The Effects of the architectural design, replacement algorithm, and size parameters of cache memory in uniprocessor computer systems

Eric Berzofsky

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
THE EFFECTS OF THE ARCHITECTURAL
DESIGN, REPLACEMENT ALGORITHM, AND SIZE
PARAMETERS OF CACHE MEMORY IN
UNIPROCESSOR COMPUTER SYSTEMS

by

Eric Benjamin Berzofsky

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Principle Advisor: ________________________________
Roy S. Czernikowski, Professor and Department Head

Date: Nov. 6, 1998

Committee Member: ______________________________
Muhammad E. Shaaban, Assistant Professor

Date: 11-6-98

Committee Member: ______________________________
Kenneth W. Hsu, Professor

Date: 11/6/98

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
November 1998
RELEASE PERMISSION FORM

Rochester Institute of Technology

The Effects of the Architectural Design, Replacement Algorithm, and Size Parameters of Cache Memory in Uniprocessor Computer Systems

I, Eric Benjamin Berzofsky, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

__________________________________________
Eric Benjamin Berzofsky

11-06-98
Date
ABSTRACT

To investigate the effects of cache coherency on multiprocessors, it is helpful to first explore coherency issues within uniprocessors, working with a small part of a big problem instead of attacking the big problem from the start. This thesis will investigate the design and implementation of three different cache designs, varying the mapping strategy, replacement algorithm, and size parameters to determine the effects each have on the cache miss ratio, coherency, and average memory access time.

VHDL is used to create software models of each cache design investigated, so that parameter values can be easily changed, and so that no money or time is wasted by first prototyping the cache design in actual hardware. These VHDL implementations are presented, along with several test-bench programs that were used to not only validate the performance of the VHDL implementation, but also to explore the program-dependent performance factors and coherency. Several snoopy cache coherence protocols are presented at the end of the thesis, in order to suggest future research into the VHDL implementation of shared-memory multiprocessors.
# TABLE OF CONTENTS

**LIST OF FIGURES** .................................................................................................................. V

**LIST OF TABLES** ...................................................................................................................... VI

**LIST OF EQUATIONS** ............................................................................................................... VIII

**GLOSSARY** ............................................................................................................................. IX

## 1 THE MEMORY HIERARCHY

1.1 Inclusion, Coherence, and Locality ....................................................................................... 2
  1.1.1 Inclusion .................................................................................................................. 3
  1.1.2 Coherence ............................................................................................................... 3
  1.1.3 Locality .................................................................................................................. 3

## 2 INTRODUCTION TO CACHE MEMORY

2.1 Types of Cache Memories ...................................................................................................... 5
  2.1.1 Direct Mapped ........................................................................................................ 6
  2.1.2 Fully Associative ..................................................................................................... 6
  2.1.3 Set Associative ......................................................................................................... 6
  2.1.4 The Effect of Associativity on the Cache Performance ............................................ 6

2.2 Finding a Block Within the Cache ........................................................................................ 7

2.3 Writing to the Cache ............................................................................................................ 7
  2.3.1 Write-Through ........................................................................................................ 7
  2.3.2 Write-Back ............................................................................................................. 8
  2.3.3 Write-Once .............................................................................................................. 8
  2.3.4 Comparing the Strategies ....................................................................................... 9
  2.3.5 Writing to the Cache on a Cache Write Miss ........................................................... 9
    2.3.5.1 Write-Allocate ............................................................................................... 9
    2.3.5.2 No-Write-Allocate ...................................................................................... 10

2.4 Sources of Cache Misses ...................................................................................................... 10
  2.4.1 Compulsory Cache-Misses .................................................................................... 10
  2.4.2 Capacity Cache-Misses ......................................................................................... 10
  2.4.3 Conflict Cache Misses ........................................................................................... 10
  2.4.4 The Overall Effect of The Three C's on the Design of Cache Memory ..................... 11
  2.4.5 Reducing the Effect of The Three C's ................................................................... 12

2.5 Replacing a Block on a Cache-Miss ...................................................................................... 14
  2.5.1 Random Replacement ........................................................................................... 15
  2.5.2 Least Recently Used (LRU) .................................................................................. 15
  2.5.3 First In First Out (FIFO) .................................................................................... 15
  2.5.4 Pseudo-Random Replacement ............................................................................. 15
  2.5.5 Comparing the Replacement Schemes ................................................................... 16

