 reads location X, the memory sends a copy to the cache of processor P2 along with a pointer to the cache of processor P1. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination acknowledges the invalidation of the chain.

**Figure 2-7 Chained Directory Cache-Coherence Protocol**
3 CACHE COHERENCE IN CMP

The last decade saw the emergence of parallel super computers while this decade saw the emergence of single-chip multiprocessors (CMP). The designs of CMP dealt with many issues regarding the need for high bandwidth, low latency, interconnection networks and very light weight inter-processor communication protocols.

A CMP integrates two or more complete processors on a single chip [URS03]. Each processor is duplicated and operates independently. This chapter gives a brief review of a number of CMPs and the cache coherence protocols they use to maintain data consistency in their different cache levels. These CMPs are: Stanford Hydra, Compaq Piranha, IBM POWER4, IBM POWER5, Raw and Raptor processors.

3.1 The Stanford Hydra CMP

3.1.1 Hydra CMP Composition

The Hydra CMP is built by the integration of four MIPS-based processors with their primary caches on a single chip together with a shared secondary cache [Ham00]. MIPS developed one of the first commercially available RISC chip sets. The system was inspired by an experimental system also using the name MIPS developed at Stanford [COA03] [Hen84].
As shown in Figure 3-1, each core has its own Level 1 (L1) instruction and data caches and all the cores share a large on-chip secondary cache. The processors support normal loads and stores together with the MIPS load-locked (LL) and store-conditional (SC) instructions for implementing synchronization.

Two buses, READ and WRITE connect the processors and the secondary cache together. The buses are logical buses as the physical wires are divided into multiple segments using repeaters and pipeline buffers where needed to avoid the slow down of core clock frequencies.

The READ bus is used to move data between the processors, secondary cache and external interface to off-chip memory. The READ bus is wide enough to carry a cache block in one clock cycle.

The WRITE bus is narrower and is used for the writes made by all 4 cores to the secondary cache. This allows the finite-state machines to be maintained in the secondary cache. The bus is pipelined to allow single-cycle occupancy by each write.
The cache-coherence protocol used to maintain coherency among the primary caches is a simple invalidation-only coherence protocol.

This design was tested by a set of parallel applications against one of the MIPS processors occupying the same chip area and a speedup was shown, but in not-easily parallelized applications the story was different. So a thread-level speculation unit was added to the design.

The bus design is helpful up to eight processors but more than that a special interconnection is needed.

3.1.2 Cache Coherence in Hydra CMP

The WRITE bus permits Hydra to use a simple invalidation protocol to maintain coherent primary caches.

Writes broadcast over the bus invalidate copies of the same line in primary caches of the other processors. No data is ever permanently lost due to these invalidations because the permanent machine state is always maintained in the secondary cache.

The WRITE bus also enforces memory consistency in Hydra. Since all writes must pass over the bus to become visible to the other processors, the order in which they pass is globally acknowledged to be the order in which they update shared memory.

3.2 The Compaq Piranha CMP

Piranha is a research prototype being developed at Compaq (jointly by Corporate Research and Non-Stop hardware development) to explore chip multiprocessing architectures targeted at parallel commercial workloads especially Online Transaction
Processing (OTP) with a small team, modest investment and a short design time [Bar00].

3.2.1 Piranha CMP Architecture

Figure 3-2 shows the block diagram of a single chip processing node.

![Figure 3-2 Block diagram of a single-chip Piranha processing node](image)

It consists of 8 processing cores (CPU’s) where each CPU is directly connected to dedicated Level 1 instruction (iL1) and Level 1 data (dL1) caches. These Level 1 caches interface to other modules through the *Intra-chip Switch* (ICS). On the other side of the ICS is a logically shared second level cache (L2) that is interleaved into 8 separate modules each with its own controller, on-chip tag and data storage.

Attached to each L2 cache is a memory controller (MC) which directly interfaces to one bank of up to 32 direct Rambus DRAM chips. Each memory bank provides bandwidth of 1.6 Gbytes/sec leading to an aggregate bandwidth of 12.8 Gbytes/sec.
Also connected to the ICS are two protocol engines: Home Engine (HE) and Remote Engine (RE) which support shared memory across multiple chips. The interconnect that links multiple Piranha chips consist of:

1. Router (RT).
2. Input Queue (IQ).
3. Output Queue (OQ).
4. Packet Switch (PS)

Total interconnect bandwidth (In/Out) for each Piranha chip is 32 Gbytes/sec.

3.2.2 Cache Coherence in Piranha CMP

Piranha uses 64 KB two-way, set-associative, blocking caches with virtual indices and physical tags. The L1 cache modules include tag compare logic, instruction and data Table Look-aside Buffers (TLB) (256 entries, 4-way associative) and a store buffer (data cache only). A 2-bit state field is maintained per cache line corresponding to the four states of a typical MESI protocol.

The second-level cache (L2) is a 1 MB unified instruction/data cache which is physically partitioned into eight banks and is logically shared among all CPU’s. The L2 banks are interleaved using the lower address bits of a cache line’s physical address (64-byte line). Each bank is 8-way set associative and uses a round-robin or LRU replacement policy if an invalid block is not available.
There are two cache-coherence protocols:

3.2.2.1 Inter-chip coherence protocol

To avoid the use of snooping at L1 caches, a duplicate copy of the L1 tags and state is kept at the L2 controllers. Each controller maintains tag/state information for L1 lines that map to it given the address interleaving. The total overhead for the duplicate L1 tag/state across all controllers is less than 1/32 of the total on-chip memory.

In order to lower miss latency and best utilize the L2 capacity, L1 misses that also miss in the L2 are filled directly from memory without allocating a line in the L2. The L2 effectively behaves as a very large victim cache that is filled only when data is replaced from the L1s. Clean lines from L1 may also cause a write-back to the L2, when they are replaced.

To avoid unnecessary write-backs when multiple L1s have copies of the same line, the duplicate L1 state is extended to include the notion of ownership. The owner of the line is either the L2 (when it has a valid copy), an L1 in the exclusive state or one of the L1s when there are multiple sharers. In case of multiple sharers, a write-back happens only when an owner L1 replaces the data.

3.2.2.2 Intra-chip coherence protocol

The L2 controllers are responsible for enforcing coherence within a chip. Each controller has complete and exact information about the on-chip cached copies for the subset of the lines that map to it. On every L2 access, the duplicate L1 tag/state and the tag/state of the L2 itself are checked in parallel. Therefore the
ina-chip coherence has similarities to a full-map centralized directory which is stored in DRAM and accessed through the memory controller.

3.3 IBM POWER4 CMP

IBM POWER4 was designed and developed in several IBM development laboratories to deliver a server that would redefine what was meant by the term server [Ten01]. POWER4 was designed according to a set of principles that were met by the designers as shown in the following section.

3.3.1 Design Principles

- **SMP optimization**

  The server should be designed for high throughput and multi-tasking environments. The system must be optimized for SMP operation where a large number of transistors could be used to improve total system performance.

- **Full system design approach**

  The process technology, packaging and micro-architecture was designed to allow software to exploit them. The processor core was designed to fit effectively in this environment. The entire system including the processor, memory and I/O chips were designed together.
• **Very high frequency design**

The system was designed to permit system balance to be preserved as technology improvements became available allowing high processor frequencies to be delivered.

• **Leadership reliability, availability and serviceability**

Reliability, Availability and Serviceability together are grouped into the term (RAS). RAS is a very important attribute in servers design because outages of any kind are not tolerated. RAS is met in POWER4 by transforming hard machine stops into synchronous machine interrupts to software to allow it to circumvent problems if it could.

• **Designed to execute both commercial and technical applications**

The system should handle a varied and robust set of workloads. The system should satisfy high performance computing requirements with its high bandwidth demands and commercial requirements with its data sharing requirements.

• **Maintain binary compatibility for both 32-bit and 64-bit applications**

For the sake of customer investments the system should have the hardware to support both 32-bit and 64-bit applications.
3.3.2 POWER4 Chip Composition

The composition of the POWER4 chip is shown in Figure 3-3

![Figure 3-3 Power4 chip logical view [Ten01]](image)

The POWER4 chip has two processors on board. Each processor has the various execution units and first level instruction and data caches. The two processors share a unified second level cache also onboard the chip through a Core Interface Unit (CIU). The CIU is a crossbar switch between the L2 and the two processors. The L2 cache is implemented as three separate autonomous cache controllers. Each cache controller can operate concurrently and feed 32 bytes of data per cycle.

The CIU connects each of the three L2 controllers to either the data cache or the instruction cache in either of the two processors. Additionally the CIU accepts stores from the processors across 8-bytes wide buses and sequences them to the L2
controllers. Each processor has associated with it a Non-Cacheable unit (NC) responsible for handling instructions, serializing functions and performing any non-cacheable operations in the storage hierarchy. Logically this is part of the L2.

The directory for another L3 cache and logically its controller are also located on the same chip but the L3 cache is on another chip. The Fabric Controller unit is responsible for controlling data flow between the L2 and L3 controllers for the chip and for Power4 communication. The GX controller is responsible for controlling the flow of information in and out of the system. Typically, this would be the interface to an I/O drawer attached to the system. But with the POWER4 architecture, this is also used where an interface to a switch is attached for clustering multiple POWER4 nodes together.

Also included on chip some functions called Pervasive functions. These include trace and debug facilities used for First Failure Data Capture, Built-in Self Test (BIST) facilities, Performance Monitoring Unit, an interface to the Service Processor (SP) used to control the overall system, Power On Reset (POR), sequencing logic and Error Detection and Logging Circuitry.

Four POWER4 chips can be packaged on a single module to form an 8-way SMP. Four of such modules can be interconnected to form a 32-way SMP.

To accomplish the interconnection each chip has five primary interfaces. To communicate to other POWER4 chips on the same module, there are logically four 16-byte buses. Physically these four buses are implemented with six buses, three on and three off. To communicate to POWER4 chips on other modules, there are two 8-byte buses one on and one off.
Each chip has its own interface to the off-chip L3 across two 16-byte wide buses, one on and one off operating at one third the processor frequency.

To communicate with I/O devices and other compute nodes, two 4-byte wide GX buses, one on and one off operating at one third the processor frequency are used.

Finally each chip has its own JTAG interface to the system service processor. Bus frequencies scale proportionately with processor frequency to maintain system balance.

3.3.3 Cache Hierarchy and Coherence in POWER4

The storage hierarchy in POWER4 consists of three levels of cache and the memory subsystem. The first and second levels L1 and L2 of the hierarchy are on board and the directory of the third level L3 are on chip while L3 cache itself is off chip. Table 3-1 shows the Power4 storage hierarchy organization and size.

<table>
<thead>
<tr>
<th>Component</th>
<th>Organization</th>
<th>Capacity per chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction Cache</td>
<td>Direct map, 128-byte line managed as four 32-byte sectors.</td>
<td>128 KB (64 KB per processor)</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>2-way, 128-byte line</td>
<td>64 KB (32 KB per processor)</td>
</tr>
<tr>
<td>L2</td>
<td>8-way, 128-byte line</td>
<td>~ 1.5 MB</td>
</tr>
<tr>
<td>L3</td>
<td>8-way, 512-byte line managed as 4 128-byte sectors</td>
<td>32 MB</td>
</tr>
<tr>
<td>Memory</td>
<td>---------------</td>
<td>0-16 GB</td>
</tr>
</tbody>
</table>

Table 3-1 Storage hierarchy organization and size [Ten01]
L1 Caches

The L1 instruction cache is single-ported and capable of either one 32-byte read or write each cycle.

The L1 data cache is triple ported capable of two 8-byte reads and one 8-byte write per cycle with no blocking. L1 data cache reloads are 32-bytes per cycle.

The L1 caches are parity protected so when there is parity violation the data is retrieved from the L2 cache.

All data stored in the L1 data cache is also available in the L2 cache and it can be in one of two states:

- **I (invalid state):** The data is invalid.
- **V (valid state):** The data is valid.

L2 Cache

L2 cache is unified and shared across the two processors on the POWER4 chip. Figure 3-4 shows a logical view of the Power4 L2 cache.
As shown, L2 cache is implemented as three identical controllers. Cache lines are hashed across the three controllers.

The L2 cache directory is implemented in two redundant 8-way set associative parity protected arrays. Redundancy provides a backup capability and two non-blocking read ports to permit snoops to proceed without causing interference to load and store requests. A pseudo LRU replacement algorithm is implemented for cache lines replacement.
Most of the L2 cache management is handled by four coherency processors in each controller.

The L2 cache implements an enhanced version of the MESI coherency protocol supporting seven states as follows:

- **I (Invalid):**
  
  Data is invalid. This is the initial state of L2 entered from a Power On Reset or a snoop invalidate hit.

- **SL (Shared Locally):**
  
  Data is valid. The cache line may also be valid in other L2 caches. From this state the data can be sourced to another L2 on the same module via intervention. This state is entered as a result of a core L1 data cache load request or instruction fetch request that misses in the L2 and is sourced from another cache or from memory when not in other L2 caches.

- **S (Shared):**
  
  Data is valid. The cache line may also be valid in other L2 caches. In this state the data cannot be sourced to another L2 via intervention. This state is entered when a snoop read hit from another processor on a chip on the same module occurs and the data and tag were in the SL state.
• **M (Modified):**

Data is valid. The data has been modified and is exclusively owned. The cache line cannot be valid in any other L2 cache. From this state, data can be sourced to another L2 in a chip on the same or remote module via intervention. This state results from a store performed by one of the cores on the chip.

• **Me (Exclusive):**

The data is valid. The data is not considered modified but exclusive to this L2 cache. This state is entered as a result of one of the cores on the chip asking for a reservation when data is sourced from memory or for a cache line being pre-fetched into the L2 that was sourced from memory.

• **Mu (Modified Unsolicited):**

Data is valid. Same as M-state with the exception that the state is entered as a result of one of the cores on chip asking for a reservation when data is sourced from another L2 in M-state or for a cache line being pre-fetched into the L2 that was sourced from another L2 in M-state.

• **T (Tagged):**

Data is valid. The data is modified with respect to the copy in memory. It has also been sourced to another cache.
L3 cache

The L3 cache is 8-way associative organized in 512 byte blocks, with coherence maintained on 128-byte sectors for compatibility with the L2 cache. The coherency protocol includes 5 states for each of the 128-byte sectors as follows:

- **I (Invalid):**
  
  Data is invalid.

- **S (Shared):**
  
  The data is valid and L3 can source data to the L2 that is caching data for.

- **T (Tagged):**
  
  The data is valid. The data is modified relative to the copy stored in memory. The data may be shared in other L2 or L3 caches.

- **Trem (Tagged remote):**
  
  It is the same as T-state, but the data was sourced from memory attached to another chip.

- **O (pre-fetch data):**
  
  The data in L3 is identical to the data in memory. The data was sourced from memory attached to this L3. The status of the data in other L2 or L3 is unknown.
3.4 IBM POWER5 CMP

POWER5 is the next generation of the POWER4 CMP. POWER5 implements 2-way simultaneous multi threaded (SMT) on each of the chip’s two processor cores [KST04].

POWER5 supports 1.875 Mbytes of on-chip L2 cache and 36 Mbytes off-chip L3 cache. The L3 cache operates as a backdoor with separate buses for reads and writes that operate at half the processors speed.

3.4.1 Cache Implementation

The L2 cache is implemented as three identical slices with separate controllers for each similar to POWER4 implementation. The L2 slices are 10-way set associative with 512 congruence classes of 128 bytes lines each. The data’s real address determines which L2 slice the data is cached in. Each L2 controller can be accessed by any of the two processor cores.

The directory for the off-chip L3 cache is also implemented on the chip. This enables the processor to check whether the data is in the L3 cache before experiencing the delays of accessing the off-chip cache. And to reduce memory latencies, the memory controller is implemented on chip also. This eliminated driver and receiver delays to an external controller.
3.5 The Raw Microprocessor

The Raw microprocessor was designed and developed in MIT. It has 122 million transistors, executes 16 different load, store and integer or floating point instructions every cycle; controls 25 Gbytes/s of input/output (I/O) bandwidth and has 2 Mbytes of on-chip distributed L1 static RAM providing on-chip memory bandwidth of 57 Gbytes/s [Tay02].

Taylor et al. designed Raw using a scalable Instruction Set Architecture (ISA) that provided a parallel software interface to enable the programmer or compiler to program every resource on the chip including gates, wires and pin resources.

The silicon area of the Raw processor is divided into 16 identical programmable tiles, where each tile contains:

1. One static communication router.
2. Two dynamic communication routers.
3. An eight stage, in-order, single-issue, MIPS-style processor.
4. A four-stage, pipelined, floating point unit.
5. 32 Kbytes data cache.
6. 96 Kbytes of software managed instruction cache.

Each tile was adjusted in such a way that the signal travels across the tile in one clock cycle.

The tiles are interconnected using a 32-bit full-duplex on chip-networks. Two networks are static where the routes are specified at compile time and two are dynamic where the routes are specified at run time as shown in Figure 3-5. Each tile
connects only to its four neighbors so the length of the longest wire in the system is no longer than the length or width of a tile.

![Figure 3-5 On-chip interconnects in Raw. The Raw microprocessor comprises 16 tiles (a). Each Tile (b) has computational resources and four networks each with eight point-to-point 32 bit buses to neighbor tiles [Tay02]](image)

All these tiles are packaged in a 1,657 pins’ ceramic-column grid array package that provides 14 full-duplex, 32-bit, 7.5-Gbps I/O ports at 225 MHz for a total of 25 Gbytes/s of bandwidth.

The Raw operating system allows both time and space multiplexing of processes by allocating a rectangular shaped number of tiles proportional to the amount of computations required by that process. When the operating system context switches in a given process it finds a contiguous region of tiles that corresponds to the dimension of the process and resumes the execution of the physical thread.
The main target in the design of Raw was to provide low-latency communication for efficient execution of software circuits and parallel, scalar codes besides providing a scalable version of the standard toolbox of architectural constructs such as data and instruction virtualization, caching, interrupts, context switches, address spaces, latency tolerance, and event counting. Raw achieves this goal by the compute processor, static router and dynamic routers.

**The Compute Processor:**

The main target in the compute processor’s design is to tightly integrate coupled network interfaces into the processor pipelines. The most common network interfaces are memory mapped, special instructions for sending and receiving and register mapped that doesn’t need special send and receive commands. The design of Raw used the register mapped and integrated this network interface directly into the bypass paths of the processor pipeline. A READ from a register will pull an element from an input FIFO buffer while a WRITE will send the data word out onto the network. Each output FIFO buffer connects to each pipeline stage. The output buffers pull the oldest value out of the pipeline as soon as it is ready rather than just at the write-back stage of through the register file.

**The Static Router:**

The two static networks are used to route values between tiles for software circuits and parallel scalar codes. The static networks provide ordered, flow controlled and reliable transfer of single-word operands and data streams between the tiles, functional units. The operands are delivered in order so that the instructions issued by the tiles are operating on the correct data. Flow control of operands allows the
program to remain correct in the face of unpredictable architectural events such as cache misses and interrupts.

The static router is a five-stage pipeline which controls two routing crossbars so it controls two networks. Each crossbar routes values between seven entities. These entities are: the static router pipeline; the north, east, south and west neighbor tiles; the compute processor; and the other crossbar. There should exist a corresponding instruction in the instruction memory of each router in the word’s path for each word sent between the tiles via the static network. The instructions are programmed at compile time and cached as the instructions of the compute processor. Since static routers know what route will be performed long before the word arrives routing can be pipelined and this helped in reducing the latency which is critical for instruction level parallelism in scalar codes.

The Dynamic Networks:

Dynamic networks were designed to support long data streams and not scalar transport. Raw has two dynamic networks which are dimension-ordered and use wormhole routing to route the packets flowing through them. To route a message on one of these networks the user injects a single header word that specifies the destination tile (or I/O), a user field and the length of the message.

The major concern with dynamic networks during their design was deadlock. There are two ways to deal with deadlocks; either by avoidance or recovery. The memory network uses deadlock avoidance while the general network uses deadlock recovery. If the general network deadlocks, an interrupt routine is activated that uses the memory network to recover.
3.6 RAPTOR Single Chip Multiprocessor

RAPTOR is a single chip multiprocessor developed by Lee et al. in 1999. RAPTOR was developed to exploit thread-level parallelism [Lee99]. Since software trends favor using multithreaded programming, an operating system service can run independent threads on independent processors simultaneously.

3.6.1 RAPTOR Hardware Architecture

RAPTOR is composed of four processor units (PU), a graphics coprocessor unit (GCU), an inter-processor bus unit (IBU), an external cache control unit (ECU), a multiprocessor control unit (MCU) and a memory interface unit (MIU). Figure 3-6 shows a block diagram of RAPTOR.

![Figure 3-6 Raptor Block Diagram][1]

**Figure 3-6** Raptor Block Diagram [Lee99]
Each PU implements the 64-bit SPARC –V9 instruction set architecture and includes an 8 Kbytes instruction (I) cache and an 8 Kbytes data (D) cache.

The PU’s are connected through bus (IBU) which supports data consistency through snoop based cache coherency protocol. There is no internal second level cache but the ECU supports a direct connection to a large external second level cache (4 Mbytes), which can provide the multiprocessor-ready external interface.

The MCU distributes and processes inter-processor interrupts. It also handles hardware support for synchronization of processor units.

The MIU handles all the interfacing transactions from/to the system, including main memory access, interrupt processing and external snoop processing.

3.6.2 Raptor Cache Coherence

To enhance the efficiency of snooping, write-once protocol is used for tracking the first level cache line status.

For a line fill to the first level cache, the line status is always set to SHARED. The second level cache adopts the well-known MESI protocol. All the caches are equipped with the dual tag which provides the pure concurrent snoop operation.
4 LOCK-BASED CACHE COHERENCE PROTOCOL FOR CMP

Chip multiprocessors that share one or more levels of on-chip cache allow inter-
processor communication between the CMP cores as explained by Olukotun and
Hammond in [OH05]. The most common form of cache sharing is sharing the on-chip
L2 cache as shown in Figure 4-1. Since the CMP cores are connected together by the
internal bus, snoopy cache coherence protocols apply. Synchronization among the
CMP cores is done by the operating system and not by the cache coherence protocol.
Integrating the synchronization in the cache coherence protocol to be done by the
hardware might be a good solution to decrease the overheads of the operating system.

This work introduces the Lock-based Cache Coherence Protocol (LCCP) which
integrates synchronization within the MESI snoopy cache coherence protocol. The
design of the (LCCP) is based on introducing a LOCK state in the MESI coherence
protocol. The protocol will be applied to maintain consistency between the individual
L1 caches and support synchronization in case of the Read-Modify-Write data. This
design optimizes the traffic between the L2 cache and the individual L1 caches. The
protocol uses memory write-back policy to reduce the traffic between the L2 cache
and the external memory. The following sections describe the details of LCCP.
4.1 Synchronization

Cache coherence protocols are optimized for memory access patterns of regular data; however data patterns for locked situations in synchronization cannot be treated in the same way due to their different nature. This nature is not accounted for in the normal cache coherence protocols.
Typically, a lock variable is obtained by a process through a read-modify-write operation at the beginning of a critical section. The processor releases the lock after a successful write operation to the lock variable at the end of the critical section.

Many synchronization mechanisms have been introduced. Most synchronization operations use an atomic read-modify-write primitive. Not all architectures provide a synchronization primitive but they provide an instruction that automatically swaps a value in a specific register with a memory location. An additional condition was made by researchers to this instruction where the swap occurs if the value read from memory is equal to a specified value. A more complex primitive as Fetch & Op can perform a simple operation such as an arithmetic operation atomically to the value in memory.

Another synchronization technique used by many microprocessors is to provide the Load-Linked (LL) / Store-Conditional (SC) pair to implement the atomic read-modify-write. The LL/SC instruction pair was introduced by Jensen, Hagensen and Broughton in [JHB87]. The LL instruction loads a memory location to a specific register in the processor. The processor then performs a sequence of operations and writes to the same memory location via a SC operation. The SC instruction is considered successful if no other processor has written to this same memory location. In case of a failure the entire process is retried again. Figure 4-2 shows the implementation of the LL/SC instruction pair.
The LL/SC synchronization method has been adapted to several architectures such as Load-Locked/Store-Conditional in the Alpha processor, Load-and-Reserve/Store-Conditional in the IBM PowerPC [TEN01] and Load-Linked/Store-Conditional in the MIPS [KH92].