## 3 IMPLEMENTING UNIPROCESSOR CACHE MEMORIES IN VHDL

3.1 What is VHDL? .................................................................................................................... 17
3.2 Why Use VHDL To Implement Cache Memory? ................................................................. 18
3.3 Description of Special Functions Used Throughout the VHDL Implementations ............... 18
  3.3.1 Log_Base_2 ......................................................................................................... 18
  3.3.2 Integer_to lv ....................................................................................................... 19
3.3.3 \textit{lv_to_integer}.......................................................... 19

3.4 VHDL IMPLEMENTATION OF A MEMORY UNIT................................. 20
3.4.1 Description of the Constants and Types Used Throughout the VHDL Implementation of a Memory Unit................................. 21
3.4.2 Description of Signals Used Within the Memory Unit Implementation................................. 22
3.4.3 Initialization Process.................................................... 24
3.4.4 Memory Access Process................................................... 24
3.4.5 Modify_Evict Process.................................................... 25
3.4.6 Modify_Write_Hit Process............................................... 25
3.4.7 Modify_Write_Miss Process............................................. 25

3.5 VHDL IMPLEMENTATION OF THE FIFO REPLACEMENT ALGORITHM.............. 26

3.6 VHDL IMPLEMENTATION OF THE LRU REPLACEMENT ALGORITHM................. 26

3.7 VHDL IMPLEMENTATION OF THE CACHE ARCHITECTURES.......................... 27
3.7.1 Description of the Constants and Types Used Throughout the VHDL Implementation of a Cache................................. 27
3.7.2 Description of the VHDL Implementation of a Write-Through Cache Using No-Write-Allocate on a Write-Miss ........................................... 32
    3.7.2.1 Description of the Signals Used Within the Write-Through Cache Implementations........................................... 32
    3.7.2.2 Read_or_Write Process............................................ 34
    3.7.2.3 Read_Write_Miss Process......................................... 35
    3.7.2.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes......................................... 36
    3.7.2.5 Dump Process................................................... 37
3.7.3 Description of the VHDL Implementation of a Write-Back Cache Using Write-Allocate on a Write-Miss ........................................... 37
    3.7.3.1 Description of the Signals Used Within the Write-Back Cache Implementations........................................... 38
    3.7.3.2 Read_or_Write Process............................................ 39
    3.7.3.3 Read_Write_Miss Process......................................... 40
    3.7.3.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes......................................... 40
    3.7.3.5 Dump Process................................................... 42
3.7.4 Description of the VHDL Implementation of a Write-Once Cache ............... 42
    3.7.4.1 Description of Signals Used Within the Implementation of a Write-Once Cache........................................... 43
    3.7.4.2 Read_or_Write Process............................................ 44
    3.7.4.3 Read_Write_Miss Process......................................... 45
    3.7.4.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes......................................... 46
    3.7.4.5 Dump Process................................................... 47

4 CACHE AND PROCESSOR PERFORMANCE PARAMETERS.................................. 49
4.1 CPU TIME.............................................................................. 49
4.2 AVERAGE MEMORY ACCESS TIME............................................. 51
4.3 KEEPING TRACK OF THE MEMORY ACCESS IN THE VHDL IMPLEMENTATIONS........... 51
4.4 KEEPING TRACK OF THE CACHE PERFORMANCE.................................... 51

5 TESTING THE VHDL CACHE IMPLEMENTATIONS........................................... 53
5.1 THE READ TEST ....................................................................... 54
5.2 THE WRITE TEST ...................................................................... 54
    5.2.1 Write-Back Cache Implementations..................................... 55
    5.2.2 Write-Once Cache Implementations..................................... 56
    5.2.3 Write-Through Cache Implementations.................................... 56
5.3 THE READ-WRITE-WRITE-READ (RWWR) TEST................................... 56
    5.3.1 Write-Back Cache Implementations..................................... 57
    5.3.2 Write-Once Cache Implementations..................................... 59
    5.3.3 Write-Through Cache Implementations.................................... 60
5.4 THE SUM TEST ........................................................................ 61
    5.4.1 Write-Back Direct Cache Implementation and Fully-Associative Cache Implementation with FIFO Replacement Algorithm........................................... 62
INTRODUCTION