One problem with LL-SC is that store-conditional may send invalidations or updates if it fails. In that case two processors may keep invalidating or updating each other and failing. This situation is called *Livelock*.

Livelock in an invalidation-based cache coherence system is caused when all processors attempt to write to the same memory location at the same time. When the cache block is loaded into the cache in the modified state for store, and before the processor is able to complete its store, the block may be invalidated by another processor attempting to load the same cache block for store. The first processor’s
store attempt will miss and it will load the cache block again. This situation can repeat indefinitely.

Livelock situations should be avoided when dealing with cache blocks containing synchronization variables. A new state has been added to the cache coherence protocol called the LOCK state to resolve Livelock. On detecting a Load-Linked command, the processor reads the cache block in the LOCK state so that no other processor can read or modify it until the lock is released after a store-conditional. Requests from other processors to access the locked block will be stored in a waiting queue until the lock is released.
4.2 Finite State-Machine of LCCP

Figure 4-3 State machine diagram for the Lock-based cache coherence protocol. PrRdL stands for Processor-Read-Locked and PrWrC stands for Processor-Write-Conditional.

Figure 4-3 shows the finite-state-machine (FSM) diagram of LCCP (lock-based cache coherence protocol). The solid line is considered as an action by the processor, while a dashed line is an action caused by the bus.

The FSM has 5 states:

Invalid (I):

The data in the cache line is not valid. This is the initial state of the lines in the L1 cache.
Shared (S):

There exists more than one copy of the same cache line in different L1 caches. All copies are consistent with each other and with the copy in main memory. The cache line is shared for reading.

Exclusive (E):

When a processor loads (reads) a cache line for the first time from main memory, it is loaded in the *Exclusive State*. This means that there is no other copy in the L1 cache of any other processor.

Modified (M):

When a processor stores (writes) data in the cache line, the line is said to be modified and the state goes to the *Modified State*. This cache line can be the only copy in the network of processors.

Locked (L):

This is the state when the processor loads a cache line that contains a critical section by load-linked (LL) instruction.

The processor issues four types of requests: reads (PrRd), writes (PrWr), loads-locked (PrRdL) and writes-conditional (PrWrC).
• Processor Read (PrRd):

This request is issued when the processor executes a load instruction. The processor checks the requested data in its own L1 data cache.

• Processor Write (PrWr):

This request is issued when the processor executes a store instruction. The variable needs to be in the processor’s L1 cache before the store instruction can be executed.

• Processor Load-Locked (PrRdL):

This request is issued when the processor executes a load-linked instruction.

• Processor Store-Conditional (PrWrC):

This request is issued when the processor executes a store-conditional instruction.

The reads and writes could be to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block. The bus allows the following transactions:

• Bus Read (BusRd):

This transaction happens when a processor misses a read by PrRd and the processor expects data as a result. The cache controller puts the address on the
bus and asks for a copy to read. Another cache or memory will provide that copy.

- **Bus Read Exclusive (BusRdX):**

  This transaction is generated when a processor misses a PrWr to a cache block. The cache controller puts the address on the bus and asks for an exclusive copy to modify. All other caches are invalidated.

- **Bus Read Locked (BusRdL):**

  This transaction is generated when a processor misses a PrRdL for a cache block. The cache controller puts the address on the bus and asks for a copy to lock until it writes to it. All other caches are invalidated.

### 4.3 Cache Coherence

#### 4.3.1 Read Miss (no other copy is found)

If a read-miss occurs in the Level 1 cache (L1$) of one of the processor cores, the processor will check if another processor has a copy of the cache line. Since this is the first thread requesting that cache block, there will be no other copy in any of the other L1 caches. If the line is found in the L2$ with state MODIFIED, the processor will get a copy in its L1$ in the EXCLUSIVE state. The state of the copy in the L2$ will be SHARED.
4.3.2 Read Miss (one or more copies are found)

If there is a cache miss in L1$ of one processor and that cache block was found in another processor’s L1$. If this line was in the EXCLUSIVE state, then this processor will provide a copy of this line in the SHARED state to the requesting processor and will change its line state to SHARED. The L2$ will be SHARED also. If this line was in the MODIFIED state, the processor will change the state of the line to SHARED and provide one copy to the requesting processor and another one to L2$. Both copies will be in the SHARED state.

4.3.3 Write hit

In case there is a write hit in one of the L1$ and the status of the line is EXCLUSIVE, the write will occur and the state of the line will be changed to MODIFIED. A copy will be written through to the L2$ with the state MODIFIED.

If the state of the line is MODIFIED, the write will occur and the L2 copy will be updated with the MODIFIED state.

If the state of the block is SHARED, the processor will write the line in the L2$ in the MODIFIED state. Being MODIFIED the L2$ will broadcast the block to all the L1$. If the L1$ has a shared copy it will update it, otherwise it will ignore the broadcast.

4.3.4 Write miss

In case of a write miss from the L1$ and the L2$, the L1$ will get a copy of the line from main memory in the EXCLUSIVE state. The Write occurs, the block changes state to MODIFIED and is written through to L2$ in the MODIFIED state. In case the
block is in SHARED or MODIFIED states in L2$, the processor writes directly to L2$ and the cache block is broadcast in the network to the other L1$.

4.3.5 Read-Modify-Write miss

In case of a read miss in a read-modify-write operation in both the L2$ and the L1$, the L1$ gets a copy from main memory in the LOCKED state. If the cache block is in the SHARED or MODIFIED states in the L2$, the L2$ will broadcast invalidation signal to all the processors. L2$ will provide the cache block to the requesting L1$ which will read it in the LOCKED state and the L2$ will invalidate its copy and go to INVALID state.

The cache block will be modified and then a write hit occurs for the store-conditional. After the write the cache block changes its state from LOCKED to MODIFIED and a copy will be written through to the L2$ in the MODIFIED state.

If a read request for the locked block comes from another processor, it will find that the block is in the LOCKED state and will wait for its release in a wait-table. Figure 4-4 shows a schematic of the wait-table.

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Line Tag</th>
<th>Processor ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Line Tag</td>
<td>Processor ID</td>
</tr>
<tr>
<td>Valid</td>
<td>Line Tag</td>
<td>Processor ID</td>
</tr>
<tr>
<td>Valid</td>
<td>Line Tag</td>
<td>Processor ID</td>
</tr>
</tbody>
</table>

**Figure 4-4** Structure of the lock-release wait-table
Each entry in the wait-table consists of:

- **Valid Bit:**

  It indicates that there is a valid tag and processor ID’s waiting on that tag.

- **Line Tag:**

  It contains the tag of the cache line that contains the synchronization variable.

- **Processor ID:**

  This is a bit-vector of the processors waiting to acquire the lock variable.

Figure 4-5 shows the structure of the Processor ID field in an entry in the wait-table.

![Figure 4-5 Structure of the Processor ID field](image)

As shown in figure 4-5, P1 and P3 are waiting to acquire the lock variable. The length of the Processor ID is equal to P bits where P is the number of processor cores on the CMP. The size of the wait-table is equal to the size of the L1 cache of a single processor core.

### 4.4 Architecture of the CMP with LCCP

Figure 4-6 shows the architecture of the CMP implementing the LCCP. The CMP has the same architecture as that shown in Figure 4-1 with the addition of the wait-table.
The wait-table is connected directly to the internal bus and is accessed by all the processor cores via this bus. It serves as a queuing mechanism for pending locked line accesses. Each entry in the wait-table consists of the locked line tag and the I.D of the processor that is requesting that line. As locked line requests are serviced, their respective entries are removed from the wait-table.

When a processor finishes an SC instruction successfully, the processor checks with wait-table controller whether another processor is requesting that line tag. If yes, it provides the copy to the requesting processor and L2 cache. The state of the line remains in the LOCK state. If not the processor changes the state of the line from the LOCK state to the MODIFIED state.
4.5 Synchronization using LCCP

An example illustrating how synchronization is performed using LCCP is shown in Table 4-1. In this example three processors P0, P1 and P2 compete to acquire the Lock.

In step 1, processor P0 has acquired the lock variable while processors P1 and P2 are competing to acquire the same lock. Processor P1 entered the wait-table followed by processor P2.
<table>
<thead>
<tr>
<th>Step</th>
<th>Processor P0</th>
<th>Processor P1</th>
<th>Processor P2</th>
<th>Coherence state of lock</th>
<th>Bus activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Tries to load-locked, enters the wait-table</td>
<td>Tries to load-locked, enters the wait-table after P1</td>
<td>Locked</td>
<td>Bus services</td>
</tr>
<tr>
<td>2</td>
<td>Release lock</td>
<td>Wait-table</td>
<td>Wait-table</td>
<td>Locked (P0,P1,P2)</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>Check wait-table</td>
<td>Wait-table</td>
<td>Wait-table</td>
<td>Locked (P0,P1,P2)</td>
<td>Bus services; data is copied to P1</td>
</tr>
<tr>
<td>4</td>
<td>Acquire lock</td>
<td>Wait-table</td>
<td>Locked (P1,P2)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Release lock</td>
<td>Wait-table</td>
<td>Locked (P1,P2)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Check wait-table</td>
<td>Wait-table</td>
<td>Locked (P1,P2)</td>
<td>Bus services; data is copied to P2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Acquire lock</td>
<td>Locked (P2)</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Release lock</td>
<td>Locked (P2)</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Check wait-table</td>
<td>Locked (P2)</td>
<td>Bus services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Modified (P2)</td>
<td>Bus services; write to memory</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4-1 Illustration of synchronization using LCCP**

The cache line containing the lock variable is in the *lock state*. In step 2, processor P0 releases the lock. In step 3, the wait-table is checked and it was found that processor P1 may acquire the lock as indicated in step 4. In step 5, processor P1 releases the lock and in step 6 the wait-table is checked again for waiting processors. Processor P2 is next in line for lock acquisition and acquires it in step 7. In step 8 processor P2 releases the lock and in step 9 the wait-table is checked. Since no processor is waiting the state of the cache line is changed to the modified state.
5 EXPERIMENTAL METHODOLOGY

5.1 Multiprocessor Simulators

A multiprocessor simulator was needed to simulate this work. There are a number of multiprocessor simulators.

5.1.1 RSIM

RSIM (Rice Simulator for ILP Multiprocessor) was developed at Rice University to study the performance of ILP (Instruction Level Parallelism) exploited in processors such as the MIPS R10000, Hewlett-Packard PA8000 and the Intel Pentium Pro [Hug02].

Each node of the simulated multiprocessor system consists of a processor and cache hierarchy along with part of the physical memory, its associated directory and a network interface. A split – transaction bus connects the secondary cache, memory and the directory module and network interface as shown in figure 5-1.

RSIM was written in a modular fashion using C++ and C for extensibility and portability. It can run on top of Sun systems running Solaris (up to version 2.8), a Hewlett-Packard Convex Exemplar running HP-UX version 10, an SGI Power Challenge running IRIX 6.2 and X86 running Linux.

The major advantage of RSIM is that it is very good in simulating processors exploiting ILP. A lot of work should have been done to convert it from an ILP simulator to the multiprocessor simulator needed to simulate this research. This work was similar to building the simulator from scratch.
5.1.2. Simics

Simics [SIM04] is a commercial simulator developed by Virutech AB in Stockholm [Ogu04]. It is a detailed simulator that runs on a number of operating systems. Currently it can run Solaris, Linux and WindowsXP. Simics is able to simulate the x86, UltraSparc, MIPS, Alpha and PowerPC target platforms. There are also a wide variety of ports or host systems that the simulator can run on.

Simics is a functional simulator so it simulates at the instruction set level. Currently Simics can simulate several uniprocessor nodes, with one restriction being the nodes must be of the same basic target architecture. However, different processing nodes
can be connected together to form a heterogeneous network through the “Simics Central”.

Simics Central is a process whose task is to connect and distribute communication and data traffic between the different processing nodes.

New processor and target models are defined in a special language developed at Virutech called SimGen. A specification written in this way is compiled into a simulator in C-code, which can then be instantiated and used in simulation.

For cache and memory statistics, timing models for memory references can be added. From what was mentioned previously, it seemed that getting cache statistics would be limited.

Simics has its greatest advantage in commercial development where speed of simulation is more important than detailed statistics information, that are more interesting in research work.

5.1.3. MP_Simplesim

MP_Simplesim [Man01] is a multiprocessor enhancement of the SimpleScalar toolset [BA97]. SimpleScalar is one of the many simulation tool sets that have been developed for computer architecture research. It has become an important tool in the research community and is widely used. SimpleScalar is efficient, flexible and includes support for fast functional simulation, cache simulation and detailed superscalar simulation.
SimpleScalar is a uniprocessor simulator and does not include support for multiprocessor simulation. The enhancements done in MP_Simplesim include a fast functional multiprocessor simulator as well as a multiprocessor cache simulator.

![Figure 5-2 MP_Simplesim Architecture Model](image_url)

Figure 5-2 shows the original MP-Simplesim architecture model. Each processor, P1, P2,.. Pn has its own L1 and L2 caches. The processors communicate with each other and with memory through the data bus.

The major advantage of MP_Simplesim is that it includes a runtime library that provides functions to create threads and perform synchronization using locks, barriers and semaphores. These functions set register arguments and execute a system call instruction that is intercepted by the simulator in order to perform the low-level thread management and synchronization task. Another advantage with MP_Simplesim is that it can perform detailed multiprocessor cache simulation to collect detailed results on cache coherence activities during the course of parallel execution.
Table 5-1 shows a summary of the advantages and disadvantages of the three simulators; RSIM, Simics and MP_Simplesim as they relate to this research.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSIM</td>
<td>- Written in a modular fashion in C/C++.</td>
<td>- Specially written to simulate multiprocessors exploiting ILP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Simulator is far from scope of research.</td>
</tr>
<tr>
<td>Simics</td>
<td>- Functional simulator that simulates at the instruction-set level.</td>
<td>- Getting cache statistics is limited.</td>
</tr>
<tr>
<td></td>
<td>- It can be run on many platforms and has ports to several targets</td>
<td>- More suitable for commercial development where speed of simulation is needed than for research work</td>
</tr>
<tr>
<td>MP_Simplesim</td>
<td>- It is based on SimpleScalar which is a popular uni-processor simulator.</td>
<td>- It supports only big-endean encoding for instructions so it is not running on x86 architectures.</td>
</tr>
<tr>
<td></td>
<td>- It supports thread creation and synchronization.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It provides detailed statistics for caches.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It can be modified easily due to the modular structure of the source code.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-1 Comparing RSIM, Simics and MP_Simplesim simulators

5.2 MP_Simplesim Modifications

As shown in figure 5-3 the multiprocessor model of the original MP_Simplesim was modified to the unified L2 cache model. Each processor element has its own L1 cache and all processors share a unified L2 cache.
For a chip multiprocessor (CMP), the processor elements, the unified L2 cache and the bus they use to communicate together are all contained within the chip. The CMP will use the external data bus to communicate with memory.

![Figure 5-3 Modified MP_Simplesim Architecture Model](image)

The instruction-set of MP_Simplesim does not support Load_locked (LL) and Store_Conditional (SC) instructions, so LL/SC instruction pair was added to the instruction set. The cache coherence protocol (MESI) was modified to accommodate the Lock state, and the wait-table was implemented. Appendix A shows the modified files of the original MP_Simplesim simulator.
5.3 The Benchmarks

The benchmarks used to simulate the experiments are from the SPLASH (Stanford Parallel Applications for Shared Memory) suite [SWG92] [SPLASH]. The benchmarks were developed at Stanford University to facilitate the evaluation of architectures that support a shared address space with coherent application [CS99].

It was replaced by the SPLASH-2 suite [SPLASH] where some applications were enhanced and new ones were introduced thus broadening the coverage of domains and characteristics substantially. The SPLASH-2 suite currently contains seven complete applications and five computational kernels. All the programs are written in C.

The following benchmarks are the ones that were used in this research:

5.3.1 Ocean

Ocean is a program that simulates currents in an ocean basin. It resembles many important applications in computational fluid dynamics [CS99]. Due to its properties it can also be considered as a representative of a wide range of applications that stream through large data structures and perform little computation at each data point.

The benchmark is implemented in two methods:

Non-contiguous partition allocation

This implementation implements the grids to be operated on with two-dimensional arrays. This data structure prevents partitions from being allocated contiguously, but leads to a conceptually simple programming implementation.
Contiguous partition allocation

This implementation implements the grids to be operated on with 3-dimensional arrays. The first dimension specifies the processor which owns the partition, and the second and third dimensions specify the x and y offset within a partition. This data structure allows partitions to be allocated contiguously and entirely in the local memory of processors that "own" them, thus enhancing data locality properties.

5.3.2 Barnes-Hut

Barnes-Hut application performs irregular, fine-grained, time-varying communication and data access patterns that are becoming increasingly prevalent even in scientific computing as it tries to model the galaxy [CS99]. It solves an n-body problem in which the major computational challenge is to compute the influences that n bodies in the system exert on one another.

5.4 Workflow

The design was tested by running Ocean in its two versions; contiguous and non-contiguous and Barnes-Hut benchmarks.
6 RESULTS AND ANALYSIS

This chapter discusses the results of the simulation experiments performed to measure the affect of *Locked Cache Coherence Protocol* (LCCP) on the performance of CMP using the modified MP_Simplesim simulator.

All experiments were performed on a Sun Ultra10 workstation with a 300 MHz UltraSPARC-IIi processor, 2 Mbytes of secondary cache memory and 128 Mbytes of main memory.

The benchmarks discussed before were chosen to evaluate the performance of LCCP. The benchmark respective problem sizes are shown in Table 6-1

<table>
<thead>
<tr>
<th>Application</th>
<th>Type</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ocean</td>
<td>Study of ocean movements</td>
<td>130x130 ocean grid</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>Study of the influence of N bodies in space on each other</td>
<td>16-K particles, Θ = 1.0, 3 time-steps</td>
</tr>
</tbody>
</table>

*Table 6-1* Benchmark types and data sets.

These benchmarks were chosen to test the proposed LCCP protocol. The gcc compiler, version 2.8.0, was used with optimization level -O3 to generate native SPARC code for the simulator. The gcc compiler, version 2.6.3, was used with the optimization level -O2 to generate SimpleScalar code for the benchmark programs to be simulated.
6.1 MP_Simplesim Verification

The original MP_Simplesim was compiled and run and its results were verified with the results in [Man01]. This verification is shown in Table 6-2.

Table 6-2 MP_Simplesim verification results

Table 6-2 (b) shows the results of running ocean non-contiguous-partitions with a problem size of 130x130, L1 cache size of 8 Kbytes, L2 cache size of 256 Kbytes and a common cache line size of 16 bytes. The results were verified with Table 6-2 (a) obtained from [Man01].
6.2 Experimental Setup

Table 6-2 shows the configurations of the L1 cache of each processor element and the unified L2 cache.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache size</td>
<td>16 KB, direct mapped</td>
</tr>
<tr>
<td>L1 block size</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>2 MB, direct mapped</td>
</tr>
<tr>
<td>L2 block size</td>
<td>16 Bytes</td>
</tr>
</tbody>
</table>

Table 6-3 Configurations of L1 and L2 caches used in experiments

All instructions and memory access are assumed to take one clock cycle; hence the details of bus contention and arbitration are not simulated. Previous work [Woo95] has taken a similar approach for multiprocessor cache simulation. This approach is valid because the non-determinism inherent in multiprocessor simulation already affects the statistics. Idealized multiprocessor cache simulation provides a useful characterization of multiprocessor execution and cache coherence interactions [Man01].

Many parameters were investigated in order to study the impact of augmenting synchronization with the MESI cache coherence protocol in the proposed protocol LCCP. The following performance parameters were monitored:

a) Total execution time of the benchmark.

b) Instruction execution rate in (Ins/sec).
c) Number of acknowledged invalidations defined as invalidations that were responded to by other processors.

6.3 Execution time

The execution time of the benchmarks on the simulator was measured in seconds (sec) on 2 configurations:

a) MP_Simplesim with a single shared L2 cache and implementing MESI as the cache coherence protocol.

b) MP_Simplesim with shared L2 cache and implementing LCCP as the cache coherence protocol.

Table 6-4 shows the measured execution times of the ocean contiguous-partitions benchmark when it was run on each of the two simulator versions.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Execution Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MESI</td>
</tr>
<tr>
<td>4</td>
<td>73</td>
</tr>
<tr>
<td>8</td>
<td>77</td>
</tr>
<tr>
<td>16</td>
<td>109</td>
</tr>
<tr>
<td>32</td>
<td>118</td>
</tr>
</tbody>
</table>

*Table 6-4* Execution times of ocean contiguous-partitions benchmark
Figure 6-1 Execution time of Ocean contiguous-partitions benchmark

Table 6-5 Execution times of ocean non-contiguous-partitions benchmark

Figure 6-2 Execution time of Ocean non-contiguous-partitions benchmark
Table 6-6 Execution times of Barnes-hut benchmark

<table>
<thead>
<tr>
<th>Processors</th>
<th>Execution Time (Sec)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MESI</td>
<td>LCCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>149</td>
<td>135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>166</td>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>196</td>
<td>155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>221</td>
<td>197</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-3 Execution time of Barnes-Hut benchmark

Figure 6-4 Percentage improvement in execution time when using LCCP
Figure 6-1, Figure 6-2 and Figure 6-3 show the execution time of running Ocean contiguous-partitions version, Ocean non-contiguous-partitions version and Barnes-hut benchmarks respectively using the two versions of the simulator as discussed before. The execution time represents the total simulation time of all the processor cores in a given simulation run and not the time taken by each individual processor to finish its simulation. The individual time of each processor core is not available from the simulator.

As shown in Figure 6-4, it was observed that the execution time of the ocean contiguous-partitions benchmark on processor cores running LCCP as a cache coherence protocol including support for synchronization reached a maximum improvement of 29% when running on 16 processors. In case of ocean non-contiguous-partitions, it reached a maximum improvement of 26% when running on 16 processors. In case of Barnes-hut, it reached a maximum improvement of 21% when running on 16 processors.

It was observed also that there was slight difference in the percentage of performance improvement between 4 and 8 processors and that this percentage decreased when going from 16 to 32 processors. This implied that in case of 4 or 8 processors, the performance of synchronization using LCCP was close to that when using the operating system. In case of 16 processors, since the number of processors increased the load on the operating system to synchronize the lock variables increased and the performance improvement was maximum in favor of LCCP.

In case of 32 processors, both execution times of LCCP and MESI increased due to the increase in the instances of the program which led also to the increase in the
copies of the lock variables that need to be synchronized and this appeared in the
decrease in the percentage of execution time improvement of LCCP compared to
MESI.

It was shown in Figure 6-1, Figure 6-2 and Figure 6-3 that the execution time
increases as the number of processors increases. That contradicts what should be
found, which is a decrease in the simulation time as the number of processors
increases. This is due to the fact that execution times represent the total simulation
time of all the processor cores in a given simulation run and not the time taken by
each individual processor to finish its simulation. However the execution time results
were plotted in this work since we are interested in the relative speedup achieved
when using LCCP compared to MESI.

6.4 Number of instructions executed per second

![Execution Rate Ocean contiguous-partitions](image)

**Figure 6-5** Execution rate of Ocean contiguous-partitions benchmark
Figure 6-6 Execution rate Ocean non-contiguous-partitions benchmark

Figure 6-7 Execution rate Barnes-Hut benchmark

Figure 6-5, Figure 6-6, and Figure 6-7 show the execution rate of the benchmarks in instructions per second. It was observed that the execution rate of the benchmarks when running LCCP cache coherence protocol was much more than that of MESI in the 16 processors configuration due to synchronization done by the operating system.
not the hardware in the cache coherence protocol. This was also reflected in the execution time as explained before.

6.5 Number of acknowledged invalidations

The number of acknowledged invalidations is an important parameter as it reflects the effect of avoiding the *livelock* condition when one processor wants to load a cache line in the lock state and another processor has another copy of the requested cache line. The requesting processor will invalidate the owner processor thus failing the conditional store. When the previous owner processor tries to store the line it will cache miss and will load another copy again. This copy is now with another processor which will be invalidated again and so on. When the cache line is locked the requesting processor knows that the owner processor is in the critical section so it will not invalidate its copy but will wait for the line in the waiting queue (spinning) until the owner processor comes out of the critical section and issues invalidation on the bus. Only at that time the requesting processor will get the cache line.

Figure 6-8, Figure 6-9 and Figure 6-10 show the distribution of acknowledged invalidations in case of Ocean contiguous-partitions, Ocean non-contiguous-partitions and Barnes-Hut benchmarks respectively.
Figure 6-8 Number of acknowledged invalidations - Ocean contiguous-partitions

Figure 6-9 Number of acknowledged invalidations - Ocean non-contiguous-partitions
Figure 6-10 Number of acknowledged invalidations-Barnes-Hut

Figure 6-11 Percentage improvement in acknowledged invalidations
As observed from Figure 6-8, Figure 6-9, Figure 6-10 and reflected in Figure 6-11, the results reflect significant reduction in the number of acknowledged invalidations when using LCCP compared with MESI. This reduction ranged from 28% up to 77% in case of ocean contiguous-partitions benchmark, from 32% up to 77% in case of ocean contiguous-partitions benchmark and from 57% to 77% in case of Barnes-hut benchmark.

The decrease in the number of acknowledged invalidations means less traffic in the transfer of the cache line from L2 cache or main memory to the individual L1 caches. This decrease in the traffic leads to an increase in the performance of the CMP.

6.6 Effect of L2 cache size

The effect obtained by varying the size of L2 cache was studied. The experiments were performed using Ocean non-contiguous-partitions benchmark. The L2 cache size was varied from 256 Kbytes to 2 Mbytes and was tested on 4, 8, 16 and 32 processor configurations. The simulator was running the Lock Cache Coherence Protocol (LCCP)

Figure 6-12 shows the execution time of Ocean non-contiguous-partitions benchmark while varying the size of L2 cache.
It is observed that by increasing the size of L2 cache, the execution time is decreased. That appeared more as the number of processors increased. This implied the conclusion that if the total size of L1 cache is near the size of L2 cache, the execution time increases and if there is a difference in size execution time decreases. This was observed in the case of 32 processors.

6.7 Interconnection network

A new interconnection network was introduced to the CMP as shown in Figure 6-13. The processor cores are arranged in a matrix $n 	imes n$ such that the processor cores belonging to the same column are connected together with a bus. No horizontal connections are found between the processor cores. The vertical buses are connected to a multi-port shared L2 cache. The $n 	imes n$ design was chosen for uniform placement of processor tiles on chip surface which is square in shape.
The effect of this design was tested on the benchmarks and the results were compared with the results from the bus configuration. All the results were obtained for the Lock Cache Coherence Protocol (LCCP).

6.7.1 Execution Time

Figure 6-14, Figure 6-15 and Figure 6-16 show the execution time of ocean contiguous-partitions, ocean non-contiguous-partitions and Barnes-hut benchmarks.
respectively when run on the CMP with the interconnection network and with the internal bus.

**Figure 6-14** Execution time of ocean contiguous-partitions using interconnection network and bus

**Figure 6-15** Execution time of ocean non-contiguous-partitions using interconnection network and bus with LCCP
Figure 6-16 Execution time of Barnes-hut using interconnection network and bus with LCCP

It was observed from the figures that when using the interconnection network the execution time is improved from 24% up to 64% at 16 processors configuration when running ocean contiguous-partitions benchmark, from 20% up to 62% at 16 processors configuration when running ocean non-contiguous-partitions benchmark and from 32% up to 66% at 16 processors configuration when running Barnes-hut benchmark. This significant improvement is due to the nature of the interconnection network and the multi-port L2 cache.

When a processor misses a cache line in its L1 cache, it snoops the bus connecting the processors on the same column and checks the L2 cache for the line. If one of the processors has the missed line, it provides it and if the L2 cache has the missed line, it will provide it. If the L2 cache doesn’t have it then the rest of the processors in the network won’t have it since a copy is kept in the L2 cache. This improves the snooping time internally on the bus and the L2 cache gets the line from the memory and provides a copy to the requesting processor.
6.7.2 Execution Rate

Figure 6-17, Figure 6-18 and Figure 6-19 show the execution rates of the benchmarks when the interconnection network and the bus are used.

**Figure 6-17** Execution rate of ocean contiguous-partitions benchmark using the interconnection and bus with LCCP

**Figure 6-18** Execution rate of ocean non-contiguous-partitions benchmark using the interconnection and bus with LCCP
From the figures the execution rate increased in case of the interconnection network especially at the 16 processors configuration and that appeared in the decrease in the execution time as shown before.

6.7.3 Acknowledged Invalidations

**Figure 6-19** Execution rate of Barnes-hut benchmark using the interconnection and bus with LCCP

**Figure 6-20** Ack. invalidations of ocean contiguous-partitions using the interconnection and bus with LCCP
Figure 6-21 Acknowledged invalidations of ocean non-contiguous partitions using the interconnection and bus with LCCP.

Figure 6-22 Acknowledged invalidations of Barnes-hut using the interconnection and bus with LCCP.
As shown from Figure 6-20, Figure 6-21 and Figure 6-22 the number of acknowledged invalidations showed a significant improvement when using the interconnection network compared to the bus configuration.

In case of the ocean contiguous-partitions, the reduction reached 87% on 4 processors, in case of ocean non-contiguous-partitions the reduction reached 88% on 4 processors and in case of Barnes-hut the reduction reached 78% on 4 processors.
7 CONCLUSIONS & RECOMMENDATIONS

In this thesis, integrating synchronization within the cache coherence protocol was studied for chip multi-processors (CMP). The performance of the CMP was studied in terms of the execution time and the execution rate of the benchmarks together with the number of acknowledged invalidations.

A novel interconnection network that can be used with multi-ported L2 cache was introduced and its performance was studied and compared to the traditional internal bus design.

7.1 Conclusions

Shared on-chip L2 cache configuration is very common in the design of chip multiprocessors. However the task of synchronization between processor cores was left to the operating system. This results in a large overhead to the synchronization needed between cores and leads to increase in delay.

This research contributed to the problem of synchronization between processor cores on CMP by introducing a new state to the MESI cache coherence protocol and a wait-table on chip using the shared on-chip L2 cache model.

The new cache coherence protocol, lock based cache coherence protocol (LCCP), was implemented. Mp_Simplesim simulator was modified to support the LCCP protocol and shared L2 cache mode. Simulation experiments using Ocean and Barnes multiprocessor benchmarks were conducted. The results show that LCCP outperformed MESI for shared on-chip L2 cache in terms of execution time due to the elimination of the spin-lock overhead. LCCP also resulted in significant improvement
in the number of acknowledged invalidations compared to MESI protocol since the wait-table saved unnecessary invalidations that could happen during spin-waiting when the processor misses the lock variable during a write instruction.

The effect of the size of the L2 cache on execution time was also studied. It was shown that as the size of the L2 cache is more that the sum of the sizes of the individual L1 caches, execution time is improved.

The novel interconnection network performed also better than the on-chip bus. This is due to the parallelism inherited in the use of the multi-ported L2 cache and the better organization of the processor cores on the chip.

7.2 Recommendations and Future Work

For future work, the performance of the designed CMP could be studied when using the mesh as an interconnection network, or using an internal cross-bar switch.

In the future work, the current design could be laid on a chip and studied in terms of the number of transistors it uses and the power it consumes, since these two factors are of high importance to CMPs. The manufactured CMP would be tested again and the real performance results would be compared to those obtained by simulation.

As for future study, the design could be scaled to accommodate a number of the designed CMP with the novel network, connected together with a bus. Cache coherence between those CMP’s could be studied. This scalable design needs to be further investigated.
REFERENCES


[SPLASH] [http://www-flash.stanford.edu/apps/SPLASH](http://www-flash.stanford.edu/apps/SPLASH)


APPENDIX A

/*
 * mpcache.c - multiprocessor cache simulation code with graphic visualization
 *
 * This file is used in conjunction with the SimpleScalar tool suite
 * originally written by Todd M. Austin for the Multiscalar Research Project
 * at the University of Wisconsin-Madison.
 *
 * The file was created by Naraig Manjikian at Queen's University,
 * Kingston, Ontario, Canada.
 *
 * Copyright (C) 2000 by Naraig Manjikian
 *
 * This source file is distributed "as is" in the hope that it will be
 * useful. The tool set comes with no warranty, and no author or
 * distributor accepts any responsibility for the consequences of its
 * use.
 *
 * Everyone is granted permission to copy, modify and redistribute
 * this tool set under the following conditions:
 *
 * This source code is distributed for non-commercial use only.
 * Please contact the maintainer for restrictions applying to
 * commercial use.
 *
 * Permission is granted to anyone to make or distribute copies
 * of this source code, either as received or modified, in any
 * medium, provided that all copyright notices, permission and
 * nonwarranty notices are preserved, and that the distributor
 * grants the recipient permission for further redistribution as
 * permitted by this document.
 *
 * Permission is granted to distribute this file in compiled
 * or executable form under the same conditions that apply for
 * source code, provided that either:
 *
 * A. it is accompanied by the corresponding machine-readable
 *    source code,
 * B. it is accompanied by a written offer, with no time limit,
 *    to give anyone a machine-readable copy of the corresponding
 *    source code in return for reimbursement of the cost of
 *    distribution. This written offer must permit verbatim
 *    duplication by anyone, or
 * C. it is distributed by someone who received only the
 *    executable form, and is accompanied by a copy of the
 *    written offer of source code that they received concurrently.
 *
 * In other words, you are welcome to use, share and improve this
 * source file. You are forbidden to forbid anyone else to use,
 * share
 * and improve what you give them.
 */
#include <stdlib.h>
#include <math.h>

#include "ss.h"
#include "sim.h"
#include "misc.h"

#include "syscall.h"
#include "memory.h"

#ifdef GRAPHICS
#include "graphics.h"
#endif /* GRAPHICS */

/* cache line states */
enum {UNTOUCHED = 0, INVALID, SHARED, EXCLUSIVE, MODIFIED, MODIFIED_ABOVE, LOCKED};

/* bus request types */
enum {READ, READ_EX, READ_LK, UPGRADE};

/* Lock Flag and Lock Address registers */
SS_WORD_TYPE regs_LFR[MAX_PROCS];    /* Ihab */
SS_ADDR_TYPE regs_LAR[MAX_PROCS];    /* Ihab */

static int L1_cache_line_size, L2_cache_line_size;
static int L1_cache_size, L2_cache_size;
static int L1_num_lines, L2_num_lines;
static int L1_line_shift, L2_line_shift;
static int L1_line_mask, L2_line_mask; /* mask after line shift */
static int L1_tag_shift, L2_tag_shift;

static int L2_to_L1_line_size_ratio;    /* should be a power of 2 */

/* arrays for storing tags and states for L1 and L2 caches of each processor */
static unsigned int *L1_tags[MAX_PROCS];
static unsigned int *L1_states[MAX_PROCS];
static unsigned int *L2_tags;    /*Ihab*/
static unsigned int *L2_states;    /*Ihab*/

typedef struct _mpcachestats
{
    unsigned int num_instructions,
    L1_accesses,
    L1_stores,
    L1_locked_loads,
    L1_conditional_stores,
    L1_hits,
    L1_misses;
    double L1_miss_ratio;
    unsigned int L1_upgrades;
    double L1_upgrade_miss_ratio;
    unsigned int L2_writebacks,
    L2_accesses,
    L2_hits,
    L2_misses;
    double L2_miss_ratio;
}
typedef struct _WaitQElement
{
    int pid;
    unsigned int L1_line;
} WaitQElement;

static WaitQElement  *WaitQ;

int num_wait_procs = 0;              /*number of waiting processors*/

static  char *stat_names[] =
{
    "Instructions",
    "L1_accesses",
    "L1_stores",
    "L1_locked_loads",
    "L1_conditional_stores",
    "L1_hits",
    "L1_misses",
    "L1_miss_ratio",
    "L1_upgrades",
    "L1_upgrade_miss_ratio",
    "L2_writebacks",
    "L2_accesses",
    "L2_hits",
    "L2_misses",
    "L2_miss_ratio",
    "L2_upgrades",
    "L2_upgrade_miss_ratio",
    "mem_writebacks",
    "excl_to_mod_changes",
    "read_requests",
    "excl_data_returns",
    "read_excl_requests",
    "upgrade_requests",
    "external_bus_requests",
    "snoop_hits",
    "snoop_hit_ratio",
    "excl_to_shrd_changes",
    "exclusive_to_modified_changes",
    "read_requests",
    "exclusive_data_returns",
    "read_excl_requests",
    "upgrade_requests",
    "external_bus_requests",
    "snoop_hits",
    "snoop_hit_ratio",
    "exclusive_to_shared_changes",
    "shared_data_responses",
    "exclusive_data_responses",
    "external_invalidations",
    "snoop_hits",
    "snoop_hit_ratio",
    "exclusive_to_shrd_changes",
};
"shrd_data_responses",
"excl_data_responses",
"external_invalidations"
};

static int double_flags[] =
{
0, /* Instructions */
0, /* L1_accesses */
0, /* L1_stores */
0, /* L1_locked_loads */
0, /* L1_conditional_stores */
0, /* L1_hits */
0, /* L1_misses */
1, /* L1_miss_ratio */
0, /* L1_upgrades */
1, /* L1_upgrade_miss_ratio */
0, /* L2_writebacks */
0, /* L2_accesses */
0, /* L2_hits */
0, /* L2_misses */
1, /* L2_miss_ratio */
0, /* L2_upgrades */
1, /* L2_upgrade_miss_ratio */
0, /* mem_writebacks */
0, /* excl_to_mod_changes */
0, /* read_requests */
0, /* excl_data_returns */
0, /* read_excl_requests */
0, /* upgrade_requests */
0, /* external_bus_requests */
0, /* snoop_hits */
1, /* snoop_hit_ratio */
0, /* excl_to_shrd_changes */
0, /* shrd_data_responses */
0, /* excl_data_responses */
0, /* external_invalidation */
};

/* compute number of statistics in a clever manner */
/* using ratio of array size and array element size */
/* (if new statistics are added, all arrays should be updated */
/* properly!) */
#define NUM_STATISTICS (sizeof (stat_names) / sizeof (char *))

/* array of per-processor statistic structures */
static MPCacheStats mpcachestats[MAX_PROCS];

/* global statistics */
static double invalidation_set_size_sum;
static unsigned int invalidation_set_size_samples;
static unsigned int

invalidation_set_size_histogram[MAX_PROCS];

/* total processes created --- defined externally */
extern volatile int num_created_processes;

/* variables to keep track of most recent L1 & L2 references/misses */
per processor to display bar graphs of miss ratios */
static int ref_count[MAX_PROCS];
static int L1_miss_count[MAX_PROCS];
static int L2_miss_count; /*Ihab*/
static int L1_last_miss_ratio[MAX_PROCS];
static int L2_last_miss_ratio; /*Ihab*/

/* macros for setting cache line states; these can be overridden
to call other functions, such as for graphical display */
#define SET_L1_STATE(pid,line,state) (L1_states[pid][line]=state)
#define SET_L2_STATE(line,state) (L2_states[line]=state) /*Ihab*/

/*------------------------------------------------------------------*/
#endif /* DEBUG */

void MPCacheInit (int L1_line_size, int L1_size,
      int L2_line_size, int L2_size,
      int gr_flag, int gr_speed)
{
  int pid;

  /* save line and cache sizes from function arguments */
  L1_cache_line_size = L1_line_size;
  L1_cache_size = L1_size;
  L2_cache_line_size = L2_line_size;
  L2_cache_size = L2_size;

  L2_to_L1_line_size_ratio = L2_cache_line_size / L1_cache_line_size;

  /* compute necessary information for maintaining caches */
  L1_num_lines  = L1_cache_size / L1_cache_line_size;
  L1_line_shift = log_base2 (L1_cache_line_size);
  L1_line_mask  = L1_num_lines - 1;
  L1_tag_shift  = L1_line_shift + log_base2 (L1_num_lines);

  L2_num_lines  = L2_cache_size / L2_cache_line_size;
  L2_line_shift = log_base2 (L2_cache_line_size);
  L2_line_mask  = L2_num_lines - 1;
  L2_tag_shift  = L2_line_shift + log_base2 (L2_num_lines);

  #ifdef DEBUG
      /* caches will be allocated for active processors only on first
      ref */
      debug ("L1_cache_line_size = %d", L1_cache_line_size);
      debug ("L2_cache_line_size = %d", L2_cache_line_size);
      debug ("L1_size = %d", L1_size);
      debug ("L2_size = %d", L2_size);
      debug ("L1_line_shift = %d", L1_line_shift);
      debug ("L2_line_shift = %d", L2_line_shift);
      debug ("L1_line_mask = %d", L1_line_mask);
      debug ("L2_line_mask = %d", L2_line_mask);
      debug ("L1_tag_shift = %d", L1_tag_shift);
      debug ("L2_tag_shift = %d", L2_tag_shift);
      debug ("L2_to_L1_line_size_ratio = %d",
            L2_to_L1_line_size_ratio);
  #endif /* DEBUG */
}
static void CreateL1Cache (int pid)
{
    /* create L1 and L2 tag and state arrays, initialized to zero
     * (for 'untouched' state) */
    L1_tags[pid] = calloc (L1_num_lines, sizeof (unsigned int));
    L1_states[pid] = calloc (L1_num_lines, sizeof (unsigned int));
}

static void CreateL2Cache (void)
{
    L2_tags = calloc (L2_num_lines, sizeof (unsigned int));
    L2_states = calloc (L2_num_lines, sizeof (unsigned int));
}

static void UpdateL1States (int current_pid,
    unsigned int L1_tag,
    unsigned int L1_line,
    int new_state)
{
    /* if tag matches and valid, update state */
    if (L1_tags[current_pid][L1_line] == L1_tag
        && L1_states[current_pid][L1_line] != INVALID)
    {
        SET_L1_STATE (current_pid, L1_line, new_state);
    }
}

static void UpdateL1StatesExceptOne (int current_pid,
    unsigned int L2_tag,
    unsigned int L2_line,
    int new_state,
    unsigned int except_L1_line)
{
    int i;
    unsigned long base_addr = (L2_tag << L2_tag_shift)
        | (L2_line << L2_line_shift);
    unsigned int base_L1_line, L1_tag;
    L1_tag = (base_addr >> L1_tag_shift);
    base_L1_line = (base_addr >> L1_line_shift) & L1_line_mask;
    for (i = 0; i < L2_to_L1_line_size_ratio; i++)
    {
        /* skip the line we do not want to change */
        if (base_L1_line + i == except_L1_line)
            continue;
        /* if tag matches and valid, update state */
        if (L1_tags[current_pid][base_L1_line + i] == L1_tag
            && L1_states[current_pid][base_L1_line + i] != INVALID)
        {
            SET_L1_STATE (current_pid, base_L1_line + i, new_state);
        }
    }
static int CheckL1States (int current_pid,
    unsigned int L2_tag,
    unsigned int L2_line,
    int state)
{
    int i, count;
    unsigned long base_addr = (L2_tag << L2_tag_shift)
        | (L2_line << L2_line_shift);
    unsigned int base_L1_line, L1_tag;
    L1_tag = (base_addr >> L1_tag_shift);
    base_L1_line = (base_addr >> L1_line_shift) & L1_line_mask;
    count = 0;
    for (i = 0; i < L2_to_L1_line_size_ratio; i++)
    {
        /* if tag and state match, increase count */
        if (L1_tags[current_pid][base_L1_line + i] == L1_tag
            && L1_states[current_pid][base_L1_line + i] == state)
        {
            ++count;
        }
    }
    return count; /* return count of matching lines */
}

static int CheckAndUpdateL1Copies (int requesting_pid,
    unsigned int L1_tag,
    unsigned int L1_line,
    int bus_req_type)
{
    int i, count, lock_count;
    count = 0; /* count of snoop hits _for this request only_ */
    for (i = 0; i < num_created_processes; i++)
    {
        if (i == requesting_pid)
            continue; /* skip processor making the request */
        /* increment count of external bus requests seen by this
           processor */
        mpcachestats[i].external_bus_requests++;
        /* special check if processor was just created and has not yet
           executed its first memory instruction that will force
           creation of cache data structures; if necessary, create caches here */
        if (L1_tags[i] == NULL) CreateL1Cache (i);
/* >>> FILTER: do nothing further if there is no valid tag match */
if (L1_tags[i][L1_line] != L1_tag || L1_states[i][L1_line] == INVALID)
    continue;
/* valid tag match, so increment count of external snoop hits */
mpcachestats[i].snoop_hits++; /* for current processor */
count++; /* for this request */
if (bus_req_type == READ)
{
    switch (L1_states[i][L1_line])
    {
        case EXCLUSIVE:
            /* increment count of exclusive to shared transitions */
            { 
                mpcachestats[i].exclusive_to_shared_changes++;
                UpdateL1States (i, L1_tag, L1_line, SHARED);
                break;
            }
        case MODIFIED:
            /* owner of modified data provides shared data response */
            { 
                mpcachestats[i].shared_data_responses++;
                UpdateL1States (i, L1_tag, L1_line, SHARED);
                break;
            }
    }
    else /* read_ex request */
    {
        switch (L1_states[i][L1_line])
        {
            case EXCLUSIVE:
            case SHARED:
                /* for read_ex, any read-only copies are just invalidated, so increment count */
                mpcachestats[i].external_invalidations++;
                break;
            case MODIFIED:
                return(0);
            }
        }
    }
}
/* owner of modified data provides excl data resp */
mpcachestats[i].exclusive_data_responses++;
break;
}

/* regardless of whatever _valid_ state (invalid state already filtered out above), current processor loses its copy */
/* change _all_ L1 line states */
UpdateL1States (i, L1_tag, L1_line, INVALID);
}

/* for read_ex requests, update invalidation statistics after all processors are dealt with */
if ((bus_req_type == READ_EX) || (bus_req_type == READ_LK))
{
 /* increment sum and count of invalidation set size */
 invalidation_set_size_sum += count;
 ++invalidation_set_size_samples;
 ++invalidation_set_size_histogram[count];
}
return count; /* return snoop hit count in other processors */

/*------------------------------------------------------------------*/
void InvalidateOtherCaches (int requesting_pid,
 unsigned int L1_tag,
 unsigned int L1_line) /*was void*/
{
 int i, invalidation_set_size;
 invalidation_set_size = 0;
 for (i = 0; i < num_created_processes; i++)
 {
  if (i == requesting_pid)
   continue; /* skip the processor that is requesting invalidation*/

  /* increment count of external bus requests for this processor */
  mpcachestats[i].external_bus_requests++;

  /* special check if processor was just created and has not yet executed its first memory instruction that will force creation of cache data structures; if necessary, create caches here */
  if (L1_tags[i] == NULL) CreateL1Cache (i);

  /* check if data is in L1 cache */
  if (L1_tags[i][L1_line] == L1_tag && L1_states[i][L1_line] != INVALID && L1_states[i][L1_line] != LOCKED)
  {
   /* increment count of external snoop hits for this processor */

   /* increment count of external bus requests for this processor */
mpcachestats[i].snoop_hits++;
/* increment count of external invalidations for this processor */
mpcachestats[i].external_invalidations++;
/* increment local count of invalidation hits */
++invalidation_set_size;
/* change _all_ L1 line states */
UpdateL1States (i, /* which L1 cache to update */
L1_tag, L1_line , /* use to build base L1 addr */
    INVALID);
}
}
/* increment sum and count of invalidation set size */
invalidation_set_size_sum += invalidation_set_size;
++invalidation_set_size_samples;
++invalidation_set_size_histogram[invalidation_set_size];
}
/*------------------------------------------------------------------*/

void MPCacheMemRef (int pid, unsigned long data_addr, int is_a_store)
{
    unsigned int L1_tag = (data_addr >> L1_tag_shift);
    unsigned int L2_tag = (data_addr >> L2_tag_shift);
    unsigned int L1_line = (data_addr >> L1_line_shift) & L1_line_mask;
    unsigned int L2_line = (data_addr >> L2_line_shift) & L2_line_mask;
    unsigned int L1_lock_tag[MAX_PROCS];
    int i;
    static int window_event_interval_count;