6 IMPROVING CACHE MEMORY PERFORMANCE

7 INTRODUCTION TO SHARED-MEMORY MULTIPROCESSORS

8 SNOOPY CACHE COHERENCE протоколы для МУЛЬТИПРОЦЕССОРОВ

9 FUTURE WORK
LIST OF FIGURES

Figure 1. Design of a Five-Level Memory Hierarchy [Hwang1993, p.189] ........................................... 1
Figure 2. The Inclusion Property and Data Transfers Between Adjacent Levels of a Memory Hierarchy. [Hwang1993, page 191]........................................................................................................... 4
Figure 3. Number of Memory Accesses vs. Degree of Associativity for the Write-Back Cache78
Figure 4. Number of Memory Accesses vs. Degree of Associativity for the Write-Once Cache78
Figure 5. Number of Memory Accesses vs. Degree of Associativity for the Write-Through Cache........................................................................................................................................ 79
Figure 6. Miss Rates vs. Degree of Associativity for all of the Cache Implementations ........ 80
Figure 7. Average Memory Access Time Versus Degree of Associativity........................................... 81
Figure 8. Placement of the Victim Cache in the Memory Hierarchy [Hennessy1996, p. 398]...... 82
Figure 9. The Sub-block Placement Strategy [Hennessy1996, p. 413]............................................... 87
Figure 10. Write Once Protocol [Hwang1993 p. 354]........................................................................... 103
Figure 11. Papa’s Protocol .................................................................................................................... 105
Figure 12. RB (Read Broadcast) Protocol [Rudolph1984]................................................................. 108
Figure 13. RWB (Read and Write Broadcast) Protocol [Rudolph1984]......................................... 110
Figure 14. Firefly Coherence Protocol [Thacker1988]........................................................................ 114
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Memory Characteristics of a Typical Mainframe [Hwang1993, p. 190]</td>
<td>2</td>
</tr>
<tr>
<td>Table 2</td>
<td>The Increase of Access Time and Decrease in Bandwidth as One Moves Away from the CPU [Hennessy1996, p. 41]</td>
<td>5</td>
</tr>
<tr>
<td>Table 3</td>
<td>The Total Miss Rate for Each Cache Size and Percentage of Each According to the Three C's [Hennessy1996, p. 391]</td>
<td>11</td>
</tr>
<tr>
<td>Table 4</td>
<td>Design Target Miss Ratios for a Unified Cache [Smith1987]</td>
<td>13</td>
</tr>
<tr>
<td>Table 5</td>
<td>Actual Miss Rate Versus Block Size for Five Different-Sized Caches [Hennessy1996, p. 394]</td>
<td>13</td>
</tr>
<tr>
<td>Table 6</td>
<td>Miss Rates Comparing LRU to Random Replacement for Several Cache Sizes and Associativities [Hennessy1996, p. 379]</td>
<td>16</td>
</tr>
<tr>
<td>Table 7</td>
<td>Description of the Performance of the lv_to_integer Function</td>
<td>20</td>
</tr>
<tr>
<td>Table 8</td>
<td>Description of the Constants and Types Used in the VHDL Implementation of a Memory Unit</td>
<td>21</td>
</tr>
<tr>
<td>Table 9</td>
<td>Description of the Signals used in the VHDL Implementation of a Memory Unit</td>
<td>23</td>
</tr>
<tr>
<td>Table 10</td>
<td>Description of the Constants and Types Used in the VHDL Implementation of a Cache</td>
<td>29</td>
</tr>
<tr>
<td>Table 11</td>
<td>Data Analysis of the Read Test</td>
<td>54</td>
</tr>
<tr>
<td>Table 12</td>
<td>Data Analysis of the Write Test</td>
<td>55</td>
</tr>
<tr>
<td>Table 13</td>
<td>Data Analysis of the RWWR Test</td>
<td>57</td>
</tr>
<tr>
<td>Table 14</td>
<td>Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Back Cache Implementations</td>
<td>58</td>
</tr>
<tr>
<td>Table 15</td>
<td>Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Once Cache Implementations</td>
<td>59</td>
</tr>
<tr>
<td>Table 16</td>
<td>Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Through Cache Implementations</td>
<td>61</td>
</tr>
<tr>
<td>Table 17</td>
<td>Data Analysis of the Sum Test</td>
<td>62</td>
</tr>
<tr>
<td>Table 18</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Direct-Mapped and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation</td>
<td>63</td>
</tr>
<tr>
<td>Table 19</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation</td>
<td>64</td>
</tr>
<tr>
<td>Table 20</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Set-Associative Mapped Cache Implementation</td>
<td>65</td>
</tr>
<tr>
<td>Table 21</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Direct-Mapped Cache Implementation and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation</td>
<td>66</td>
</tr>
<tr>
<td>Table 22</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation</td>
<td>68</td>
</tr>
<tr>
<td>Table 23</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Set-Associative Mapped Cache Implementation</td>
<td>69</td>
</tr>
<tr>
<td>Table 24</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Through Direct-Mapped Cache Implementation and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation</td>
<td>70</td>
</tr>
<tr>
<td>Table 25</td>
<td>Description of the Cache and Main Memory Performance for the Sum Test on the Write-Through Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation</td>
<td>72</td>
</tr>
</tbody>
</table>
TABLE 26. DESCRIPTION OF THE CACHE AND MAIN MEMORY PERFORMANCE FOR THE SUM TEST ON THE WRITE-THROUGH SET-ASSOCIATIVE MAPPED CACHE IMPLEMENTATION