    /* caches for each pid are allocated on first reference */
    if (L1_tags[pid] == NULL)
    {
        CreateL1Cache (pid);
    }
    if (L2_tags == NULL) CreateL2Cache(); /*Ihab*/

    mpcachestats[pid].L1_accesses++;  
    if (is_a_store)
        mpcachestats[pid].L1_stores++;
        /* probe L1 cache */
    if (L1_tags[pid][L1_line] == L1_tag && L1_states[pid][L1_line] != INVALID)
    {
        /* tag match with _valid_ state, so now handle specific states */
        switch (L1_states[pid][L1_line])
        {
        case MODIFIED:
/ * hit in L1 for both loads and stores */
mpcachestats[pid].L1_hits++;
break;

case LOCKED:
{
mpcachestats[pid].L1_hits++;
if (is_a_store)
{
mpcachestats[pid].L1_conditional_stores++;
void ss_syscall(mem_access_fn mem_fn, SS_INST_TYPE sc,
int pid);
regs_LFR[pid] = 0;
regs_LAR[pid] = 0;
L1_lock_tag[pid] = 0;
SET_L1_STATE(pid, L1_line, MODIFIED);
if(num_wait_procs > 0)
{
if (DeQueue(L1_line) > -1)
{
SET_L2_STATE(L2_line, LOCKED);
}
else SET_L2_STATE(L2_line, MODIFIED_ABOVE);
}
else
SET_L2_STATE(L2_line, MODIFIED_ABOVE);
else
mpcachestats[pid].L1_hits++;
break;
}
}

case EXCLUSIVE:
/* hit in L1 for loads and stores */
mpcachestats[pid].L1_hits++;
/* but for stores, change L1 and L2 states to modified */
if (is_a_store)
{
mpcachestats[pid].exclusive_to_modified_changes++;
SET_L1_STATE (pid, L1_line, MODIFIED);
SET_L2_STATE (L2_line, MODIFIED_ABOVE);
}
break;

case SHARED:
if (! is_a_store)
mpcachestats[pid].L1_hits++; /* simple hit for reads */
else
{
/* writes requires an upgrade request */
mpcachestats[pid].L1_upgrades++;
InvalidateOtherCaches (pid, /* who is invalidating */
L1_tag, L1_line);
/* change L1 and L2 states to modified */
SET_L1_STATE (pid, L1_line, MODIFIED);
SET_L2_STATE (L2_line, MODIFIED_ABOVE);
}
break;
else /* no tag match, _or_ invalid state with tag match */
{
/* no tag match in L1, so pure miss */
mpcachestats[pid].L1_misses++;
/* determine if writeback of current _L1_ line to L2 is needed */
/* (note that this would not be needed if tag matched with invalid */
if (L1_states[pid][L1_line] == MODIFIED)
{
/* reconstruct address of line being replaced */
unsigned long L2_wback_addr =
   (L1_tags[pid][L1_line] << L1_tag_shift)
   | (L1_line << L1_line_shift);
/* determine which line in L2 it belongs in */
unsigned int L2_wback_line =
   (L2_wback_addr >> L2_line_shift) & L2_line_mask;
unsigned int L2_wback_tag =
   (L2_wback_addr >> L2_tag_shift);
/* mark current L1 line (which will be written back to L2) as INVALID to allow 'CheckL1States()' to work properly (note that graphics are not informed of this change) */
L1_states[pid][L1_line] = INVALID;
/* check if other modified subblocks of wback L2 line are in L1 */
/* if none, then change state of wback L2 line to just modified */
/* (otherwise, remains modified_above) */
if (CheckL1States (pid, L2_wback_tag, L2_wback_line,
MODIFIED)== 0)
   SET_L2_STATE (L2_wback_line, MODIFIED);
/* update count of writebacks to L2 */
mpcachestats[pid].L2_writebacks++;
}
/* now check if we have the missing data in L2 */
if ((L2_tags[L2_line] == L2_tag)
   && (L2_states[L2_line] != INVALID))
{
if (! is_a_store)
{
mpcachestats[pid].L2_hits++; /* valid tag match for read */
/* this is the easy case; just use same state as L2, except if state is modified or modified_above */
SET_L1_STATE (pid, L1_line,
   (L2_states[L2_line] == MODIFIED
      || L2_states[L2_line] == MODIFIED_ABOVE) ?
      EXCLUSIVE : L2_states[L2_line]);
L1_tags[pid][L1_line] = L1_tag;
}
else /* is_a_store */
{
/* set tag in L1 */
L1_tags[pid][L1_line] = L1_tag;
/* check if an upgrade request is needed */
if (L2_states[L2_line] == SHARED)
{
    mpcachestats[pid].L2_upgrades++;
    InvalidateOtherCaches (pid, /* who is invalidating */
        L1_tag, L1_line);
    /* make all other L1 subblocks exclusive
       except for the one that will change to modified */
    UpdateL1StatesExceptOne (pid, L2_tag, L2_line,
        EXCLUSIVE, L1_line);
    /* change L1 state to modified */
    SET_L1_STATE (pid, L1_line, MODIFIED);
    /* change L2 state */
    SET_L2_STATE (L2_line, MODIFIED_ABOVE);
}
else /* either exclusive or already modified in L2 */
{
    mpcachestats[pid].L2_hits++; /* hit */
    /* change L1 and L2 states */
    SET_L1_STATE (pid, L1_line, MODIFIED);
    SET_L2_STATE (L2_line, MODIFIED_ABOVE);
}
else /* pure miss in L2 */
{
    mpcachestats[pid].L2_misses++;
    /* check whether location of missing line in L2 needs mem
       wback */
    if (   L2_states[L2_line] == MODIFIED
        || L2_states[L2_line] == MODIFIED_ABOVE)
    {
        /* mem writeback needed */
        mpcachestats[pid].mem_writebacks++;
        /* we must invalidate _all_ subblocks of this L2 line in
           L1 */
        UpdateL1States (pid, L1_tag, L1_line, INVALID);
    }
    /* set tags in L1 and L2 */
    L1_tags[pid][L1_line] = L1_tag;
    L2_tags[L2_line] = L2_tag;
    /* base on type of mem ref, update statistics for bus
       requests, update other caches, and finally update states in L1 and
       L2 */
    if (is_a_store)
    {
        mpcachestats[pid].read_excl_requests++;
        (void) CheckAndUpdateL1Copies (pid, L1_tag, L1_line,
            READ_EX);
SET_L1_STATE (pid, L1_line, MODIFIED);
SET_L2_STATE (L2_line, MODIFIED_ABOVE);
}
else /* is a load */
{
    mpcachestats[pid].read_requests++;

    /* with snooping, the data response would indicate
    whether this processor could set its final line state */
    if (CheckAndUpdateL1Copies (pid, L1_tag, L1_line, READ) > 0)
    {
        /* at least one other copy in another cache */
        if (regs_LFR[pid] == 0)
        {
            SET_L1_STATE (pid, L1_line, SHARED);
            SET_L2_STATE (L2_line, SHARED);
        }

        if (regs_LFR[pid] == 1)
        {
            if (L1_lock_tag[pid] != L1_tag)
            {
                if (L1_states[pid][L1_line] != LOCKED)
                {
                    mpcachestats[pid].L1_locked_loads++;
                    regs_LAR[pid] = data_addr;
                    L1_lock_tag[pid] = L1_tag;
                    UpdateL1StatesExceptOne (pid, L2_tag, L2_line,
                    INVALID,L1_line);
                    SET_L1_STATE (pid, L1_line, LOCKED);
                    SET_L2_STATE (L2_line, LOCKED);
                    CheckAndUpdateL1Copies (pid, L1_tag, L1_line,
                    READ_LK);
                }
                else
                {
                    EnQueue(pid, L1_line);
                }
            }
            else
            {
                SET_L1_STATE (pid, L1_line, LOCKED);
                SET_L2_STATE (L2_line, LOCKED);
            }
        }
        else
        {
            /* no other copies in any other cache */
            if (regs_LFR[pid] == 0)
            {
                SET_L1_STATE (pid, L1_line, EXCLUSIVE);
                /* keep track of how many exclusive responses for
                reads */
                mpcachestats[pid].exclusive_data_returns++;
            }
            if (regs_LFR[pid] == 1)
{  
    if (L1_lock_tag[pid] != L1_tag)
    {  
        if (L1_states[pid][L1_line] != LOCKED)
        {  
            mpcachestats[pid].L1_locked_loads++;
            regs_LAR[pid] = data_addr;
            L1_lock_tag[pid] = L1_tag;
            UpdateL1StatesExceptOne (pid, L2_tag, L2_line, INVALID, L1_line);
            SET_L1_STATE (pid, L1_line, LOCKED);
            SET_L2_STATE (L2_line, LOCKED);
            CheckAndUpdateL1Copies (pid, L1_tag, L1_line, READ_LK);
        }
        else
        {  
            SET_L1_STATE (pid, L1_line, EXCLUSIVE);
        }
    }
    else
    {  
        SET_L1_STATE (pid, L1_line, LOCKED);
        SET_L2_STATE (L2_line, LOCKED);
    }
}

/*------------------------------------------------------------------
*/

static int InitWaitQueue ()
{
    if (num_wait_procs == 0)
    {  
        WaitQ = NULL;
        return(0);
    }
    else
    {  
        WaitQ = (WaitQElement *) realloc (WaitQ, num_wait_procs * sizeof(WaitQElement));
        printf("\n Wait Processors = %d \n", num_wait_procs);
        return(1);
    }
}

/*----------------------------------------------------------------*/
int EnQueue (int pid, unsigned int L1_line)
{
    num_wait_procs ++;
    InitWaitQueue();
}
int DeQueue (unsigned int L1_line)
{
    int i, j, pid;
    for (i=0; i<num_wait_procs; i++)
    {
        pid = WaitQ[i].pid;
        if (WaitQ[i].L1_line == L1_line)
        {
            SET_L1_STATE (pid, L1_line, LOCKED);
            regs_LFR[i] = 1;
            mpcachestats[pid].L1_locked_loads++;
            for(j = i; j < num_wait_procs-1; j++)
            {
                WaitQ[j].L1_line = WaitQ[j+1].L1_line;
                WaitQ[j].pid = WaitQ[j+1].pid;
            }
            num_wait_procs --;
            InitWaitQueue();
            return(pid);
        }
        else continue;
    }
    return (-1);
}

void MPCacheReport (void)
{
    int  i, j, **grid;
    double **fgrid;
    unsigned int total_read_requests = 0,
                total_locked_loads = 0,
                total_read_excl_requests = 0,
                total_mem_writebacks = 0,
                total_upgrade_requests = 0,
                total_shared_responses = 0,
                total_exclusive_responses = 0,
                total_bus_requests = 0;
    struct stat_stat_t *stat_ptr;
    unsigned int *num_instructions;

    /* retrieve array of instruction counts from statistics database */
    stat_ptr = stat_find_stat (sim_sdb, "sim_num_instructions");
    if (stat_ptr == NULL)
        fatal ("unable to retrieve array of instruction counts"
               "for final report");
    /* make pointer to array of counts */
    num_instructions = stat_ptr->variant.for_uintarray.var;
    for (i = 0; i < num_created_processes; i++)
    {
/* compute final statistics */
mp cachestats[i].num_instructions = num_instructions[i];
mp cachestats[i].upgrade_requests =
mp cachestats[i].L1_upgrades + mpc cachestats[i].L2_upgrades;
mp cachestats[i].L1_miss_ratio =
(double) mpc cachestats[i].L1_misses /
mp cachestats[i].L1_accesses
* 100.0;
mp cachestats[i].L1_upgrade_miss_ratio =
(double) mpc cachestats[i].L1_upgrades /
mp cachestats[i].L1_accesses
* 100.0;
mp cachestats[i].L2_accesses = mpc cachestats[i].L1_misses;
mp cachestats[i].L2_miss_ratio =
(double) mpc cachestats[i].L2_misses /
mp cachestats[i].L2_accesses
* 100.0;
mp cachestats[i].L2_upgrade_miss_ratio =
(double) mpc cachestats[i].L1_upgrades /
mp cachestats[i].L2_accesses
* 100.0;
mp cachestats[i].snoop_hit_ratio =
(double) mpc cachestats[i].snoop_hits /
mp cachestats[i].external_bus_requests
* 100.0;

total_read_requests += mpc cachestats[i].read_requests;
total_read_excl_requests += mpc cachestats[i].read_excl_requests;
total_mem_writebacks += mpc cachestats[i].mem_writebacks;
total_upgrade_requests += mpc cachestats[i].upgrade_requests;
total_shared_responses +=
mp cachestats[i].shared_data_responses;
total_exclusive_responses +=
mp cachestats[i].exclusive_data_responses;
}

/* create a integer "grid" and a floating-point "grid"
for the final output of statistics (note that both grids are
allocated to match the total number of statistics, but only a
portion of each one is used; inefficient, but simple) */
grid = (int **) malloc (NUM_STATISTICS * sizeof (int *));
for (j = 0; j < NUM_STATISTICS; j++)
grid[j] = (int *) malloc (num_created_processes * sizeof (int));
fgrid = (double **) malloc (NUM_STATISTICS * sizeof (double *));
for (j = 0; j < NUM_STATISTICS; j++)
fgrid[j] = (double *) malloc (num_created_processes * sizeof (double));

for (i = 0; i < num_created_processes; i++) /* loop through processors */
{
  int j = 0; /* counter to step through the "grids" */

grid[j++][i] = mpc cachestats[i].num_instructions;
grid[j++][i] = mpc cachestats[i].L1_accesses;
grid[j++][i] = mpc cachestats[i].L1_stores;
grid[j++][i] = mpc cachestats[i].L1_locked_loads;
grid[j++][i] = mpc cachestats[i].L1_conditional_stores;
grid[j++][i] = mpc cachestats[i].L1_hits;
grid[j++][i] = mpc cachestats[i].L1_misses;
fgrid[j++][i] = mpcachestats[i].L1_miss_ratio;
grid[j++][i] = mpcachestats[i].L1_upgrades;
fgrid[j++][i] = mpcachestats[i].L1_upgrade_miss_ratio;
grid[j++][i] = mpcachestats[i].L2_writebacks;
grid[j++][i] = mpcachestats[i].L2_accesses;
grid[j++][i] = mpcachestats[i].L2_hits;
grid[j++][i] = mpcachestats[i].L2_misses;
fgrid[j++][i] = mpcachestats[i].L2_miss_ratio;
grid[j++][i] = mpcachestats[i].L2_upgrades;
fgrid[j++][i] = mpcachestats[i].L2_upgrade_miss_ratio;
grid[j++][i] = mpcachestats[i].exclusive_to_modified_changes;
grid[j++][i] = mpcachestats[i].read_requests;
grid[j++][i] = mpcachestats[i].exclusive_data_returns;
grid[j++][i] = mpcachestats[i].read_excl_requests;
grid[j++][i] = mpcachestats[i].upgrade_requests;
grid[j++][i] = mpcachestats[i].external_bus_requests;
grid[j++][i] = mpcachestats[i].snoop_hits;
fgrid[j++][i] = mpcachestats[i].snoop_hit_ratio;
grid[j++][i] = mpcachestats[i].exclusive_to_shared_changes;
grid[j++][i] = mpcachestats[i].shared_data_responses;
grid[j++][i] = mpcachestats[i].exclusive_data_responses;
grid[j++][i] = mpcachestats[i].external_invalidations;
}

printf ("\n");
printf ("statistic\t");
for (i = 0; i < num_created_processes; i++)
  printf ("tPID %d", i);
printf ("\n\n");

for (j = 0; j < NUM_STATISTICS; j++)
{
  printf ("%-23s", stat_names[j]);
  for (i = 0; i < num_created_processes; i++)
    if (double_flags[j])
      printf ("\t%.2f\%", fgrid[j][i]);
    else
      printf ("\t%d", grid[j][i]);
  printf ("\n");
}

for (j = 0; j < NUM_STATISTICS; j++)
{
  free (grid[j]);
  free (fgrid[j]);
}
free (grid);
free (fgrid);

printf ("\n");
printf ("===== Invalidation set size statistics ===== \n");
printf ("\n");
printf ("#procs\tfrequency\n");
for (i = 0; i <= num_created_processes - 1; i++)
  printf ("%d\td%d\n", i, invalidation_set_size_histogram[i]);
printf ("\n");
if (invalidation_set_size_samples > 0)
  printf ("avg. invalidation set size = %.2f\n",
(double)invalidation_set_size_sum / invalidation_set_size_samples);
else
    printf ("avg. invalidation set size = 0\n");

/***********************IHAB******************************/
int total_invalidations = 0;
for(i = 0; i <= num_created_processes -1; i++)
{
    total_invalidations += invalidation_set_size_histogram[i];
}
printf("\n Total Invalidation = %d \n", total_invalidations);
/***********************IHAB******************************/

total_bus_requests =
    total_read_requests
+ total_read_excl_requests
+ total_upgrade_requests;

printf ("\n");
printf ("====== Total bus activity statistics =====\n");
printf ("\n");
printf ("total_requests\t\t%d\n", total_bus_requests);
printf ("\n");
printf ("read_requests\t\t%d\t(%5.2f%% of all requests)\n", total_read_requests, 
(double) total_read_requests / total_bus_requests * 100.0);
printf (" cache-to-cache xfers\t%d\t(%5.2f%% of total_read_requests)\n", 
    total_shared_responses, 
(double) total_shared_responses / total_read_requests * 100.0);
printf (" read_excl_requests\t%d\t(%5.2f%% of all requests)\n", total_read_excl_requests, 
(double) total_read_excl_requests / total_bus_requests * 100.0);
printf (" cache-to-cache xfers\t%d\t(%5.2f%% of total_read_excl_requests)\n", 
    total_exclusive_responses, 
(double) total_exclusive_responses / total_read_excl_requests * 100.0);
printf (" upgrade_requests\t%d\t(%5.2f%% of all requests)\n", total_upgrade_requests, 
(double) total_upgrade_requests / total_bus_requests * 100.0);

printf ("\n");
printf ("read & read_excl requests\t\t%d\t(%5.2f%% of all requests)\n", 
    total_read_requests + total_read_excl_requests, 
(double) (total_read_requests + total_read_excl_requests) / total_bus_requests * 100.0);
printf (" total_mem_writebacks\t\t%d\t(%5.2f%% of read/read_excl)\n", 
    total_mem_writebacks, (double) total_mem_writebacks / (total_read_requests + total_read_excl_requests) * 100.0);
printf (" sharing_writebacks\t\t%d\n", total_shared_responses);
printf (" writebacks per read/read_excl\t\t%d\n", 
    (double) (total_mem_writebacks + total_shared_responses) / (total_read_requests + total_read_excl_requests));
printf ("writebacks per bus request\t%.2f\n",
    (double) (total_mem_writebacks + total_shared_responses)
    / total_bus_requests);
}

/*
 * regs.c - architected registers state routines
 *
 * This file is an adaptation of the software in the SimpleScalar tool suite
 * originally written by Todd M. Austin for the Multiscalar Research Project
 * at the University of Wisconsin-Madison.
 *
 * The modifications were made by Naraig Manjikian at Queen's University,
 * Kingston, Ontario, Canada.
 *
 * The remainder of this header comment is unchanged from the original text.
 *
 *....................................................................
 ......
 * Copyright (C) 1994, 1995, 1996, 1997 by Todd M. Austin
 *
 * This source file is distributed "as is" in the hope that it will be
 * useful. The tool set comes with no warranty, and no author or
 * distributor accepts any responsibility for the consequences of its
 * use.
 *
 * Everyone is granted permission to copy, modify and redistribute
 * this tool set under the following conditions:
 *
 * This source code is distributed for non-commercial use only.
 * Please contact the maintainer for restrictions applying to
 * commercial use.
 *
 * Permission is granted to anyone to make or distribute copies
 * of this source code, either as received or modified, in any
 * medium, provided that all copyright notices, permission and
 * nonwarranty notices are preserved, and that the distributor
 * grants the recipient permission for further redistribution as
 * permitted by this document.
 *
 * Permission is granted to distribute this file in compiled
 * or executable form under the same conditions that apply for
 * source code, provided that either:
 *
 * A. it is accompanied by the corresponding machine-readable
 * source code,
 * B. it is accompanied by a written offer, with no time limit,
 * to give anyone a machine-readable copy of the corresponding
 * source code in return for reimbursement of the cost of
 * distribution. This written offer must permit verbatim
 * duplication by anyone, or
 * C. it is distributed by someone who received only the
 * executable form, and is accompanied by a copy of the
 * written offer of source code that they received concurrently.
 *
 * In other words, you are welcome to use, share and improve this
 * source file. You are forbidden to forbid anyone else to use, share
* and improve what you give them.
* INTERNET: dburger@cs.wisc.edu
* US Mail: 1210 W. Dayton Street, Madison, WI 53706
* $Id: regs.c,v 1.4 1997/03/11 01:19:28 taustin Exp taustin $
* $Log: regs.c,v $
* Revision 1.4 1997/03/11 01:19:28 taustin
* updated copyright
* long/int tweaks made for ALPHA target support
* Revision 1.3 1997/01/06 16:02:36 taustin
* comments updated
* Revision 1.1 1996/12/05 18:52:32 taustin
* Initial revision
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "misc.h"
#include "ss.h"
#include "loader.h"
#include "regs.h"

/* (signed) integer register file */
SS_WORD_TYPE regs_R[MAX_PROCS][SS_NUM_REGS];

/* floating point register file */
union regs_FP regs_F[MAX_PROCS];

/* (signed) hi register, holds mult/div results */
SS_WORD_TYPE regs_HI[MAX_PROCS];
/* (signed) lo register, holds mult/div results */
SS_WORD_TYPE regs_LO[MAX_PROCS];

/* floating point condition codes */
int regs_FCC[MAX_PROCS];

/* program counter */
SS_ADDR_TYPE regs_PC[MAX_PROCS];

/* Lock Flag */
SS_WORD_TYPE regs_LFR[MAX_PROCS];   /* Ihab */

/* Lock Address */
SS_ADDR_TYPE regs_LAR[MAX_PROCS];   /* Ihab */

/* initialize architected register state */
void
regs_init(int pid, SS_ADDR_TYPE init_stack_ptr, SS_ADDR_TYPE entry_ptr)
{
    int i;

    for (i=0; i<SS_NUM_REGS; i++)
regs_F[pid].l[i] = regs_R[pid][i] = 0;
regs_HI[pid] = 0;
regs_LO[pid] = 0;
regs_FCC[pid] = 0;
regs_PC[pid] = 0;
regs_LFR[pid] = 0;  /* Ihab */
regs_LAR[pid] = 0;  /* Ihab */
/* set up initial register state (pid determines how to set) */
if (pid == 0)
{
  regs_R[pid][SS_STACK_REGNO] = ld_environ_base;
  regs_PC[pid] = ld_prog_entry;
} else
{
  regs_R[pid][SS_STACK_REGNO] = init_stack_ptr;
  regs_PC[pid] = entry_ptr;
  /* get current value of global pointer */
  regs_R[pid][SS_GP_REGNO] = regs_R[0][SS_GP_REGNO];
}
/* dump all architected register state values to output stream STREAM */
void
regs_dump(FILE *stream, int pid)  /* output stream */
{
  int i;

  /* stderr is the default output stream */
  if (!stream)
    stream = stderr;

  /* dump processor register state */
  fprintf(stream, "Processor state:\n");
  fprintf(stream, "    PC: 0x%08x\n", regs_PC[pid]);
  for (i=0; i<SS_NUM_REGS; i += 2)
    {
      fprintf(stream, "    R[%2d]: %12d/0x%08x",
        i, regs_R[pid][i], regs_R[pid][i]);
      fprintf(stream, "    R[%2d]: %12d/0x%08x\n",
        i+1, regs_R[pid][i+1], regs_R[pid][i+1]);
    }
  fprintf(stream, "    HI:      %10d/0x%08x  LO:      %10d/0x%08x\n",
    regs_HI[pid], regs_HI[pid], regs_LO[pid], regs_LO[pid]);
  fprintf(stream, "    LF:      %10d/0x%08x  LA:      %x%08x\n", /* Ihab */
    regs_LFR[pid], regs_LAR[pid]);
  for (i=0; i<SS_NUM_REGS; i += 2)
    {
      fprintf(stream, "    F[%2d]: %12d/0x%08x",
        i, regs_F[pid].l[i], regs_F[pid].l[i]);
      fprintf(stream, "    F[%2d]: %12d/0x%08x\n",
        i+1, regs_F[pid].l[i+1], regs_F[pid].l[i+1]);
    }
  fprintf(stream, "    FCC:                0x%08x\n", regs_FCC[pid]);
}
/* syscall.c - proxy system call handler routines */
/* This file is an adaptation of the software in the SimpleScalar */
tool suite
/* originally written by Todd M. Austin for the Multiscalar Research */
Project
/* at the University of Wisconsin-Madison. */
/* The modifications were made by Naraig Manjikian at Queen's */
University,
/* Kingston, Ontario, Canada. */
/* The remainder of this header comment is unchanged from the */
original text. */

.................................................................
....... *
* Copyright (C) 1994, 1995, 1996, 1997 by Todd M. Austin *
* This source file is distributed "as is" in the hope that it will be *
useful. The tool set comes with no warranty, and no author or *
distributor accepts any responsibility for the consequences of its *
use. *
* Everyone is granted permission to copy, modify and redistribute *
this tool set under the following conditions: *
* This source code is distributed for non-commercial use only. *
Please contact the maintainer for restrictions applying to *
commercial use. *
* Permission is granted to anyone to make or distribute copies *
of this source code, either as received or modified, in any *
medium, provided that all copyright notices, permission and *
nonwarranty notices are preserved, and that the distributor *
grants the recipient permission for further redistribution as *
permitted by this document. *
* Permission is granted to distribute this file in compiled *
or executable form under the same conditions that apply for *
source code, provided that either: *
* A. it is accompanied by the corresponding machine-readable *
source code, *
* B. it is accompanied by a written offer, with no time limit, *
to give anyone a machine-readable copy of the corresponding *
source code in return for reimbursement of the cost of *
distribution. This written offer must permit verbatim *
duplication by anyone, or *
* C. it is distributed by someone who received only the *
executable form, and is accompanied by a copy of the *
written offer of source code that they received concurrently. *
* In other words, you are welcome to use, share and improve this
* source file. You are forbidden to forbid anyone else to use, share
* and improve what you give them.
*
* INTERNET: dburger@cs.wisc.edu
* US Mail: 1210 W. Dayton Street, Madison, WI 53706
*
* $Id: syscall.c,v 1.5 1997/04/16 22:12:17 taustin Exp taustin $
*
* $Log: syscall.c,v $
* Revision 1.5 1997/04/16 22:12:17 taustin
* added Ultrix host support
*
* Revision 1.4 1997/03/11 01:37:37 taustin
* updated copyright
* long/int tweaks made for ALPHA target support
* syscall structures are now more portable across platforms
* various target supports added
*
* Revision 1.3 1996/12/27 15:56:09 taustin
* updated comments
* removed system prototypes
*
* Revision 1.1 1996/12/05 18:52:32 taustin
* Initial revision
* *
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/param.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <setjmp.h>
#include <sys/limits.h>
#include <sys/select.h>

#if !defined(linux) && !defined(sparc) && !defined(hpux)
&& !defined(__hpux) && !defined(__CYGWIN32__) && !defined(ultrix)
#include <sys/select.h>
#endif

#ifndef linux
#include <bsd/sgtty.h>
#endif /* linux */

#if defined(__CYGWIN32__)
#include <sgtty.h>
#endif /* __CYGWIN32__ */
#if defined(__svr4__) || defined(__USLC__)
#include <dirent.h>
#else
#include <sys/dir.h>
#endif

/* dorks */
#undef NL0
#undef NL1
#undef CR0
#undef CR1
#undef CR2
#undef CR3
#undef TAB0
#undef TAB1
#undef TAB2
#undef XTABS
#undef BS0
#undef BS1
#undef FF0
#undef FF1
#undef ECHO
#undef NOFLSH
#undef TOSTOP
#undef FLUSHO
#undef PENDIN
#endif

#if defined(hpux) || defined(__hpux)
#undef CR0
#endif

#ifdef __FreeBSD__
#include <sys/ioctl_compat.h>
#else
#include <termio.h>
#endif

#if defined(hpux) || defined(__hpux)
/* et tu, dorks! */
#undef HUPCL
#undef ECHO
#undef B50
#undef B75
#undef B110
#undef B134
#undef B150
#undef B200
#undef B300
#undef B600
#undef B1200
#undef B1800
#undef B2400
#undef B4800
#undef B9600
#undef B19200
#undef B38400
#undef NL0
#undef NL1
#undef CR0
#define SS_NFLAGS (sizeof(ss_flag_table)/sizeof(ss_flag_table[0]))

/*-------------------------------------------------------------------
*/

/* the following are additions to support multiprocessing */

typedef struct _synchqnnode
{
  struct _synchqnnode *next;
  int pid;
  enum {FREE,
         WAITING_FOR_LOCK,
}
HOLDING_LOCK,
BLOCKED_ON_BARRIER,
BLOCKED_ON_SEMAPHORE} synch_state;
int synch_var; /* which synch var for above state */
} SynchQNode;

/* amount by which to increase array sizes for synchronization
variables */
#define LOCK_INCREMENT 1024
#define BARRIER_INCREMENT 64
#define SEMAPHORE_INCREMENT 64

/* a (reasonable?) upper bound on number of synch vars of each type */
#define SANITY_LIMIT 65535

/* each processor has a node that may appear in exactly _one_ queue
associated with a synchronization variable */
SynchQNode synchq[MAX_PROCS];

/* There is a queue pointer for each lock, barrier, or semaphore,
and each of these pointers is actually a pointer to the _tail_
item of the queue, and the 'next' field of the tail item points
to the head item:

array of
pointers
(one array
for each
type of
synch var)

The array of pointers may grow in size as more synchronization
variables
of a particular type are needed by the user program.
*/

static SynchQNode **locks;
static int num_allocated_locks = 0;
static int num_used_locks = 0;

static SynchQNode **barriers;
static int *barrier_counts;
static int num_allocated_barriers = 0;
static int num_used_barriers = 0;

static SynchQNode **semaphores;
static int *sema_counts;
static int num_allocated_semaphores = 0;
static int num_used_semaphores = 0;

#define NEW_THREAD_STACK_BASE 0x6ffffff0 /* arbitrary choice */
#define NEW_THREAD_STACK_SIZE (1 << 20) /* 1 Megabyte */

/* the following array of flags is used in the main simulation loop */
int active[MAX_PROCS];

/* the following is made volatile because it is changed with a new process */
/* and it is the upper bound of a 'for' loop nested in the main sim loop */
volatile int num_created_processes;

static int num_terminated_processes = 0;

/*-------------------------------------------------------------------
-*/

/* This function is called from ss_syscall() below when the user program executes the runtime library code for thread creation. The 'wrapper' function is also in the runtime library code, and is the function that is actually called when the new thread is created. The 'func_ptr' refers to the user program function that will then be called from the wrapper function in the runtime library. Upon return from the user function, the wrapper function will invoke the 'terminate thread' system call and thereby support the semantics of the return from the user function causing thread termination. */

static void CreateNewProcess (int pid,
    void (*func_ptr) (void),
    void (*wrapper) (void *))
{
    int new_pid;
    SS_ADDR_TYPE stack_top;

    new_pid = num_created_processes++;
    if (num_created_processes > MAX_PROCS)
        panic("too many processes; the limit is %d\n", MAX_PROCS);

    /* determine new top of stack; this pointer will be passed on to register initialization */
    stack_top = NEW_THREAD_STACK_BASE - (new_pid-1) * NEW_THREAD_STACK_SIZE;

    /* the entry point (PC value) for a new thread is the 'wrapper' function */
    regs_init (new_pid, stack_top, (SS_ADDR_TYPE) wrapper);

    /* after registers have been initialized, set register $a0 ($4)
with the user-supplied function pointer; wrapper will call it
*/
    regs_R[new_pid][4] = (SS_ADDR_TYPE) func_ptr;

    /* for synch queue node for this thread, set the pid and state
    fields */
    synchq[new_pid].pid = new_pid;
    synchq[new_pid].synch_state = FREE; /* not waiting, holding, or
    blocked */

    /* mark the new thread/process as active */
    active[new_pid] = 1;

    /* because the thread is now marked as active, and the total
    number of created processes has been incremented, the main simulation
    loop will shortly execute the first instruction of the new thread
    */
}

/*----------------------------------------------------------------------
   *-------------------------------------------------------------------
   * syscall proxy handler, architect registers and memory are assumed
   * to be precise when this function is called, register and memory are
   * updated with the results of the system call */
void
ss_syscall(mem_access_fn mem_fn, /* generic memory accessor */
           SS_INST_TYPE inst,  /* system call inst */
           int pid)   /* which processor */
{
    SS_WORD_TYPE syscode = regs_R[pid][2];

    switch (syscode)
    {
    case SS_SYS_exit:
    {
        ++num_terminated_processes;
        active[pid] = 0; /* mark this one as inactive */

        if (num_terminated_processes == num_created_processes)
        {
            /* exit jumps to the target set in main() */
            longjmp(sim_exit_buf, /* exitcode + fudge
                      */regs_R[pid][4]+1);
        }
    } /* else we must wait until last process finishes
        (this one has been marked inactive, so when we return to
        the main simulation loop, no further instructions
        will be fetched for this process) */
    break;

#if 0
    case SS_SYS_fork:
        break;
#endif
#if 0
    case SS_SYS_vfork:
        break;
#endif

case SS_SYS_read:
    {
        char *buf;
        
        /* allocate same-sized input buffer in host memory */
        if (!buf = (char *)calloc(/*nbytes*/regs_R[pid][6],
                       sizeof(char)))
            fatal("out of memory in SYS_read");
        
        /* read data from file */
        /*nread*/regs_R[pid][2] = read(/*fd*/regs_R[pid][4], buf,
            /*nbytes*/regs_R[pid][6]);

        /* check for error condition */
        if (regs_R[pid][2] != -1)
            regs_R[pid][7] = 0;
        else
            {
                /* got an error, return details */
                regs_R[pid][2] = errno;
                regs_R[pid][7] = 1;
            }

        /* copy results back into host memory */
        mem_bcopy(mem_fn, Write, /*buf*/regs_R[pid][5], buf,
            /*nread*/regs_R[pid][2]);

        /* done with input buffer */
        free(buf);
    }
    break;

case SS_SYS_write:
    {
        char *buf;
        
        /* allocate same-sized output buffer in host memory */
        if (!buf = (char *)calloc(/*nbytes*/regs_R[pid][6],
                       sizeof(char)))
            fatal("out of memory in SYS_write");
        
        /* copy inputs into host memory */
        mem_bcopy(mem_fn, Read, /*buf*/regs_R[pid][5], buf,
            /*nbytes*/regs_R[pid][6]);

        /* write data to file */
        /*nwritten*/regs_R[pid][2] = write(/*fd*/regs_R[pid][4],
            buf, /*nbytes*/regs_R[pid][6]);

        /* check for an error condition */
        if (regs_R[pid][2] == regs_R[pid][6])
            /*result*/regs_R[pid][7] = 0;
        else
            {
                /* got an error, return details */

                /* result*/regs_R[pid][7] = 1;
            }
    }
    break;
regs_R[pid][2] = errno;
regs_R[pid][7] = 1;
}

/* done with output buffer */
free(buf);
} break;

case SS_SYS_open:
{
char buf[MAXBUFSIZE];
unsigned int i;
int ss_flags = regs_R[pid][5], local_flags = 0;

/* translate open(2) flags */
for (i=0; i<SS_NFLAGS; i++)
{
  if (ss_flags & ss_flag_table[i].ss_flag)
  {
    ss_flags &= -ss_flag_table[i].ss_flag;
    local_flags |= ss_flag_table[i].local_flag;
  }
}
/* any target flags left? */
if (ss_flags != 0)
  fatal("syscall: open: cannot decode flags: 0x%08x", ss_flags);

/* copy filename to host memory */
mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);

/* open the file */
/*fd*/regs_R[pid][2] = open(buf, local_flags,
/*mode*/regs_R[pid][6]);

/* check for an error condition */
if (regs_R[pid][2] != -1)
  regs_R[pid][7] = 0;
else
  {
    /* got an error, return details */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
  }
break;

case SS_SYS_close:
/* don't close stdin, stdout, or stderr as this messes up sim logs */
if (//*fd*/regs_R[pid][4] == 0 || /*fd*/regs_R[pid][4] == 1 ||
/*fd*/regs_R[pid][4] == 2)
  {
    regs_R[pid][7] = 0;
    break;
  }

/* close the file */
regs_R[pid][2] = close(/*fd*/regs_R[pid][4]);
/** check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
break;

case SS_SYS_creat:
    {
        char buf[MAXBUFSIZE];
        /* copy filename to host memory */
        mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);
        /* create the file */
        /*fd*/regs_R[pid][2] = creat(buf, /*mode*/regs_R[pid][5]);
        /* check for an error condition */
        if (regs_R[pid][2] != -1)
            regs_R[pid][7] = 0;
        else
            {
                /* got an error, return details */
                regs_R[pid][2] = errno;
                regs_R[pid][7] = 1;
            }
    }
break;

case SS_SYS_unlink:
    {
        char buf[MAXBUFSIZE];
        /* copy filename to host memory */
        mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);
        /* delete the file */
        /*result*/regs_R[pid][2] = unlink(buf);
        /* check for an error condition */
        if (regs_R[pid][2] != -1)
            regs_R[pid][7] = 0;
        else
            {
                /* got an error, return details */
                regs_R[pid][2] = errno;
                regs_R[pid][7] = 1;
            }
    }
break;

case SS_SYS_chdir:
    {
        char buf[MAXBUFSIZE];
/* copy filename to host memory */
mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);

/* change the working directory */
/*result*/regs_R[pid][2] = chdir(buf);

/* check for an error condition */
if (regs_R[pid][2] != -1)
  regs_R[pid][7] = 0;
else
  { /* got an error, return details */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
  }
}
break;

case SS_SYS_chmod:
  {
    char buf[MAXBUFSIZE];

    /* copy filename to host memory */
    mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);

    /* chmod the file */
    /*result*/regs_R[pid][2] = chmod(buf, /*mode*/regs_R[pid][5]);

    /* check for an error condition */
    if (regs_R[pid][2] != -1)
      regs_R[pid][7] = 0;
    else
      { /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
      }
  }
break;

case SS_SYS_chown:
  {
    char buf[MAXBUFSIZE];

    /* copy filename to host memory */
    mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);

    /* chown the file */
    /*result*/regs_R[pid][2] = chown(buf, /*owner*/regs_R[pid][5],
                          /*group*/regs_R[pid][6]);

    /* check for an error condition */
    if (regs_R[pid][2] != -1)
      regs_R[pid][7] = 0;
    else
      { /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
      }
  }
}

break;

case SS_SYS_brk:
{
    SS_ADDR_TYPE addr;
    /* round the new heap pointer to its page boundary */
    addr = ROUND_UP(/*base*/regs_R[pid][4], SS_PAGE_SIZE);
    /* check whether heap area has merged with stack area */
    if (addr >= mem_brk_point && addr < (unsigned int)regs_R[pid][29])
    {
        regs_R[pid][2] = 0;
        regs_R[pid][7] = 0;
        mem_brk_point = addr;
    }
    else
    {
      /* out of address space, indicate error */
        regs_R[pid][2] = ENOMEM;
        regs_R[pid][7] = 1;
    }
}
break;

case SS_SYS_lseek:
    /* seek into file */
    regs_R[pid][2] = lseek(/*fd*/regs_R[pid][4],
    /*off*/regs_R[pid][5], /*dir*/regs_R[pid][6]);
    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
      /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_getpid:
    /* get the simulator process id */
    /*result*/regs_R[pid][2] = getpid();
    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
      /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_getuid:
    /* get current user id */
/first result*/regs_R[pid][2] = getuid();
/* get effective user id */
/*second result*/regs_R[pid][3] = geteuid();

/* check for an error condition */
if (regs_R[pid][2] != -1)
regs_R[pid][7] = 0;
else
{
/* got an error, return details */
regs_R[pid][2] = errno;
regs_R[pid][7] = 1;
}
break;

case SS_SYS_access:
{
char buf[MAXBUFSIZE];
/* copy filename to host memory */
mem_strcpy(mem_fn, Read, /* fName */regs_R[pid][4], buf);
/* check access on the file */
/*result*/regs_R[pid][2] = access(buf, /* mode */regs_R[pid][5]);
/* check for an error condition */
if (regs_R[pid][2] != -1)
regs_R[pid][7] = 0;
else
{
/* got an error, return details */
regs_R[pid][2] = errno;
regs_R[pid][7] = 1;
}
}
break;

case SS_SYS_stat:
case SS_SYS_lstat:
{
char buf[MAXBUFSIZE];
struct ss_statbuf ss_sbuf;
struct stat sbuf;
/* copy filename to host memory */
mem_strcpy(mem_fn, Read, /* fName */regs_R[pid][4], buf);
/* stat() the file */
if (syscode == SS_SYS_stat)
/*result*/regs_R[pid][2] = stat(buf, &sbuf);
else /* syscode == SS_SYS_lstat */
/*result*/regs_R[pid][2] = lstat(buf, &sbuf);
/* check for an error condition */
if (regs_R[pid][2] != -1)
regs_R[pid][7] = 0;
else
{
/* got an error, return details */

```c
regs_R[pid][2] = errno;
regs_R[pid][7] = 1;
}

/* translate from host stat structure to target format */
ss_sbuf.ss_st_dev = SWAP_HALF(sbuf.st_dev);
ss_sbuf.ss_st_ino = SWAP_WORD(sbuf.st_ino);
ss_sbuf.ss_st_mode = SWAP_HALF(sbuf.st_mode);
ss_sbuf.ss_st_nlink = SWAP_HALF(sbuf.st_nlink);
ss_sbuf.ss_st_uid = SWAP_HALF(sbuf.st_uid);
ss_sbuf.ss_st_gid = SWAP_HALF(sbuf.st_gid);
ss_sbuf.ss_st_rdev = SWAP_HALF(sbuf.st_rdev);
ss_sbuf.ss_st_size = SWAP_WORD(sbuf.st_size);
ss_sbuf.ss_st_atime = SWAP_WORD(sbuf.st_atime);
ss_sbuf.ss_st_mtime = SWAP_WORD(sbuf.st_mtime);
ss_sbuf.ss_st_ctime = SWAP_WORD(sbuf.st_ctime);
ss_sbuf.ss_st_blksize = SWAP_WORD(sbuf.st_blksize);
ss_sbuf.ss_st_blocks = SWAP_WORD(sbuf.st_blocks);

/* copy stat() results to simulator memory */
mem_bcopy(mem_fn, Write, /*sbuf*/regs_R[pid][5],
        &ss_sbuf, sizeof(struct ss_statbuf));
break;

case SS_SYS_dup:
    /* dup() the file descriptor */
    /*fd*/regs_R[pid][2] = dup(/*fd*/regs_R[pid][4]);

    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
        {
            /* got an error, return details */
            regs_R[pid][2] = errno;
            regs_R[pid][7] = 1;
        }
break;

case SS_SYS_pipe:
{
    int fd[2];

    /* copy pipe descriptors to host memory */
    mem_bcopy(mem_fn, Read, /*fd's*/regs_R[pid][4], fd,
        sizeof(fd));

    /* create a pipe */
    /*result*/regs_R[pid][7] = pipe(fd);

    /* copy descriptor results to result registers */
    /*pipe1*/regs_R[pid][2] = fd[0];
    /*pipe 2*/regs_R[pid][3] = fd[1];

    /* check for an error condition */
    if (regs_R[pid][7] == -1)
        {
            regs_R[pid][2] = errno;
            regs_R[pid][7] = 1;
```

case SS_SYS_getgid:
    /* get current group id */
    /* first result*/regs_R[pid][2] = getgid();
    /* get current effective group id */
    /* second result*/regs_R[pid][3] = getegid();
    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_ioctl:
    {
        char buf[NUM_IOCTL_BYTES];
        int local_req = 0;
        /* convert target ioctl() request to host ioctl() request
        values */
        switch (/*req*/regs_R[pid][5]) {
        /* #if !defined(__CYGWIN32__) */
        case SS_IOCTL_TIOCGETP:
            local_req = TIOCGETP;
            break;
        case SS_IOCTL_TIOCSETP:
            local_req = TIOCSETP;
            break;
        case SS_IOCTL_TCGETP:
            local_req = TIOCGETP;
            break;
        /* #endif */
        #ifdef TCGETA
        case SS_IOCTL_TCGETA:
            local_req = TCGETA;
            break;
        #endif
        #ifdef TIOCGLTC
        case SS_IOCTL_TIOCGLTC:
            local_req = TIOCGLTC;
            break;
        #endif
        #ifdef TIOCSLTC
        case SS_IOCTL_TIOCSLTC:
            local_req = TIOCSLTC;
            break;
        #endif
        case SS_IOCTL_TIOCGWINSZ:
            local_req = TIOCGWINSZ;
            break;
        #ifdef TCSETAW
        }
case SS_IOCTL_TCSETAW:
    local_req = TCSETAW;
    break;
#endif
#endif
#endif TIOCGETC
    case SS_IOCTL_TIOCGETC:
    local_req = TIOCGETC;
    break;
#endif
#endif TIOCSETC
    case SS_IOCTL_TIOCSETC:
    local_req = TIOCSETC;
    break;
#endif
#endif TIOCLBIC
    case SS_IOCTL_TIOCLBIC:
    local_req = TIOCLBIC;
    break;
#endif
#endif TIOCLBIS
    case SS_IOCTL_TIOCLBIS:
    local_req = TIOCLBIS;
    break;
#endif
#endif TIOCLGET
    case SS_IOCTL_TIOCLGET:
    local_req = TIOCLGET;
    break;
#endif
#endif TIOCLSET
    case SS_IOCTL_TIOCLSET:
    local_req = TIOCLSET;
    break;
#endif

if (!local_req)
{
    /* FIXME: could not translate the ioctl() request, just
    warn user
    and ignore the request */
    warn("syscall: ioctl: ioctl code not supported d=%d, 
    req=%d", 
    regs_R[pid][4], regs_R[pid][5]);
    regs_R[pid][2] = 0;
    regs_R[pid][7] = 0;
}
else
{
    /* ioctl() code was successfully translated to a host code */
    /* if arg ptr exists, copy NUM_IOCTL_BYTES bytes to host
    mem */
    if (/*argp*/regs_R[pid][6] != 0)
        mem_bcopy(mem_fn, Read, /*argp*/regs_R[pid][6], buf,
        NUM_IOCTL_BYTES);
    /* perform the ioctl() call */
/*result*/regs_R[pid][2] = ioctl(/*fd*/regs_R[pid][4], local_req, buf);

/* if arg ptr exists, copy NUM_IOCTL_BYTES bytes from host mem */
if (/*argp*/regs_R[pid][6] != 0)
    mem_bcopy(mem_fn, Write, regs_R[pid][6], buf, NUM_IOCTL_BYTES);

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
break;

case SS_SYS_fstat:
{
    struct ss_statbuf ss_sbuf;
    struct stat sbuf;

    /* fstat() the file */
    /*result*/regs_R[pid][2] = fstat(/*fd*/regs_R[pid][4], &sbuf);

    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
        {
            /* got an error, return details */
            regs_R[pid][2] = errno;
            regs_R[pid][7] = 1;
        }

    /* translate the stat structure to host format */
    ss_sbuf.ss_st_dev = SWAP_HALF(sbuf.st_dev);
    ss_sbuf.ss_st_ino = SWAP_WORD(sbuf.st_ino);
    ss_sbuf.ss_st_mode = SWAP_HALF(sbuf.st_mode);
    ss_sbuf.ss_st_nlink = SWAP_HALF(sbuf.st_nlink);
    ss_sbuf.ss_st_uid = SWAP_HALF(sbuf.st_uid);
    ss_sbuf.ss_st_gid = SWAP_HALF(sbuf.st_gid);
    ss_sbuf.ss_st_rdev = SWAP_HALF(sbuf.st_rdev);
    ss_sbuf.ss_st_size = SWAP_WORD(sbuf.st_size);
    ss_sbuf.ss_st_atime = SWAP_WORD(sbuf.st_atime);
    ss_sbuf.ss_st_mtime = SWAP_WORD(sbuf.st_mtime);
    ss_sbuf.ss_st_ctime = SWAP_WORD(sbuf.st_ctime);
    ss_sbuf.ss_st_blksize = SWAP_WORD(sbuf.st_blksize);
    ss_sbuf.ss_st_blocks = SWAP_WORD(sbuf.st_blocks);

    /* copy fstat() results to simulator memory */
    mem_bcopy(mem_fn, Write, /*sbuf*/regs_R[pid][5], &ss_sbuf, sizeof(struct ss_statbuf));
}
break;
case SS_SYS_getpagesize:
    /* get target pagesize */
    regs_R[pid][2] = /* was: getpagesize() */ SS_PAGE_SIZE;