TABLE 27. THE COHERENCY STATUS OF EACH TEST-BENCH ON EACH CACHE ARCHITECTURE

TABLE 28. AVERAGE MEMORY ACCESS TIME FOR DIFFERENT CACHE SIZES AND ASSOCIATIVITIES [HENNESSY1996, P. 397]

TABLE 29. CACHE AND MAIN MEMORY PERFORMANCE OF THE SUM TEST FOR DIFFERENT DEGREES OF ASSOCIATIVITY

TABLE 29. COHERENCY STATUS OF EACH TEST-BENCH ON EACH CACHE ARCHITECTURE

TABLE 30. AVERAGE MEMORY ACCESS TIME FOR DIFFERENT DEGREES OF ASSOCIATIVITY

TABLE 31. MISS RATES FOR THE VARYING DEGREES OF ASSOCIATIVITY

TABLE 32. AVERAGE MEMORY ACCESS TIME FOR VARYING DEGREES OF ASSOCIATIVITY

TABLE 32. TYPICAL VALUES AND PARAMETERS OF A SECOND-LEVEL CACHE [HENNESSY1996, P. 471]

TABLE 33. SUMMARY OF CACHE OPTIMIZATIONS AND IMPACT ON THE THREE ASPECTS OF CACHE PERFORMANCE AND ON CACHE COMPLEXITY [HENNESSY1996, P. 427]

TABLE 34. VISUAL DESCRIPTION OF THE CACHE COHERENCE PROBLEM [HENNESSY1996, P. 655]