    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_setitimer:
    /* FIXME: the sigvec system call is ignored */
    regs_R[pid][2] = regs_R[pid][7] = 0;
    warn("syscall: setitimer ignored");
    break;

case SS_SYS_getdtablesize:
    #if defined(_AIX)
        /* get descriptor table size */
        regs_R[pid][2] = getdtablesize();

        /* check for an error condition */
        if (regs_R[pid][2] != -1)
            regs_R[pid][7] = 0;
        else
        {
            /* got an error, return details */
            regs_R[pid][2] = errno;
            regs_R[pid][7] = 1;
        }
    #elif defined(__CYGWIN32__)
    {
        /* no comparable system call found, try some reasonable defaults */
        warn("syscall: called getdtablesize\n");
        regs_R[pid][2] = 16;
        regs_R[pid][7] = 0;
    }
    #elif defined(ultrix)
    {
        /* no comparable system call found, try some reasonable defaults */
        warn("syscall: called getdtablesize\n");
        regs_R[pid][2] = 16;
        regs_R[pid][7] = 0;
    }
    #else
    {
        struct rlimit rl;

        /* get descriptor table size in rlimit structure */
        if (getrlimit(RLIMIT_NOFILE, &rl) != -1)
        {
            regs_R[pid][2] = rl.rlim_cur;
        }
    }

136
regs_R[pid][7] = 0;
}
else
{
    /* got an error, return details */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}
}
#endif
break;

case SS_SYS_dup2:
    /* dup2() the file descriptor */
    regs_R[pid][2] = dup2(/* fd1 */regs_R[pid][4], /* fd2 */regs_R[pid][5]);
    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_fcntl:
    /* get fcntl() information on the file */
    regs_R[pid][2] = fcntl(/*fd*/regs_R[pid][4], /*cmd*/regs_R[pid][5], /*arg*/regs_R[pid][6]);
    /* check for an error condition */
    if (regs_R[pid][2] != -1)
        regs_R[pid][7] = 0;
    else
    {
        /* got an error, return details */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }
    break;

case SS_SYS_select:
{
    fd_set readfd, writefd, exceptfd;
    fd_set *readfdp, *writefdp, *exceptfdp;
    struct timeval timeout, *timeoutp;
    SS_WORD_TYPE param5;
    /* FIXME: swap words? */

    /* read the 5th parameter (timeout) from the stack */
    mem_bcopy(mem_fn, Read, regs_R[pid][29]+16, &param5, sizeof(SS_WORD_TYPE));

    /* copy read file descriptor set into host memory */
    if (/*readfd*/regs_R[pid][5] != 0)
mem_bcopy(mem_fn, Read, /*readfd*/regs_R[pid][5],
           &readfd, sizeof(fd_set));
readfdp = &readfd;
}
else
readfdp = NULL;

/* copy write file descriptor set into host memory */
if (/*writefd*/regs_R[pid][6] != 0)
{
    mem_bcopy(mem_fn, Read, /*writefd*/regs_R[pid][6],
               &writefd, sizeof(fd_set));
    writefdp = &writefd;
}
else
writefdp = NULL;

/* copy exception file descriptor set into host memory */
if (/*exceptfd*/regs_R[pid][7] != 0)
{
    mem_bcopy(mem_fn, Read, /*exceptfd*/regs_R[pid][7],
               &exceptfd, sizeof(fd_set));
    exceptfdp = &exceptfd;
}
else
exceptfdp = NULL;

/* copy timeout value into host memory */
if (/*timeout*/param5 != 0)
{
    mem_bcopy(mem_fn, Read, /*timeout*/param5,
               &timeout, sizeof(struct timeval));
    timeoutp = &timeout;
}
else
timeoutp = NULL;

#if defined(hpux) || defined(__hpux)
/* select() on the specified file descriptors */
/*result*/regs_R[pid][2] = select(/*nfd*/regs_R[pid][4],
      (int *)readfdp, (int *)writefdp, (int *)exceptfdp,
      timeoutp);
#else
/* select() on the specified file descriptors */
/*result*/regs_R[pid][2] = select(/*nfd*/regs_R[pid][4], readfdp, writefdp, exceptfdp,
      timeoutp);
#endif

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
{
    /* got an error, return details */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
/* copy read file descriptor set to target memory */
if (/*readfd*/regs_R[pid][5] != 0)
    mem_bcopy(mem_fn, Write, /*readfd*/regs_R[pid][5],
               &readfd, sizeof(fd_set));

/* copy write file descriptor set to target memory */
if (/*writefd*/regs_R[pid][6] != 0)
    mem_bcopy(mem_fn, Write, /*writefd*/regs_R[pid][6],
               &writefd, sizeof(fd_set));

/* copy exception file descriptor set to target memory */
if (/*exceptfd*/regs_R[pid][7] != 0)
    mem_bcopy(mem_fn, Write, /*exceptfd*/regs_R[pid][7],
               &exceptfd, sizeof(fd_set));

/* copy timeout value result to target memory */
if (/* timeout */param5 != 0)
    mem_bcopy(mem_fn, Write, /*timeout*/param5,
               &timeout, sizeof(struct timeval));
}
break;

case SS_SYS_sigvec:
    /* FIXME: the sigvec system call is ignored */
    regs_R[pid][2] = regs_R[pid][7] = 0;
    warn("syscall: sigvec ignored");
    break;

case SS_SYS_sigblock:
    /* FIXME: the sigblock system call is ignored */
    regs_R[pid][2] = regs_R[pid][7] = 0;
    warn("syscall: sigblock ignored");
    break;

case SS_SYS_sigsetmask:
    /* FIXME: the sigsetmask system call is ignored */
    regs_R[pid][2] = regs_R[pid][7] = 0;
    warn("syscall: sigsetmask ignored");
    break;
#endif

    case SS_SYS_gettimeofday:
    {
        struct ss_timeval ss_tv;
        struct timeval tv, *tvp;
        struct ss_timezone ss_tz;
        struct timezone tz, *tzp;

        if (/*timeval*/regs_R[pid][4] != 0)
        {
            /* copy timeval into host memory */
            mem_bcopy(mem_fn, Read, /*timeval*/regs_R[pid][4],
                      &ss_tv, sizeof(struct ss_timeval));

            /* convert target timeval structure to host format */
            tv.tv_sec = SWAP_WORD(ss_tv.ss_tv_sec);
tv.tv_usec = SWAP_WORD(ss_tv.ss_tv_usec);
tvp = &tv;
}
else
tvp = NULL;

if (/*timezone*/regs_R[pid][5] != 0)
{
    /* copy timezone into host memory */
    mem_bcopy(mem_fn, Read, /*timezone*/regs_R[pid][5],
               &ss_tz, sizeof(struct ss_timezone));

    /* convert target timezone structure to host format */
    tz.tz_minuteswest = SWAP_WORD(ss_tz.ss_tz_minuteswest);
    tz.tz_dsttime = SWAP_WORD(ss_tz.ss_tz_dsttime);
    tzp = &tz;
}
else
tzp = NULL;

/* get time of day */
/*result*/regs_R[pid][2] = gettimeofday(tvp, tzp);

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
{
    /* got an error, indicate result */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}

if (/*timeval*/regs_R[pid][4] != 0)
{
    /* convert host timeval structure to target format */
    ss_tv.ss_tv_sec = SWAP_WORD(tv.tv_sec);
    ss_tv.ss_tv_usec = SWAP_WORD(tv.tv_usec);

    /* copy timeval to target memory */
    mem_bcopy(mem_fn, Write, /*timeval*/regs_R[pid][4],
               &ss_tv, sizeof(struct ss_timeval));
}

if (/*timezone*/regs_R[pid][5] != 0)
{
    /* convert host timezone structure to target format */
    ss_tz.ss_tz_minuteswest = SWAP_WORD(tz.tz_minuteswest);
    ss_tz.ss_tz_dsttime = SWAP_WORD(tz.tz_dsttime);

    /* copy timezone to target memory */
    mem_bcopy(mem_fn, Write, /*timezone*/regs_R[pid][5],
               &ss_tz, sizeof(struct ss_timezone));
}
}
break;

case SS_SYS_getrusage:
#if defined(__svr4__) || defined(__USLC__) || defined(hpux) || defined(__hpux) || defined(_AIX)
struct tms tms_buf;
struct ss_rusage rusage;

/* get user and system times */
if (times(&tms_buf) != -1)
{
    /* no error */
    regs_R[pid][2] = 0;
    regs_R[pid][7] = 0;
}
else
{
    /* got an error, indicate result */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}

/* initialize target rusage result structure */
#if defined(__svr4__)
    memset(&rusage, '\0', sizeof(struct ss_rusage));
#else /* !defined(__svr4__) */
    bzero(&rusage, sizeof(struct ss_rusage));
#endif

/* convert from host rusage structure to target format */
rusage.ss_ru_utime.ss_tv_sec = tms_buf.tms_utime/CLK_TCK;
rusage.ss_ru_utime.ss_tv_sec = SWAP_WORD(rusage.ss_ru_utime.ss_tv_sec);
rusage.ss_ru_utime.ss_tv_usec = 0;
rusage.ss_ru_stime.ss_tv_sec = tms_buf.tms_stime/CLK_TCK;
rusage.ss_ru_stime.ss_tv_sec = SWAP_WORD(rusage.ss_ru_stime.ss_tv_sec);
rusage.ss_ru_stime.ss_tv_usec = 0;

/* copy rusage results into target memory */
mem_bcopy(mem_fn, Write, /*rusage*/regs_R[pid][5], 
&rusage, sizeof(struct ss_rusage));
#endif

#endif

/* get rusage information */
/*result*/regs_R[pid][2] = getrusage(/*who*/regs_R[pid][4], 
&local_rusage);

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
{
    /* got an error, indicate result */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}

/* convert from host rusage structure to target format */
rusage.ss_ru_utime.ss_tv_sec = local_rusage.ru_utime.tv_sec;
rusage.ss_ru_utime.ss_tv_usec = local_rusage.ru_utime.tv_usec;
rusage.ss_ru_utime.ss_tv_usec =
SWAP_WORD(local_rusage.ru_utime.tv_usec);

rusage.ss_ru_stime.ss_tv_usec = local_rusage.ru_stime.tv_usec;
rusage.ss_ru_stime.ss_tv_usec =
SWAP_WORD(local_rusage.ru_stime.tv_usec);

rusage.ss_ru_maxrss = SWAP_WORD(local_rusage.ru_maxrss);
rusage.ss_ru_ixrss = SWAP_WORD(local_rusage.ru_ixrss);
rusage.ss_ru_idrss = SWAP_WORD(local_rusage.ru_idrss);
rusage.ss_ru_minflt = SWAP_WORD(local_rusage.ru_minflt);
rusage.ss_ru_majflt = SWAP_WORD(local_rusage.ru_majflt);

/* copy rusage results into target memory */
mem_bcopy(mem_fn, Write, /*rusage*/regs_R[pid][5],
&rusage, sizeof(struct ss_rusage));

#elif defined(__CYGWIN32__)
   warn("syscall: called getrusage\n");
   regs_R[pid][7] = 0;
#else
   #error No getrusage() implementation!
#endif
break;

case SS_SYS_writev:
{
   int i;
   char *buf;
   struct iovec *iov;

   /* allocate host side I/O vectors */
   iov =
   (struct iovec *)&malloc(/*iovcnt*/regs_R[pid][6] *
   sizeof(struct iovec));
   if (!iov)
      fatal("out of virtual memory in SYS_writev");

   /* copy target side pointer data into host side vector */
   mem_bcopy(mem_fn, Read, /*iov*/regs_R[pid][5],
   iov, /*iovcnt*/regs_R[pid][6] * sizeof(struct iovec));

   /* copy target side I/O vector buffers to host memory */
   for (i=0; i < /*iovcnt*/regs_R[pid][6]; i++)
      { 
         iov[i].iov_base = (char *)SWAP_WORD((unsigned)iov[i].iov_base);
iov[i].iov_len = SWAP_WORD(iov[i].iov_len);
if (iov[i].iov_base != NULL)
{
    buf = (char *)calloc(iov[i].iov_len, sizeof(char));
    if (!buf)
        fatal("out of virtual memory in SYS_writev");
    mem_bcopy(mem_fn, Read, (SS_ADDR_TYPE)iov[i].iov_base,
              buf, iov[i].iov_len);
    iov[i].iov_base = buf;
}

/* perform the vector'ed write */
*result*/regs_R[pid][2] =
writev(*fd*/regs_R[pid][4], iov, /*iovcnt*/regs_R[pid][6]);

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
    /* got an error, indicate results */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;

/* free all the allocated memory */
for (i=0; i < /*iovcnt*/regs_R[pid][6]; i++)
{
    if (iov[i].iov_base)
        free(iov[i].iov_base);
    iov[i].iov_base = NULL;
}
free(iov);
break;

case SS_SYS_utimes:
{
    char buf[MAXBUFSIZE];

    /* copy filename to host memory */
    mem_strcpy(mem_fn, Read, /*fname*/regs_R[pid][4], buf);
    if (*/*timeval*/regs_R[pid][5] == 0)
    {
        #if defined(hpux) || defined(__hpux) || defined(__i386__)
        /* no utimes() in hpux, use utime() instead */
        *result*/regs_R[pid][2] = utime(buf, NULL);
        #elif defined(__svr4__) || defined(__USLC__) || defined(_AIX)
        /*result*/regs_R[pid][2] = utimes(buf, NULL);
        #elif defined(__CYGWIN32__)
        warn("syscall: called utimes\n");
        #else
        #error No utimes() implementation!
        #endif
    }
}
else
{
    struct ss_timeval ss_tval[2];
    struct timeval tval[2];

    /* copy timeval structure to host memory */
    mem_bcopy(mem_fn, Read, /*timeout*/regs_R[pid][5],
               ss_tval, 2*sizeof(struct ss_timeval));

    /* convert timeval structure to host format */
    tval[0].tv_sec = SWAP_WORD(ss_tval[0].ss_tv_sec);
    tval[0].tv_usec = SWAP_WORD(ss_tval[0].ss_tv_usec);
    tval[1].tv_sec = SWAP_WORD(ss_tval[1].ss_tv_sec);
    tval[1].tv_usec = SWAP_WORD(ss_tval[1].ss_tv_usec);

    #if defined(hpux) || defined(__hpux)
    /* no utimes() in hpux, use utime() instead */
    {
        struct utimbuf ubuf;
        ubuf.actime = tval[0].tv_sec;
        ubuf.modtime = tval[1].tv_sec;

        /* result */regs_R[pid][2] = utime(buf, &ubuf);
    }
    #elif defined(__svr4__) || defined(__USLC__) || defined(unix) ||
    defined(_AIX)
    /* result */regs_R[pid][2] = utimes(buf, tval);
    #elif defined(__CYGWIN32__)  
        warn("syscall: called utimes\n");
    #else
    #error No utimes() implementation!
    #endif
}
/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
{
    /* got an error, indicate results */
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}
}  
break;

case SS_SYS_getrlimit:
case SS_SYS_setrlimit:
    #if defined(__CYGWIN32__)  
    { 
        warn("syscall: called get/setrlimit\n");
        regs_R[pid][7] = 0;
    }
    #else
    {  
        struct rlimit ss_rl;
        struct rlimit rl;

        /* copy rlimit structure to host memory */
        }  

mem_bcopy(mem_fn, Read, /*rlimit*/regs_R[pid][5], &ss_rl, sizeof(struct ss_rlimit));

/* convert rlimit structure to host format */
rl.rlim_cur = SWAP_WORD(ss_rl.rlim_cur);
rl.rlim_max = SWAP_WORD(ss_rl.rlim_max);

/* get rlimit information */
if (syscode == SS_SYS_getrlimit)
    /*result*/regs_R[pid][2] = getrlimit(regs_R[pid][4], &rl);
else /* syscode == SS_SYS_setrlimit */
    /*result*/regs_R[pid][2] = setrlimit(regs_R[pid][4], &rl);

/* check for an error condition */
if (regs_R[pid][2] != -1)
    regs_R[pid][7] = 0;
else
    {
        /* got an error, indicate results */
        regs_R[pid][2] = errno;
        regs_R[pid][7] = 1;
    }

/* convert rlimit structure to target format */
ss_rl.rlim_cur = SWAP_WORD(rl.rlim_cur);
ss_rl.rlim_max = SWAP_WORD(rl.rlim_max);

/* copy rlimit structure to target memory */
mem_bcopy(mem_fn, Write, /*rlimit*/regs_R[pid][5], &ss_rl, sizeof(struct ss_rlimit));

#endif
break;

#if 0
case SS_SYS_getdirent:
    /* FIXME: this is currently broken due to incompatabilities in
    disk directory formats */
    {
        unsigned int i;
        char *buf;
        int base;

        buf = (char *)calloc(/* nbytes */regs_R[pid][6] + 1,
        sizeof(char));
        if (!buf)
            fatal("out of memory in SYS_getdirent");

        /* copy in */
        for (i=0; i</* nbytes */regs_R[pid][6]; i++)
            (*maf)(Read, /* buf */regs_R[pid][5] + i, (unsigned char *)
            &buf[i], 1);
        (*maf)(Read, /* basep */regs_R[pid][7], (unsigned char *)&base,
        4);

        /*cc*/regs_R[pid][2] =
        getdirent(/*fd*/regs_R[pid][4], buf,
        /*nbytes*/regs_R[pid][6], &base);

        if (regs_R[pid][2] != -1)

regs_R[pid][7] = 0;
else
{
    regs_R[pid][2] = errno;
    regs_R[pid][7] = 1;
}

/* copy out */
for (i=0; i</* nbytes */regs_R[pid][6]; i++)
    (*maf)(Write, /* buf */regs_R[pid][5]+i, (unsigned char *)&buf[i], 1);
    (*maf)(Write, /* basep */regs_R[pid][7], (unsigned char *)&base, 4);

free(buf);
}
break;
#endif

case SS_MP_ACQUIRE_LOCK:
{
    int lock_id = regs_R[pid][4];
    //unsigned long lock_ptr = regs_R[pid][4];
    /* check if attempting to acquire another or same lock */
    if (synchq[pid].synch_state == HOLDING_LOCK)
        fatal("pid %d attempting to get lock %d"
             " but already has lock %d", pid, lock_id,
             synchq[pid].synch_var);
    else if (synchq[pid].synch_state != FREE)
        fatal("pid %d attempting to get lock %d"
             " but already blocked on barrier or semaphore",
             pid, lock_id);

    /* check if at least one item in queue, which means lock is already held by a thread */
    if (locks[lock_id])
    {
        /* add ourselves to the end of the queue for this lock
        */
        synchq[pid].next = locks[lock_id]->next; /* point to
        head */
        locks[lock_id]->next = &synchq[pid]; /* add to tail */
        locks[lock_id] = &synchq[pid]; /* update pointer to
        tail */

        /* set synch state and var */
        synchq[pid].synch_state = WAITING_FOR_LOCK;
        synchq[pid].synch_var = lock_id;

        /* mark as inactive until lock is acquired */
        active[pid] = 0;
        debug("pid %d must wait for lock %d", pid, lock_id);
    }
    else
    {
        /* we will get the lock; put ourselves at head of queue
        */
synchq[pid].next = &synchq[pid]; /* tail points to head 

locks[lock_id] = &synchq[pid]; /* update pointer to 

tail */

/* set synch state and var */
synchq[pid].synch_state = HOLDING_LOCK;
synchq[pid].synch_var = lock_id;
regs_LFR[pid] = 1;
ddebug ("pid %d acquired lock %d", pid, lock_id);

break;

case SS_MP_RELEASE_LOCK:
{
    int lock_id = regs_R[pid][4];

    /* sanity check: we must be at the head of the queue 
    (if lock is free, any thread can release) */
    if (locks[lock_id] && locks[lock_id]->next->pid != pid)
    {
        fatal ("attempt to release lock held by another 
                thread");
    }
    /* set synch state of releasing thread*/
synchq[pid].synch_state = FREE;

    /* check if current lock holder is the only one in queue 
    */
    if (locks[lock_id]->next == locks[lock_id])
    {
        locks[lock_id] = NULL; /* lock is now free */
        debug ("pid %d released lock %d", pid, lock_id);
    }
    else /* at least one other thread waiting for lock */
    {
        /* advance head of queue */
        locks[lock_id]->next = locks[lock_id]->next->next;
        active[locks[lock_id]->next->pid] = 1; /* mark as 
            active */
        /* set synch state and var of thread being given the 
            lock */
        locks[lock_id]->next->synch_state = HOLDING_LOCK;
        locks[lock_id]->next->synch_var = lock_id;
        debug ("pid %d given lock %d by pid %d", 
                locks[lock_id]->next->pid, lock_id, pid);
    }
}
break;

case SS_MP_INIT_LOCK:
{
    SS_ADDR_TYPE lock_ptr = regs_R[pid][4];
    int i;
    int lock_id = num_used_locks++;

if (num_used_locks > SANITY_LIMIT)
  fatal ("do you _really_ need %d locks ??\n",
         num_used_locks);

if (num_used_locks > num_allocated_locks)
{
  num_allocated_locks += LOCK_INCREMENT;
  locks = (SynchQNode **) realloc (locks,
         num_allocated_locks * sizeof (SynchQNode *)�]
  /* initialize newly-allocated portion */
  for (i = lock_id; i < num_allocated_locks; i++)
    locks[i] = NULL;
}
/* provide lock id to user program */
__MEM_WRITE_WORD(lock_ptr, lock_id);
}
break;
case SS_MP_BARRIER:
{
  int barrier_id = regs_R[pid][4];
  /* note that for simplicity, all threads, include the last
   to arrive, are placed in the synch queue for the barrier */
  /* check if at least one item already in queue */
  if (barriers[barrier_id])
    /* add ourselves to the end of the queue for this barrier
head*/
    synchq[pid].next = barriers[barrier_id]->next;/* point to
  */
    barriers[barrier_id]->next = &synchq[pid]; /* add to tail
  */
    barriers[barrier_id] = &synchq[pid]; /* update pointer to
tail*/
  /* increment number of arrivals at barrier */
  ++barrier_counts[barrier_id];
} else
{
  /* first to arrive at this barrier */
  synchq[pid].next = &synchq[pid]; /* tail points to head
tail */
  barriers[barrier_id] = &synchq[pid]; /* update pointer to
  */
  barrier_counts[barrier_id] = 1; /* one arrival so far
/* set synch state and var */
synchq[pid].synch_state = BLOCKED_ON_BARRIER;
synchq[pid].synch_var = barrier_id;

/* mark as inactive until we know that all have arrived */
active[pid] = 0;
ddebug ("pid %d arrived at barrier %d", pid, barrier_id);

/* now check if all have arrived at barrier */
if (regs_R[pid][5] == barrier_counts[barrier_id]) {
    SynchQNode *p;

    /* traverse the queue and mark _all_ threads as active */
    /* (including the last arrival) */
    p = barriers[barrier_id]->next; /* get head of list */
    do {
        active[p->pid] = 1;
        /* set synch state and var */
        synchq[p->pid].synch_state = FREE;
        debug ("pid %d leaving barrier %d", p->pid, barrier_id);
        p = p->next;
    } while (p != barriers[barrier_id]->next); /* back at head? */

    /* reset count to zero */
    barrier_counts[barrier_id] = 0;

    /* last step is to set queue tail pointer to null */
    barriers[barrier_id] = NULL;
}
break;

case SS_MP_INIT_BARRIER:
{
    SS_ADDR_TYPE barrier_ptr = regs_R[pid][4];
    int i;
    int barrier_id = num_used_barriers++;

    if (num_used_barriers > SANITY_LIMIT)
        fatal ("do you _really_ need %d barriers ??\n", num_used_barriers);

    if (num_used_barriers > num_allocated_barriers)
        {
            num_allocated_barriers += BARRIER_INCREMENT;
            barriers = (SynchQNode **) realloc (barriers,
                num_allocated_barriers * sizeof (SynchQNode *)
            );
            barrier_counts = (int *) realloc (barrier_counts,
                num_allocated_barriers * sizeof (int));
        }
/* initialize newly-allocated portions */
for (i = barrier_id; i < num_allocated_barriers; i++)
{
    barriers[i] = NULL;
    barrier_counts[i] = 0;
}

/* provide barrier id to user program */
__MEM_WRITE_WORD(barrier_ptr, barrier_id);
break;

case SS_MP_SEMA_WAIT:
{
    int sema_id = regs_R[pid][4];
    /* if we decrement the count and it is negative, we must
    block */
    if (--sema_counts[sema_id] < 0)
    {
        /* mark as inactive until matching signal unblocks it */
        active[pid] = 0;
        debug ("pid %d blocked on semaphore %d", pid, sema_id);

        /* set synch state and var */
        synchq[pid].synch_state = BLOCKED_ON_SEMAPHORE;
        synchq[pid].synch_var = sema_id;

        /* check if at least one item in queue, which means
        at least one thread is already blocked on semaphore */
        if (semaphores[sema_id])
        {
            /* add ourselves to end of queue for this semaphore */
            synchq[pid].next = semaphores[sema_id]->next;
            semaphores[sema_id]->next = &synchq[pid];
            semaphores[sema_id] = &synchq[pid];
        }
        else
        {
            /* first to block; put ourselves at head of queue */
            synchq[pid].next = &synchq[pid];
            semaphores[sema_id] = &synchq[pid];
        }
    }
    else
    {  
        debug ("pid %d did not block on semaphore %d", pid, sema_id);
    }

break;

case SS_MP_SEMA_SIGNAL:
{
    int sema_id = regs_R[pid][4];
/* increment count; if result is zero or negative, 
there is at least one thread blocked, so unblock it */
if (++sema_counts[sema_id] <= 0)
{
  /* sanity check: there must be at least one blocked 
thread 
   if the count is <= 0 */
  if (semaphores[sema_id] == NULL)
  {
    fatal ("semaphore signal expected a blocked 
thread");
  }
  /* make thread at head of queue active */
  active[semaphores[sema_id]->next->pid] = 1;
  /* set synch state of newly-reactivated thread */
  semaphores[sema_id]->next->synch_state = FREE;
  debug ("pid %d unblocked on semaphore %d by pid %d",
        semaphores[sema_id]->next->pid, sema_id, pid);
  /* check if queue has only one item */
  if (semaphores[sema_id]->next == semaphores[sema_id])
   { /* queue is now empty */
    semaphores[sema_id] = NULL;
  }
  else /* at least one other thread was blocked */
  {
    /* advance head of queue */
    semaphores[sema_id]->next = 
        semaphores[sema_id]->next->next;
  }/*
}
break;

case SS_MP_INIT_SEMA:
{
  SS_ADDR_TYPE sema_ptr = regs_R[pid][4];
  int i;
  int sema_id = num_used_semaphores++;
  if (num_used_semaphores > SANITY_LIMIT)
    fatal ("do you _really_ need %d semaphores ??\n",
           num_used_semaphores);
  if (num_used_semaphores > num_allocated_semaphores)
  {
    num_allocated_semaphores += SEMAPHORE_INCREMENT;
    semaphores = (SynchQNode **) 
      realloc (semaphores,
               num_allocated_semaphores*sizeof(SynchQNode
               *));
    sema_counts = (int *)
      realloc (sema_counts,
               num_allocated_semaphores * sizeof (int));
    /* initialize newly-allocated portions */
for (i = sema_id; i < num_allocated_semaphores; i++)
{
    semaphores[i] = NULL;
    sema_counts[i] = 0;
}

/* set initial semaphore count value */
sema_counts[sema_id] = regs_R[pid][5];
/* provide semaphore id to user program */
MEM_WRITE_WORD(sema_ptr, sema_id);

break;

case SS_MP_THREAD_ID:
    regs_R[pid][7] = 0;
    regs_R[pid][2] = pid;
    break;

case SS_MP_CREATE_THREAD:
{
    CreateNewProcess (pid,
        /* func_ptr */ (void (*)(void)) regs_R[pid][4],
        /* wrapper */ (void (*)(void))
            regs_R[pid][5]);
    regs_R[pid][7] = 0;
    regs_R[pid][2] = 0;
}
    break;

case SS_MP_EXIT_THREAD:
{
    if (pid == 0)
        panic("ERROR: main thread should not call exit_thread()\n");
    ++num_terminated_processes;
    active[pid] = 0; /* mark this one as inactive */
    if (num_terminated_processes == num_created_processes)
    {
        /* exit jumps to the target set in main() */
        longjmp(sim_exit_buf, /* exitcode + fudge */
            regs_R[pid][4]+1);
    }
    /* else we must wait until last thread finishes
       (this one has been marked inactive, so when we return
       to the main simulation loop, no further instructions
       will be fetched for this thread) */
}
    break;

default:
    panic("invalid/unimplemented system call encountered,
        code %d", syscode);
}
/* This doesn't look like -*- C -*-, but it is!
* ss.def - simplescalar machine definition
* This file is an adaptation of the software in the SimpleScalar
* tool suite
* originally written by Todd M. Austin for the Multiscalar Research
* Project
* at the University of Wisconsin-Madison.
* The modifications were made by Naraig Manjikian at Queen's
* University,
* Kingston, Ontario, Canada.
* The remainder of this header comment is unchanged from the
* original text.
*
* .................................................................
* ....
* Copyright (C) 1994, 1995, 1996, 1997 by Todd M. Austin
* This source file is distributed "as is" in the hope that it will be
* useful. The tool set comes with no warranty, and no author or
distributor accepts any responsibility for the consequences of its
* use.
* Everyone is granted permission to copy, modify and redistribute
this tool set under the following conditions:
* This source code is distributed for non-commercial use only.
* Please contact the maintainer for restrictions applying to
* commercial use.
* Permission is granted to anyone to make or distribute copies
of this source code, either as received or modified, in any
medium, provided that all copyright notices, permission and
nonwarranty notices are preserved, and that the distributor
grants the recipient permission for further redistribution as
permitted by this document.
* Permission is granted to distribute this file in compiled
or executable form under the same conditions that apply for
source code, provided that either:
* A. it is accompanied by the corresponding machine-readable
source code,
* B. it is accompanied by a written offer, with no time limit,
to give anyone a machine-readable copy of the corresponding
source code in return for reimbursement of the cost of
distribution. This written offer must permit verbatim
duplication by anyone, or
* C. it is distributed by someone who received only the
executable form, and is accompanied by a copy of the
written offer of source code that they received concurrently.
* In other words, you are welcome to use, share and improve this

153
This file defines all aspects of the SimpleScalar instruction set architecture. Each instruction set in the architecture has a DEFINST() macro call included below. The contents of a instruction definition are as follows:

```
DEFINST(<enum>, <opcode>,
   <opname>, <operands>,
   <fu_req>, <iflags>,
   <output deps...>, <input deps...>,
   <expr>)
```

Where:

- `<enum>` - is an enumerator that is returned when an instruction is decoded by SS_OP_ENUM()
- `<opcode>` - is the opcode of this instruction
- `<opname>` - name of this instruction as a string, used by disassembler
- `<operands>` - specified the instruction operand fields and their printed order for disassembly, used by disassembler, the recognized operand field are (the instruction format is detailed in the header file `ss.h`):
  - J - target field
\* j - PC relative target (offset + PC)
\* s - S register field
\* b - S register field (base register)
\* t - T register field
\* d - D register field
\* S - S register field (FP register)
\* T - T register field (FP register)
\* D - D register field (FP register)
\* o - load address offset (offset)
\* i - signed immediate field value
\* u - unsigned immediate field value
\* U - upper immediate field value
\* H - shift amount immediate field value
\* B - break code
\* <fu_req> - functional unit requirements for this instruction
\* <iflags> - instruction flags, accessible via the SS_OP_FLAGS() macro, flags are defined with F_* prefix in ss.h
\* <output deps...>
\* - a list of up to two output dependency designators, the following designators are recognized (place an DNA in any unused fields:
\* DGPR(N) - general purpose register N
\* DGPR_D(N) - double word general purpose register N
\* DCGPR(N) - general purpose register conditional on pre/post- increment/decrement mode
\* DFPR_L(N) - floating-point register N, as word
\* DFPR_F(N) - floating-point reg N, as single-prec float
\* DFPR_D(N) - floating-point reg N, as double-prec double
\* DHI - HI result register
\* DLO - LO result register
\* DFCC - floating point condition codes
\* DCPC - current PC
\* DNPC - next PC
\* DNA - no dependence
\* <input deps...>
\* - a list of up to three input dependency designators, the designators are defined above (place an DNA in any unused fields.
\* <expr> - a C expression that implements the instruction being defined, the expression must modify all architected state affected by the instruction's execution, by default, the next PC (NPC) value defaults to the current PC (CPC) plus set SS_INST_SIZE, as a result, only taken branches need to in
\*
\* The following predefined macros are available for use
\*
* DEFINST() instruction expressions to access the value
* of
*  * RS     - RS register field value
*  * RT     - RT register field value
*  * RD     - RD register field value
*  * FS     - RS register field value
*  * FT     - RT register field value
*  * FD     - RD register field value
*  * BS     - RS register field value
*  * TARG   - jump target field value
*  * OFS    - signed offset field value
*  * IMM    - signed offset field value
*  * UIMM   - unsigned offset field value
*  * SHAMT  - shift amount field value
*  * BCODE  - break code field value
*
* To facilitate the construction of performance
* simulators
* (which may want to specialize their architected state
* storage format), all architected register and memory
* state is accessed through the following macros:
*
*  * GPR(N)         - read general purpose register N
*  * SET_GPR(N,E)   - write general purpose register N
*  * GPR_D(N)       - read double word general purpose reg
*  * SET_GPR_D(N,E) - write double word gen purpose reg N
*  * FPR_L(N)       - read floating-point register N, as
*  * SET_FPR_L(N,E) - floating-point reg N, as word, with
*  * FPR_F(N)       - read FP reg N, as single-prec float
*  * SET_FPR_F(N,E) - write FP reg N, as single-prec float
*  * FPR_D(N)       - read FP reg N, as double-prec double
*  * SET_FPR_D(N,E) - write FP reg N, as double-prec double
*  * HI            - read HI result register
*  * SET_HI(E)      - write HI result register with E
*  * LO            - read LO result register
*  * SET_LO(E)      - write LO result register with E
*  * FCC           - read floating point condition
*  * SET_FCC(E)     - write floating point condition codes
*  * CPC           - read current PC register
*  * NPC           - read next PC register
*  * SET_NPC(E)    - write next PC register with E
*  * TPC           - read target PC register
*  * SET_TPC(E)    - write target PC register with E
*  * READ_SIGNED_BYTE(A)  - read signed byte from address
*  * READ_UNSIGNED_BYTE(A) - read unsigned byte from
* READ_SIGNED_HALF(A)  - read signed half from address A
* READ_UNSIGNED_HALF(A)  - read unsigned half from address A
* READ_WORD(A)  - read word from address A
* WRITE_BYTE(E,A)  - write byte value E to address A
* WRITE_HALF(E,A)  - write half value E to address A
* WRITE_WORD(E,A)  - write word value E to address A

Finally, the following helper functions are available to assist in the construction of instruction expressions:

* INC_DEC(E,N,S) - execute E and update N as per pre/post-incr/decr addressing sementics for an access of S bytes
* OVER(X,Y) - check for overflow for X+Y, both signed
* UNDER(X,Y) - check for underflow for X-Y, both signed
* DIV0(N) - check for divide by zero, N is denom
* INTALIGN(N) - check double word int reg N alignment
* FPALIGN(N) - check double word FP reg N alignment
* TALIGN(T) - check jump target T alignment

/* no operation */
DEFINST(NOP, 0x00,
   "nop", "==",
   IntALU, F_ICOMP,
   DNA, DNA, DNA, DNA, DNA,
   /* do nothing */((void) 0))

/* control operations */

DEFINST(JUMP, 0x01,
   "j", "J",
   NA, F_CTRL|F_UNCOND|F_DIRJMP,
   DNA, DNA, DNA, DNA, DNA,
   (SET_TPC((CPC & 036000000000) | (TARG << 2)),
    SET_NPC((CPC & 036000000000) | (TARG << 2))))

DEFINST(JAL, 0x02,
   "jal", "J",
   IntALU, F_CTRL|F_UNCOND|F_DIRJMP|F_CALL,
   DGPR(31), DNA, DNA, DNA, DNA,
   (SET_TPC((CPC & 036000000000) | (TARG << 2)),
    SET_NPC((CPC & 036000000000) | (TARG << 2)),
    SET_GPR(31, CPC + 8)))

DEFINST(JR, 0x03,
   "jr", "s",
   NA, F_CTRL|F_UNCOND|F_INDIRJMP,
   DNA, DNA, DGPR(RS), DNA, DNA,
(TALIGN(GPR(RS)), SET_TPC(GPR(RS)), SET_NPC(GPR(RS))))
DEFINST(JALR, 0x04,
    "jalr", "d,s",
    IntALU, F_CTRL|F_UNCOND|F_INDIRJMP|F_CALL,
    DGPR(RD), DNA, DGPR(RS), DNA, DNA,
    (TALIGN(GPR(RS)), SET_GPR(RD, CPC + 8),
    SET_TPC(GPR(RS)), SET_NPC(GPR(RS))))
DEFINST(BEQ, 0x05,
    "beq", "s,t,j",
    IntALU, F_CTRL|F_COND|F_DIRJMP,
    DNA, DNA, DGPR(RS), DGPR(RT), DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    (GPR(RS) == GPR(RT)) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))
DEFINST(BNE, 0x06,
    "bne", "s,t,j",
    IntALU, F_CTRL|F_COND|F_DIRJMP,
    DNA, DNA, DGPR(RS), DGPR(RT), DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    (GPR(RS) != GPR(RT)) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))
DEFINST(BLEZ, 0x07,
    "blez", "s,j",
    IntALU, F_CTRL|F_COND|F_DIRJMP,
    DNA, DNA, DGPR(RS), DNA, DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    (GPR(RS) <= 0) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))
DEFINST(BGEZ, 0x08,
    "bgez", "s,j",
    IntALU, F_CTRL|F_COND|F_DIRJMP,
    DNA, DNA, DGPR(RS), DNA, DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    (GPR(RS) >= 0) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))
DEFINST(BC1F, 0x0b,
    "bc1f", "j",
    IntALU, F_CTRL|F_COND|F_DIRJMP|F_FPCOND,
    DNA, DNA, DGPR(RS), DNA, DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    ((!FCC) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))
DEFINST(BC1T, 0x0c,
    "bc1t", "j",
    IntALU, F_CTRL|F_COND|F_DIRJMP|F_FPCOND,
    DNA, DNA, DGPR(RS), DNA, DNA,
    (SET_TPC(CPC + 8 + (OFS << 2)),
    (FCC) ? SET_NPC(CPC + 8 + (OFS << 2)) : 0))

/*
 * load/store operations
 */
* NOTE: the out-of-order issue simulator(s) require that load and store
* address computation input dependencies be placed in slots 1 and 2 of
* the input dependency list slot 0 is reserved for the input dependency
* of store values for store instructions
*/

DEFINST(LB, 0x20,
"lb", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_gpr(RT, read_signed_byte(GPR(BS)+OFS)), BS, 1))

DEFINST(LBU, 0x22,
"lbu", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_gpr(RT, read_unsigned_byte(GPR(BS)+OFS)), BS, 1))

DEFINST(LH, 0x24,
"lh", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_gpr(RT, read_signed_half(GPR(BS)+OFS)), BS, 2))

DEFINST(LHU, 0x26,
"lhu", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_gpr(RT, read_unsigned_half(GPR(BS)+OFS)), BS, 2))

DEFINST(LW, 0x28,
"lw", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_gpr(RT, read_word(GPR(BS)+OFS)), BS, 4))

DEFINST(DLW, 0x29,
"dlw", "t,o(b)",
RdPort_D(RT), F_MEM|F_LOAD|F_DISP,
DGPR_D(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC((intalign(RT), temp_bs = GPR(BS),
set_gpr((RT)^sim_swap_words, read_word(temp_bs+OFS)),
set_gpr(((RT)+1)^sim_swap_words,
read_word(temp_bs+OFS+4)),
BS, 8)))

DEFINST(L_S, 0x2a,
"l.s", "T,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DFPR_L(FT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC(set_fpr_L(FT, read_word(GPR(BS)+OFS)), BS, 4))

DEFINST(L_D, 0x2b,
"l.d", "T,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DFPR_D(FT), DCGPR(BS), DNA, DGPR(BS), DNA,
INC_DEC((fpalign(FT),
set_fpr_L((FT)^sim_swap_words, read_word(GPR(BS)+OFS)),
set_fpr_L(((FT)+1)^sim_swap_words,
read_word(GPR(BS)+OFS+4)),
BS, 8)))

DEFINST(LWL, 0x2c,
"lw1", "t,o(b)",
RdPort, F_MEM|F_LOAD|F_DISP,
DGPR(RT), DNA, DNA, DGPR(BS), DNA,
   (/* inc/dec not supported */
   /* BS may == RT */temp_bs = GPR(BS),
   ss_lr_temp = READ_WORD(WL_BASE(temp_bs+OFS)),
   SET_GPR(RT, ((GPR(RT) & WL_PROT_MASK(temp_bs+OFS)) |
                     (ss_lr_temp & ~WL_PROT_MASK(temp_bs+OFS))))))
DEFINST(LWR, 0x2d,
   "lwr", "t,o(b)",
   RdPort, F_MEM|F_LOAD|F_DISP,
   DGPR(RT), DNA, DNA, DGPR(BS), DNA,
   (/* inc/dec not supported */
   /* BS may == RT */temp_bs = GPR(BS),
   ss_lr_temp = READ_WORD(WR_BASE(temp_bs+OFS)),
   SET_GPR(RT, ((GPR(RT) & WR_PROT_MASK(temp_bs+OFS)) |
                     (ss_lr_temp & ~WR_PROT_MASK(temp_bs+OFS))))))
DEFINST(SB, 0x30,
   "sb", "t,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DNA,
   INC_DEC(WRITE_BYTE(GPR(RT), GPR(BS)+OFS), BS, 1))
DEFINST(SH, 0x32,
   "sh", "t,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DNA,
   INC_DEC(WRITE_HALF(GPR(RT), GPR(BS)+OFS), BS, 2))
DEFINST(SW, 0x34,
   "sw", "t,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DNA,
   INC_DEC(WRITE_WORD(GPR(RT), GPR(BS)+OFS), BS, 4))
DEFINST(DSW, 0x35,
   "dsw", "t,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DGPR_D(RT), DGPR(BS), DNA,
   INC_DEC((INTALIGN(RT),
                     WRITE_WORD(GPR((RT)^sim_swap_words), GPR(BS)+OFS),
                     WRITE_WORD(GPR(((RT)+1)^sim_swap_words), GPR(BS)+OFS+4)),
                     BS, 8))
DEFINST(DSZ, 0x38,
   "dsz", "o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DNA, DGPR(BS), DNA,
   INC_DEC((WRITE_WORD(GPR(0), GPR(BS)+OFS),
                     WRITE_WORD(GPR(0), GPR(BS)+OFS+4)),
                     BS, 8))
DEFINST(S_S, 0x36,
   "s.s", "T,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DFPR_L(FT), DGPR(BS), DNA,
   INC_DEC(WRITE Word(FPR_L(FT), GPR(BS)+OFS), BS, 4))
DEFINST(S_D, 0x37,
   "s.d", "T,o(b)",
   WrPort, F_MEM|F_STORE|F_DISP,
   DCGPR(BS), DNA, DFPR_D(FT), DGPR(BS), DNA,
   INC_DEC((FPALIGN(FT),
                     WRITE_WORD(FPR_L(L(FT)), GPR(BS)+OFS),
                     WRITE_WORD(FPR_L((FT)+1), GPR(BS)+OFS+4)),
                     BS, 4))
DEFINST(SWL, 0x39,
"swl",
WrPort, F_MEM|F_STORE|F_DISP,
DNA, DNA, DGPR(RT), DGPR(BS), DNA,
/* inc/dec not supported */
ss_lr_temp = READ_WORD(WL_BASE(GPR(BS)+OFS)),
ss_lr_temp = ((GPR(RT) & ~WL_PROT_MASK(GPR(BS)+OFS)) |
(ss_lr_temp & WL_PROT_MASK(GPR(BS)+OFS))),
WRITE_WORD(ss_lr_temp, WL_BASE(GPR(BS)+OFS)))
DEFINST(SWR,
  0x3a,
"swr",
WrPort, F_MEM|F_STORE|F_DISP,
DNA, DNA, DGPR(RT), DGPR(BS), DNA,
/* inc/dec not supported */
ss_lr_temp = READ_WORD(WR_BASE(GPR(BS)+OFS)),
ss_lr_temp = ((GPR(RT) & ~WR_PROT_MASK(GPR(BS)+OFS)) |
(ss_lr_temp & WR_PROT_MASK(GPR(BS)+OFS))),
WRITE_WORD(ss_lr_temp, WR_BASE(GPR(BS)+OFS)))
/* reg+reg loads and stores */
DEFINST(LB_RR, 0xc0,
  "lb",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_GPR(RT, READ_SIGNED_BYTE(GPR(BS)+GPR(RD))), BS, 1))
DEFINST(LBU_RR, 0xc1,
  "lbu",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_GPR(RT, READ_UNSIGNED_BYTE(GPR(BS)+GPR(RD))), BS, 1))
DEFINST(LH_RR, 0xc2,
  "lh",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_GPR(RT, READ_SIGNED_HALF(GPR(BS)+GPR(RD))), BS, 2))
DEFINST(LHU_RR, 0xc3,
  "lhu",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_GPR(RT, READ_UNSIGNED_HALF(GPR(BS)+GPR(RD))), BS, 2))
DEFINST(LW_RR, 0xc4,
  "lw",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_GPR(RT, READ_WORD(GPR(BS)+GPR(RD))), BS, 4))
DEFINST(DLW_RR, 0xce,
  "dlw",
RdPort, F_MEM|F_LOAD|F_RR,
DGPR_D(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC((INTALIGN(RT),
temp_bs = GPR(BS), temp_rd = GPR(RD),
SET_GPR(RT)^sim_swap_words, READ_WORD(temp_bs+temp_rd)),
SET_GPR(((RT)+1)^sim_swap_words,
READ_WORD(temp Bs+temp_rd+4))),
BS, 8))
DEFINST(L_S_RR, 0xc5,
  "l.s",
RdPort, F_MEM|F_LOAD|F_RR,
DFPR_L(FT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC(SET_FPR_L(FT, READ_WORD(GPR(BS)+GPR(RD))), BS, 4)) /* was
INC_DEC(SET_FPR_L(RT, READ_WORD(GPR(BS)+GPR(RD))), BS, 4)) */
DEFINST(L_D_RR, 0xcf,
"l.d", "T,(b+d)",
RdPort, F_MEM|F_LOAD|F_RR,
DFPR_D(FT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
INC_DEC((FPALIGN(FT),
SET_FPR_L((FT)^sim_swap_words,
READ_WORD(GPR(BS)+GPR(RD))),
SET_FPR_L(((FT)+1)^sim_swap_words,
READ_WORD(GPR(BS)+GPR(RD)+4))),
BS, 8))
DEFINST(SB_RR, 0xc6,
"sb", "t,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DGPR(RD),
INC_DEC(WRITE_BYTE(GPR(RT), GPR(BS)+GPR(RD)), BS, 1))
DEFINST(SH_RR, 0xc7,
"sh", "t,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DGPR(RD),
INC_DEC(WRITE_HALF(GPR(RT), GPR(BS)+GPR(RD)), BS, 2))
DEFINST(SW_RR, 0xc8,
"sw", "t,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DGPR(RD),
INC_DEC(WRITE_WORD(GPR(RT), GPR(BS)+GPR(RD)), BS, 4))
DEFINST(DSW_RR, 0xd0,
"dsw", "t,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DGPR_D(RT), DGPR(BS), DGPR(RD),
INC_DEC((INTALIGN(RT),
WRITE_WORD(GPR((RT)^sim_swap_words), GPR(BS)+GPR(RD)),
WRITE_WORD(GPR(((RT)+1)^sim_swap_words),
GPR(BS)+GPR(RD)+4)),
BS, 8))
DEFINST(DSZ_RR, 0xd1,
"dsz", "(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DNA, DGPR(BS), DGPR(RD),
INC_DEC((WRITE_WORD(GPR(0), GPR(BS)+GPR(RD)),
WRITE_WORD(GPR(0), GPR(BS)+GPR(RD)+4)),
BS, 8))
DEFINST(S_S_RR, 0xc9,
"s.s", "T,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DFPR_L(FT), DGPR(BS), DGPR(RD),
INC_DEC(WRITE_WORD(FPR_L(FT), GPR(BS)+GPR(RD)), BS, 4))
DEFINST(S_D_RR, 0xd2,
"s.d", "T,(b+d)",
WrPort, F_MEM|F_STORE|F_RR,
DCGPR(BS), DNA, DFPR_D(FT), DGPR(BS), DGPR(RD),
INC_DEC((FPALIGN(FT),
WRITE_WORD(FPR_L((FT)^sim_swap_words), GPR(BS)+GPR(RD)),
WRITE_WORD(FPR_L(((FT)+1)^sim_swap_words),
GPR(BS)+GPR(RD)+4)),
BS, 8))
/* reg + reg + 4 addressing mode, used to synthesize `l.d r,(s+t)^++'
 */
/* FIXME: obsolete */
DEFINST(l_S_RR_R2, 0xca,  "l.s.r2", "T,(b+d)",
        RdPort, F_MEM|F_LOAD|F_RR,
        DFPR_L(FT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
        INC_DEC(SET_FPR_L(FT, READ_WORD(GPR(BS)+GPR(RD)+4)), BS, 4))
/* was  INC_DEC(SET_FPR_L(RT, READ_WORD(GPR(BS)+GPR(RD)+4)), BS, 4)) */
/* FIXME: obsolete */
DEFINST(S_S_RR_R2, 0xcb,  "s.s.r2", "T,(b+d)",
        WrPort, F_MEM|F_STORE|F_RR,
        DCGPR(BS), DNA, DFPR_L(FT), DGPR(BS), DGPR(RD),
        INC_DEC(WRITE_WORD(FPR_L(FT), GPR(BS)+GPR(RD)+4), BS, 4))
/* FIXME: obsolete */
DEFINST(LW_RR_R2, 0xcc,  "lw.r2", "t,(b+d)",
        RdPort, F_MEM|F_LOAD|F_RR,
        DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DGPR(RD),
        INC_DEC(SET_GPR(RT, READ_WORD(GPR(BS)+GPR(RD)+4)), BS, 4))
/* FIXME: obsolete */
DEFINST(SW_RR_R2, 0xcd,  "sw.r2", "t,(b+d)",
        WrPort, F_MEM|F_STORE|F_RR,
        DCGPR(BS), DNA, DGPR(RT), DGPR(BS), DGPR(RD),
        INC_DEC(WRITE_WORD(GPR(RT), GPR(BS)+GPR(RD)+4), BS, 4))