TABLE 35. EXAMPLE CACHE COHERENCE SOLUTION IN EXISTING MULTIPROCESSORS [DUBOIS1988]
LIST OF EQUATIONS

EQUATION 1. THE NUMBER OF BITS NEEDED FOR ADDRESSING MAIN MEMORY .............................................22
EQUATION 2. ADDRESS BITS NEEDED IN A DIRECT-MAPPED CACHE .........................................................30
EQUATION 3. ADDRESS BITS NEEDED IN THE SET-ASSOCIATIVE MAPPED CACHE ....................................31
EQUATION 4. CONVERTING THE ADDRESS PORTION OF A CACHE BLOCK IN A DIRECT-MAPPED CACHE TO
              COMPARE TO THE REQUESTED ADDRESS .................................................................................34
EQUATION 5. CONVERTING THE ADDRESS PORTION OF A BLOCK IN A SET-ASSOCIATIVE CACHE TO COMPARE
              TO THE REQUESTED ADDRESS ..................................................................................................35
EQUATION 7. MEMORY STALL CYCLES PER READ-MISS AND WRITE-MISS SEPARATELY [HENNESSY1996, P. 386] ..........................................................................................................................................49
EQUATION 9. CPU TIME IN TERMS OF MEMORY ACCESSES PER INSTRUCTION [HENNESSY1996, P. 386] ....50
EQUATION 10. MISS RATE IN TERMS OF MISSES PER INSTRUCTION [HENNESSY1996, P. 386] .................50
EQUATION 11. CPU TIME USING THE MISS RATE IN TERMS OF MISSES PER INSTRUCTION [HENNESSY1996, P. 386] ........................................................................................................................................50
EQUATION 13. AVERAGE MEMORY ACCESS TIME BROKEN DOWN INTO MEMORY ACCESSES DUE TO
              INSTRUCTION AND MEMORY ACCESSES DUE TO DATA [HENNESSY1996, P. 385] ...............51
EQUATION 16. EXPANDED AVERAGE MEMORY ACCESS TIME FORMULA FOR TWO-LEVEL CACHE
              [HENNESSY1996, P. 417] .............................................................................................................89
2 to 1 cache rule of thumb
A main rule of cache memory which states that a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2.

access time
The total time it takes to access the CPU from the ith level of the memory hierarchy.

address_in_cache
An input signal indicating the address referenced by the processor.

address_in_mem
An input signal controlled by the processor that denotes which address has been referenced by the processor.

address_needed_in_cache
An output signal that is sent to main memory which contains the address that was referenced by the processor, but which does not exist in the cache.

address_to_evict
An output signal sent to main memory which contains the address of a dirty block that was evicted from the cache.

address_to_evict_from_cache
An input signal that receives the address that was evicted from the cache.

average memory access time
A performance parameter that is a measure of the hit time, miss rate, and miss penalty associated with the cache.

bandwidth
The rate at which information is transferred between two levels of the memory hierarchy.

Berkeley Ownership cache coherence protocol
A cache coherence protocol which uses ownership-based multiprocessor cache consistency protocol.

BLOCKS_PER_SET
A VHDL constant storing the number of blocks per set, or the degree of associativity.

byte
A unit of measure that measures 8 bits.

cache block
The unit of transfer between the cache and main memory.

cache coherence problem
A problem that occurs in a shared-memory multiprocessor when two or more processors each have a local copy of a data value, and one of the processors changes the data value within its local cache, as part of an instruction that is executed in its program.

cache flush
Invalidating the contents of the cache. This usually occurs after a critical section has been left during the execution of program on a processor.

cache interrogate signal
A signal used to obtain exclusive ownership of a shared block so that modifications of this block can be performed without jeopardizing the coherency of this block with respect to other processors that may share its value.

cache memory
A small, fast, memory unit that is usually placed between the CPU and main memory and which stores the most recently used data values of the program executing on the CPU.

**CACHE_BLOCK_DATA_BITS**
A VHDL constant storing the number of bits used for data.

**CACHE_BLOCK_STATUS_BITS**
A VHDL constant storing the number of status bits to use with the cache architecture.

**cache_dump**
An input signal controlled by the processor that requests that the contents of the cache, and the FIFO queue or LRU array where applicable, be written, or "dumped," to a file.

**CACHE_SIZE**
A VHDL constant storing the cache size in units of kilobytes.

**cache-miss**
When a referenced block is not present in the cache.

**capacity cache-miss**
A type of cache miss that occurs when the cache is not capable of storing all of the data needed within a cache block during the execution of the program on the CPU.

**centralized global table**
A table that is used to store the status of each block in the cache that is shared by more than one cache.

**coherence property**
A fundamental property of the memory hierarchy which states that copies of the same information item at higher levels of the memory hierarchy must be consistent with those of the lower levels.

**coherent**
A scenario in which every read by any processor always returns the value produced by the last previous write, no matter which processor performed the write.

**cold start miss**
See compulsory cache miss.

**collision misses**
See conflict cache miss.

**compulsory cache-miss**
A type of cache miss that occurs when the very first access to a needed block results in a miss.

**conflict cache-miss**
A type of cache miss that occurs when a block is discarded in order to make room for another block that maps to the same location or same set. This type of cache miss can only occur within a set-associative or direct-mapped cache.