/* Integer ALU operations */
DEFINST(ADD, 0x40,  "add", "d,s,t",
        IntALU, F_ICOMP,
        DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
        (OVER(GPR(RS),GPR(RT)), SET_GPR(RD, GPR(RS) + GPR(RT))))
DEFINST(ADDI, 0x41,  "addi", "t,s,i",
        IntALU, F_ICOMP,
        DGPR(RT), DNA, DGPR(RS), DNA, DNA,
        (OVER(GPR(RS),IMM), SET_GPR(RT, GPR(RS) + IMM)))
DEFINST(ADDU, 0x42,  "addu", "d,s,t",
        IntALU, F_ICOMP,
        DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
        SET_GPR(RD, GPR(RS) + GPR(RT)))
DEFINST(ADDIU, 0x43,  "addiu", "t,s,i",
        IntALU, F_ICOMP,
        DGPR(RT), DNA, DGPR(RS), DNA, DNA,
        SET_GPR(RT, GPR(RS) + IMM))
DEFINST(SUB, 0x44,  "sub", "d,s,t",
        IntALU, F_ICOMP,
DGPR(RD), DNA,  
DGPR(RS), DGPR(RT), DNA,  
(UNDER(GPR(RS),GPR(RT)), SET_GPR(RD, GPR(RS) - GPR(RT))))
DEFINST(SUBU,  
0x45,  
"subu",  
"d,s,t",  
IntALU,  
F_ICOMP,  
DGPR(RD), DNA,  
DGPR(RS), DGPR(RT), DNA,  
SET_GPR(RD, GPR(RS) - GPR(RT)))
DEFINST(MULT,  
0x46,  
"mult",  
"s,t",  
IntMULT,  
F_ICOMP|F_LONGLAT,  
DHI, DLO,  
DGPR(RT), DGPR(RS), DNA,  
InstMULT(inst, pid))
DEFINST(MULTU,  
0x47,  
"multu",  
"s,t",  
IntMULT,  
F_ICOMP|F_LONGLAT,  
DHI, DLO,  
DGPR(RT), DGPR(RS), DNA,  
InstMULTU(inst, pid))
DEFINST(DIV,  
0x48,  
"div",  
"s,t",  
IntDIV,  
F_ICOMP|F_LONGLAT,  
DHI, DLO,  
DGPR(RT), DGPR(RS), DNA,  
(DIVO(GPR(RT)),  
SET_LO(IDIV(GPR(RS), GPR(RT))), SET_HI(IMOD(GPR(RS), GPR(RT))))
DEFINST(DIVU,  
0x49,  
"divu",  
"s,t",  
IntDIV,  
F_ICOMP|F_LONGLAT,  
DHI, DLO,  
DGPR(RT), DGPR(RS), DNA,  
(DIVO(GPR(RT)),  
SET_LO(IDIV(((unsigned)GPR(RS)), ((unsigned)GPR(RT)))),  
SET_HI(IMOD(((unsigned)GPR(RS)), ((unsigned)GPR(RT))))))
DEFINST(MFHI,  
0x4a,  
"mfhi",  
"d",  
IntALU,  
F_ICOMP,  
DGPR(RD), DNA,  
DHI, DNA, DNA,  
SET_GPR(RD, HI))
DEFINST(MTHI,  
0x4b,  
"mthi",  
"s",  
IntALU,  
F_ICOMP,  
DHI, DNA,  
DGPR(RS), DNA, DNA,  
SET_HI(GPR(RS)))
DEFINST(MFLO,  
0x4c,  
"mflo",  
"d",  
IntALU,  
F_ICOMP,  
DGPR(RD), DNA,  
DLO, DNA, DNA,  
SET_GPR(RD, LO))
DEFINST(MTLO,  
0x4d,  
"mtlo",  
"s",  
IntALU,  
F_ICOMP,  
DLO, DNA,  
DGPR(RS), DNA, DNA,  
SET_LO(GPR(RS)))
/* AND conflicts with GNU defs */
DEFINST(AND_,  
0x4e,  
"and",  
"d,s,t",  
IntALU,  
F_ICOMP,  
DGPR(RD), DNA,  
DGPR(RS), DGPR(RT), DNA,  
SET_GPR(RD, GPR(RS) & GPR(RT)))
DEFINST(ANDI, 0x4f,
     "andi", "t,s,u",
     IntALU, F_ICOMP,
     DGPR(RT), DNA, DGPR(RS), DNA, DNA,
     SET_GPR(RT, GPR(RS) & UIMM))

DEFINST(OR, 0x50,
     "or", "d,s,t",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
     SET_GPR(RD, GPR(RS) | GPR(RT)))

DEFINST(ORI, 0x51,
     "ori", "t,s,u",
     IntALU, F_ICOMP,
     DGPR(RT), DNA, DGPR(RS), DNA, DNA,
     SET_GPR(RT, GPR(RS) | UIMM))

DEFINST(XOR, 0x52,
     "xor", "d,s,t",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
     SET_GPR(RD, GPR(RS) ^ GPR(RT)))

DEFINST(XORI, 0x53,
     "xori", "t,s,u",
     IntALU, F_ICOMP,
     DGPR(RT), DNA, DGPR(RS), DNA, DNA,
     SET_GPR(RT, GPR(RS) ^ UIMM))

DEFINST(NOR, 0x54,
     "nor", "d,s,t",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
     SET_GPR(RD, ~(GPR(RS) | GPR(RT))))

DEFINST(SLL, 0x55,
     "sll", "d,t,H",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RT), DNA, DNA,
     SET_GPR(RD, GPR(RT) << SHAMT))

DEFINST(SLLV, 0x56,
     "sllv", "d,t,s",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
     SET_GPR(RD, GPR(RT) << (GPR(RS) & 037)))

DEFINST(SRL, 0x57,
     "srl", "d,t,H",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RT), DNA, DNA,
     SET_GPR(RD, ((unsigned)GPR(RT)) >> SHAMT))

DEFINST(SRLV, 0x58,
     "srlv", "d,t,s",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA,
     SET_GPR(RD, ((unsigned)GPR(RT)) >> (GPR(RS) & 037)))

DEFINST(SRA, 0x59,
     "sra", "d,t,H",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RT), DNA, DNA,
     InstSRA(inst, pid))

DEFINST(SRAV, 0x5a,
     "sraw", "d,t,s",
     IntALU, F_ICOMP,
     DGPR(RD), DNA, DGPR(RT), DNA, DNA,
\texttt{InstSRAV(inst,pid)}

\begin{verbatim}
DEFINST(SLT, 0x5b, "slt", "d,s,t", IntALU, F_ICOMP, DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA, SET_GPR(RD, (GPR(RS) < GPR(RT)) ? 1 : 0))
DEFINST(SLTI, 0x5c, "slti", "t,s,i", IntALU, F_ICOMP, DGPR(RT), DNA, DGPR(RS), DNA, SET_GPR(RT, (GPR(RS) < IMM) ? 1 : 0))
DEFINST(SLTU, 0x5d, "sltu", "d,s,t", IntALU, F_ICOMP, DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA, SET_GPR(RD, ((unsigned)GPR(RS)) < ((unsigned)GPR(RT))) ? 1 : 0))
DEFINST(SLTIU, 0x5e, "sltiu", "t,s,i", IntALU, F_ICOMP, DGPR(RT), DNA, DGPR(RS), DNA, DNA, SET_GPR(RT, ((unsigned)GPR(RS) < (unsigned)IMM)) ? 1 : 0))

/\* Floating Point ALU operations */
DEFINST(FADD_S, 0x70, "add.s", "D,S,T", FloatADD, F_FCOMP, DFPR_F(FD), DNA, DFPR_F(FS), DFPR_F(FT), DNA, /* FIXME: check precedences here */ (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT), SET_FPR_F(FD, FPR_F(FS) + FPR_F(FT))))
DEFINST(FADD_D, 0x71, "add.d", "D,S,T", FloatADD, F_FCOMP, DFPR_D(FD), DNA, DFPR_D(FS), DFPR_D(FT), DNA, /* FIXME: check precedences here */ (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT), SET_FPR_D(FD, FPR_D(FS) + FPR_D(FT))))
DEFINST(FSUB_S, 0x72, "sub.s", "D,S,T", FloatADD, F_FCOMP, DFPR_F(FD), DNA, DFPR_F(FS), DFPR_F(FT), DNA, (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT), SET_FPR_F(FD, FPR_F(FS) - FPR_F(FT))))
DEFINST(FSUB_D, 0x73, "sub.d", "D,S,T", FloatADD, F_FCOMP, DFPR_D(FD), DNA, DFPR_D(FS), DFPR_D(FT), DNA, (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT), SET_FPR_D(FD, FPR_D(FS) - FPR_D(FT))))
DEFINST(FMUL_S, 0x74, "mul.s", "D,S,T", FloatMULT, F_FCOMP|F_LONGLAT, DFPR_F(FD), DNA, DFPR_F(FS), DFPR_F(FT), DNA, (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT), TPALIGN(FT)))
\end{verbatim}
SET_FPR_F(FD, FPR_F(FS) * FPR_F(FT)))
DEFINST(FMUL_D, 0x75,
      "mul.d",    "D,S,T",
      FloatMULT,  F_FCOMP|F_LONGLAT,
      DFPR_D(FD), DNA, DFPR_D(FS), DFPR_D(FT), DNA,
      (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT),
       SET_FPR_D(FD, FPR_D(FS) * FPR_D(FT))))
DEFINST(FDIV_S, 0x76,
      "div.s",    "D,S,T",
      FloatDIV,   F_FCOMP|F_LONGLAT,
      DFPR_F(FD), DNA, DFPR_F(FS), DFPR_F(FT), DNA,
      (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT),
       (DIV0(FPR_F(FT)), SET_FPR_F(FD, FDIV(FPR_F(FS), FPR_F(FT))))))
DEFINST(FDIV_D, 0x77,
      "div.d",    "D,S,T",
      FloatDIV,   F_FCOMP|F_LONGLAT,
      DFPR_D(FD), DNA, DFPR_D(FS), DFPR_D(FT), DNA,
      (FPALIGN(FD), FPALIGN(FS), FPALIGN(FT),
       (DIV0(FPR_D(FT)), SET_FPR_D(FD, FDIV(FPR_D(FS), FPR_D(FT))))))
DEFINST(FABS_S, 0x78,
      "abs.s",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_F(FD), DNA, DFPR_F(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, 
       fabs((double)FPR_F(FS)))))
DEFINST(FABS_D, 0x79,
      "abs.d",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_D(FD), DNA, DFPR_D(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, fabs(FPR_D(FS))))
DEFINST(FMOV_S, 0x7a,
      "mov.s",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_F(FD), DNA, DFPR_F(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, FPR_F(FS))))
DEFINST(FMOV_D, 0x7b,
      "mov.d",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_D(FD), DNA, DFPR_D(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, FPR_D(FS))))
DEFINST(FNEG_S, 0x7c,
      "neg.s",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_F(FD), DNA, DFPR_F(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, -FPR_F(FS))))
DEFINST(FNEG_D, 0x7d,
      "neg.d",    "D,S",
      FloatADD,   F_FCOMP,
      DFPR_D(FD), DNA, DFPR_D(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, -FPR_D(FS))))
DEFINST(CVT_S_D, 0x80,
      "cvt.s.d",  "D,S",
      FloatCVT,   F_FCOMP,
      DFPR_F(FD), DNA, DFPR_F(FS), DNA, DNA,
      (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, (float)FPR_D(FS))))
DEFINST(CVT_S_W, 0x81,
      "cvt.s.w",  "D,S",
      FloatCVT,   F_FCOMP,
      DFPR_F(FD), DNA, DFPR_L(FS), DNA, DNA,
DEFINST(CVT_D_S, 0x82, "cvt.d.s", "D,S", FloatCVT, F_FCOMP, DFPR_D(FD), DNA, DFPR_F(FS), DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, (float)FPR_F(FS))))
DEFINST(CVT_D_W, 0x83, "cvt.d.w", "D,S", FloatCVT, F_FCOMP, DFPR_D(FD), DNA, DFPR_L(FS), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, (double)FPR_L(FS))))
DEFINST(CVT_W_S, 0x84, "cvt.w.s", "D,S", FloatCVT, F_FCOMP, DFPR_L(FD), DNA, DFPR_F(FS), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_L(FD, FINT(FPR_F(FS))))
DEFINST(CVT_W_D, 0x85, "cvt.w.d", "D,S", FloatCVT, F_FCOMP, DFPR_L(FD), DNA, DFPR_D(FS), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_L(FD, FINT(FPR_D(FS))))
DEFINST(C_EQ_S, 0x90, "c.eq.s", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_F(FS), DFPR_F(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_F(FS) == FPR_F(FT))))
DEFINST(C_EQ_D, 0x91, "c.eq.d", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_D(FS), DFPR_D(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_D(FS) == FPR_D(FT))))
DEFINST(C_LT_S, 0x92, "c.lt.s", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_F(FS), DFPR_F(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_F(FS) < FPR_F(FT))))
DEFINST(C_LT_D, 0x93, "c.lt.d", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_D(FS), DFPR_D(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_D(FS) < FPR_D(FT))))
DEFINST(C_LE_S, 0x94, "c.le.s", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_F(FS), DFPR_F(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_F(FS) <= FPR_F(FT))))
DEFINST(C_LE_D, 0x95, "c.le.d", "S,T", FloatCMP, F_FCOMP, DFCC, DNA, DFPR_D(FS), DFPR_D(FT), DNA, (FPALIGN(FS), FPALIGN(FT), SET_FCC(FPR_D(FS) <= FPR_D(FT))))
DEFINST(FSQRT_S, 0x96, "sqrt.s", "D,S", FloatSQRT, F_FCOMP|F_LONGLAT, DFPR_F(FD), DNA, DFPR_F(FT), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, sqrt((double)FPR_F(FS))))
DEFINST(FSQRT_D, 0x97, "sqrt.d", "D,S", FloatSQRT, F_FCOMP|F_LONGLAT, DFPR_D(FD), DNA, DFPR_D(FT), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, sqrt((double)FPR_D(FS))))
DEFINST(FSQRT_S, 0x96, "sqrt.s", "D,S", FloatSQRT, F_FCOMP|F_LONGLAT, DFPR_F(FD), DNA, DFPR_F(FT), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_F(FD, sqrt((double)FPR_F(FS))))
DEFINST(FSQRT_D, 0x97, "sqrt.d", "D,S", FloatSQRT, F_FCOMP|F_LONGLAT, DFPR_D(FD), DNA, DFPR_D(FT), DNA, DNA, (FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, sqrt((double)FPR_D(FS))))
"sqrt.d", "D,S",
FloatSQRT, F_FCOMP|F_LONGLAT,
DFPR_D(FD), DNA, DFPR_D(FS), DNA, DNA,
(FPALIGN(FD), FPALIGN(FS), SET_FPR_D(FD, sqrt(FPR_D(FS))))

/*
 * miscellaneous
 */

DEFINST(SYSCALL, 0xa0, 
"syscall", 
NA, F_TRAP, DNA, DNA, DNA, DNA, SYSCALL(inst))

DEFINST(BREAK, 0xa1, 
"break", "B",
NA, F_TRAP, DNA, DNA, DNA, DNA, DNA,
/* NOTE: these are decoded speculatively, as they occur in integer
divide sequences, however, they should NEVER be executed under
non-exception conditions */
/* abort() */(void) 0)

DEFINST(LUI, 0xa2, 
"lui", "t,U",
IntALU, F_ICOMP, DGPR(RT), DNA, DNA, DNA, DNA,
SET_GPR(RT, UIMM << 16))

DEFINST(MFC1, 0xa3, 
"mfc1", "t,S",
IntALU, F_ICOMP, DGPR(RT), DNA, DFPR_L(FS), DNA, DNA,
SET_GPR(RT, FPR_L(FS)))

DEFINST(DMFC1, 0xa7, 
"dmfc1", "t,S",
IntALU, F_ICOMP, DGPR_D(RT), DNA, DFPR_D(FS), DNA, DNA,
(INTALIGN(RT), FPALIGN(FS),
SET_GPR(RT, FPR_L(FS)), SET_GPR((RT)+1, FPR_L((FS)+1))))

DEFINST(CFC1, 0xa4, 
"cfc1", "t,S",
IntALU, F_ICOMP, DNA, DNA, DNA, DNA, DNA,
/* FIXME: is this needed */((void) 0))

DEFINST(MTC1, 0xa5, 
"mtc1", "t,S",
IntALU, F_ICOMP, DFPR_L(FS), DNA, DGPR(RT), DNA, DNA,
SET_FPR_L(FS, GPR(RT)))

DEFINST(DMTC1, 0xa8, 
"dmtc1", "t,S",
IntALU, F_ICOMP, DFPR_D(FS), DNA, DGPR_D(RT), DNA, DNA,
(FPALIGN(FS), INTALIGN(RT),
SET_FPR_L(FS, GPR(RT)), SET_FPR_L((FS)+1, GPR((RT)+1))))

DEFINST(CTC1, 0xa6, 
"ctc1", "t,S",

```c
IntALU, F_ICOMP,
DNA, DNA, DNA, DNA, DNA,
/* FIXME: is this needed */((void) 0))

/*---------My LL/SC instruction definition----------------------
----*/
DEFINST(LL,                     0xb0,
"ll",                   "t,o(b)",
RdPort,   F_MEM|F_LOAD|F_DISP,
DGPR(RT), DCGPR(BS), DNA, DGPR(BS), DNA,
InstLL(inst,pid))
/* INC_DEC(SET_GPR(RT, READ_WORD(GPR(BS)+OFS)), BS, 4))*/
DEFINST(SC,                     0xb1,
"sc",                   "t,o(b)",
WrPort,   F_MEM|F_STORE|F_DISP,
DCGPR(BS), DNA,   DGPR(RT), DGPR(BS), DNA,
InstSC(inst,pid))
/* INC_DEC(WRITE_WORD(GPR(RT), GPR(BS)+OFS), BS, 4))*/
#endif IMPL

/*
* non-expression instruction implementations
*/

/*
* rd <- [rt] >> SHAMT
*/
static void
InstSRA(SS_INST_TYPE inst, int pid)
{
  unsigned int i;

  /* Although SRA could be implemented with the >> operator in most
   * machines, there are other machines that perform a logical
   * right shift with the >> operator. */
  if (GPR(RT) & 0200000000000) {
    SET_GPR(RD, GPR(RT));
    for (i = 0; i < SHAMT; i++) {
      SET_GPR(RD, (GPR(RD) >> 1) | 0200000000000);
    }
  }
  else {
    SET_GPR(RD, GPR(RT) >> SHAMT);
  }
}

/*
* rd <- [rt] >> [5 LSBs of rs])
*/
static void
InstSRAV(SS_INST_TYPE inst, int pid)
{
  unsigned int i;
  unsigned int shamt = GPR(RS) & 037;

  if (GPR(RT) & 0200000000000) {
    SET_GPR(RD, GPR(RT));
    for (i = 0; i < shamt; i++) {
      SET_GPR(RD, (GPR(RD) >> 1) | 0200000000000);
    }
  }
```
} else {
    SET_GPR(RD, GPR(RT) >> shamt);
}

/*
* HI,LO <- [rs] * [rt], integer product of [rs] & [rt] to HI/LO
*/
static void
InstMULT(SS_INST_TYPE inst, int pid)
{
    int sign1, sign2;
    int i, op1, op2;

    sign1 = sign2 = 0;
    SET_HI(0);
    SET_LO(0);
    op1 = GPR(RS);
    op2 = GPR(RT);

    /* For multiplication, treat -ve numbers as +ve numbers by
       converting 2's complement -ve numbers to ordinary notation */
    if (op1 & 020000000000) {
        sign1 = 1;
        op1 = (~op1) + 1;
    }
    if (op2 & 020000000000) {
        sign2 = 1;
        op2 = (~op2) + 1;
    }
    if (op1 & 020000000000)
        SET_LO(op2);
    for (i = 0; i < 31; i++) {
        SET_HI(HI << 1);
        SET_HI(HI + extractl(LO, 31, 1));
        SET_LO(LO << 1);
        if ((extractl(op1, 30-i, 1)) == 1) {
            if (((unsigned)037777777777 - (unsigned)LO) < (unsigned)op2) {
                SET_HI(HI + 1);
            }
            SET_LO(LO + op2);
        }
    }

    /* Take 2's complement of the result if the result is negative */
    if (sign1 ^ sign2) {
        SET_LO(~LO);
        SET_HI(~HI);
        if ((unsigned)LO == 037777777777) {
            SET_HI(HI + 1);
        }
        SET_LO(LO + 1);
    }
}

/*
* HI,LO <- [rs] * [rt], integer product of [rs] & [rt] to HI/LO
*/
static void
InstMULTU(SS_INST_TYPE inst, int pid)
{
    int i;

    SET_HI(0);
    SET_LO(0);
    if (GPR(RS) & 020000000000)
        SET_LO(GPR(RT));
    for (i = 0; i < 31; i++) {
        SET_HI(HI << 1);
        SET_HI(HI + extractl(LO, 31, 1));
        SET_LO(LO << 1);
        if ((extractl(GPR(RS), 30-i, 1)) == 1) {
            if (((unsigned)037777777777 - (unsigned)LO) < 
                (unsigned)GPR(RT)) {
                SET_HI(HI + 1);
            }
            SET_LO(LO + GPR(RT));
        }
    }
}

static void InstLL(SS_INST_TYPE inst, int pid)
{
    if(LFR == 0)
    {
        INC_DEC(SET_GPR(RT, READ_WORD(GPR(BS)+OFS)), BS, 4);
        SET_LFR(1);
        SET_LAR(GPR(BS)+OFS);
        printf("\n Lock acquired successfully by processor %d \n", pid);
    }
}

static void InstSC(SS_INST_TYPE inst, int pid)
{
    if((LFR == 1) && (LAR == GPR(BS)+OFS))
    {
        INC_DEC(WRITE_WORD(GPR(RT), GPR(BS)+OFS), BS, 4);
        SET_LFR(0);
        SET_LAR(0);
        SET_GPR(RT,1);
        printf("\n Lock released successfully by processor %d \n", pid);
    }
}

#endif /* IMPL */