**copy back**
See write back cache.

**cost per byte**
The cost per byte of the ith level of the memory hierarchy. This quantity is usually estimated as the product of the cost and the size of the memory level.

**CPI**
Cycles Per Instruction. A measure of the number of CPU cycles required to execute the instruction.

**CPU**
Central Processing Unit. The part of the computer that performs the execution of a program on the computer. Also, the highest level of the memory hierarchy.

**CPU time**
A performance parameter that is a measure of the total number of clock cycles that the CPU spends executing its program, and the time that the CPU is spent waiting for a memory access to return with the necessary data.

**critical word first**
A cache improvement technique in which the required word is read into the cache first, sent to the CPU, and then the rest of the block is read into the cache.

**data_in_cache**
An input signal indicating the data associated with the address specified by the address_in_cache signal.
data_in_from_mem
An input signal that receives the data contents from main memory of the corresponding block in main memory that is indicated by the address_in_cache signal. 

data_in_mem
An input signal controlled by the processor that denotes the data to write to the cache block. 

data_out_from_mem
An output signal that returns to the cache the data corresponding to the address supplied to memory in the address_in_mem signal. 

data_read_out_cache
An output signal that returns the requested data contents to the processor. 

data_to_evict
An output signal sent to main memory that contains the data corresponding to the evicted block whose address is address_to_evict. 

data_to_evict_from_cache
An input signal to main memory that receives the data corresponding to the evicted cache block whose address is address_to_evict_from_cache. 

data_to_write_from_cache
An output signal that is sent to main memory which contains the data to write to a block in main memory when an update of a block must be performed. 

degree of associativity
n-way set associative. See 

design target miss ratio
A cache design parameter that is used to achieve a rough estimate of the expected miss ratio as a function of the cache size. 

direct mapped cache
A type of cache mapping strategy in which there is only one place that a referenced block could reside. 

DIRECT_CACHE_BLOCK_ADDRESS_BITS
A VHDL constant storing the number of address bits to use in a direct-mapped cache. 

DIRECT_CACHE_BLOCK_SIZE
A VHDL constant storing the size of a block in a direct-mapped cache. 

direct_cache_block
A VHDL sub-type used to describe a cache block in a direct-mapped cache. 

direct_cache_unit
A VHDL type used to describe a direct-mapped cache. 

direct_mapped_add_process
A VHDL process used within the cache implementations of the thesis to handle the adding of a block to a direct-mapped cache. 

dirty
A state in the write-once cache in which the block has been locally modified more than once, and hence its data value is inconsistent with that in main memory. 

dirty_bit
A bit that is used to determine the status of a block in the cache. If this bit is a one, the block is clean. If it is a zero, the block is dirty. 

dirty_block
A block in a cache that has been written to and its modification has not yet been reported to main memory. 

DRAM
Dynamic Random Access Memory. A design technology using dynamic memory cells in the design of a memory unit. In a dynamic memory cell, the contents must be occasionally refreshed so that its contents are not lost. Usually used with the design of a main memory unit. 

dump process
A VHDL process used within the cache implementations of thesis to copy the contents of the cache, LRU array, and FIFO queue (where applicable) to a file for analysis.
E

early restart
A cache improvement technique in which the requested word is sent to the CPU as soon as it arrives into the cache so that the CPU may continue with its execution as soon as possible............................ 88

ECL
Emitter Coupled Logic. A device technology used in the design of CPU's and other computer hardware 2

exclusive-modified state
A state used in Papa's protocol that indicates that no other cache has this block and that the data in the block is inconsistent with that in main memory since the data has been locally modified.............. 104

exclusive-unmodified state
A state used in Papa's protocol that indicates that no other cache has this block, and that the data in the block is consistent with that in main memory ................................................................. 104

F

fetch on write.................................................................................................................. See write-allocate

FIFO replacement strategy
First In First Out. A replacement method in which the first block that was placed into the cache is the first to be evicted from it. ................................................................. 15

Firefly cache coherence protocol
A cache coherence protocol which allows multiple caches to contain a writeable cache block simultaneously, with no-pre-arrangement required for a processor to write to a shared location..... 113

first reference miss........................................................................................................ See compulsory cache miss.

first-write
A state used in the RWB protocol that indicates the first write to a block in the cache ................. 109

fully associative cache
A cache mapping strategy in which a referenced block could reside anywhere within the cache .......... 6

fully_associative_add
A VHDL process used within the cache implementations of the thesis to handle the adding of a block to a fully-associative mapped cache .............................................................. 36

FULLY_CACHE_BLOCK_SIZE
A VHDL constant storing the size of a block in a fully-associative mapped cache ......................... 28

fully_cache_block
A VHDL sub-type used to describe a cache block in a fully-associative mapped cache .................. 28

FULLY_CACHE_BLOCK_ADDRESS_BITS
A VHDL constant storing the number of address bits to use in a fully-associative mapped cache......... 28

fully_cache_unit
A VHDL type used to describe a fully-associative mapped cache ............................................. 29

fusing
The joining of two loops that access the same array with the same loops, but perform different computations on the common data in order to reduce the number of memory accesses required to perform the calculations specified in the loops ......................................................... 84

G

Gabyte or GB
Gigabyte. A unit of measure that measures 2^30, or 1073741824 bytes........................................... 2

global miss rate
The number of misses in the cache divided by the total number of memory accesses generated by the CPU ................................................................. 89

I

IEEE
Institute of Electrical and Electronics Engineers. The governing body the determined the standards for
VHDL. ........................................................................................................................................ 17

inclusion property
A fundamental property of the memory hierarchy which states that all information contained in level i is
also present i level i+1 ........................................................................................................................... 3

init
An input signal controlled by the processor that initializes the contents of main memory .................. 23

initialization process
A VHDL process used within the thesis to initialize the contents of the main memory. ......................... 24

integer_to_lv function
A function used within the thesis that takes an integer and writes in binary ........................................ 19

invalid state
A state in the write-once cache in which the block contains no data .................................................... 8

K

Kbytes or KB
Kilobytes. A unit of measure measuring \(2^{10}\), or 1024 bytes ............................................................. 2

L

local configuration
A configuration used in the RB protocol that indicates that a variable X that is local to processing
element i will be in the local state in cache i and in the invalid state in any other cache containing
variable X. This configuration allows cache i to have exclusive ownership of X, in order to be able to
modify the contents of it .......................................................................................................................... 106

local miss rate
The number of misses in the cache divided by the total number of memory accesses to the cache .......... 89

local state
A state used with the RB protocol that indicates that the data in the cache block can be read or written to
locally, causing no bus activity ................................................................................................................ 106

log_base_2 function
A function used within the thesis that receives an integer and determines the number of bits needed to
express it in binary .................................................................................................................................. 18

LRU replacement algorithm
Least Recently Used. A replacement method in which the block that has not been used in the longest
amount of time is evicted from the cache ............................................................................................... 15

lv_to_integer
A function used within the thesis that takes a standard logic vector and converts it to its decimal integer
equivalent .............................................................................................................................................. 19

M

Mbytes or MB
Megabytes. A unit of measure that measure \(2^{20}\), or 1048576 bytes .................................................... 2

memory size
The number of bytes in level i of the memory hierarchy ........................................................................ 1

memory stall cycles
A performance parameter that is a measure of the read misses per program and the write misses per
program, and the penalty associated with each ....................................................................................... 49

memory_access process
A VHDL process used within the thesis to handle basic queries from the cache about different locations
in main memory ........................................................................................................................................ 24

memory_block
A VHDL subtype used to describe a block entry in main memory ......................................................... 22

MEMORY_BLOCK_ADDRESS_BITS
A VHDL constant used to store the number of bits in the memory block that are used for addressing... 21
MEMORY BLOCK DATA BITS
A VHDL constant used to store the number of bits in the memory block that are used for data. ............. 21
MEMORY BLOCK SIZE
A VHDL constant used to store the size of a block in main memory, in units of bits................................. 21
MEMORY SIZE
A VHDL constant used to store the value of the size of main memory in units of blocks............................  21
MEMORY_TYPE
A VHDL package used within the thesis which stores the constants used to implement a memory unit. 21
memory_unit
A VHDL type used to describe a memory unit. ........................................................................................................ 22
miss-ratio
A ratio of the number of cache misses that occur in the program to the number of memory references in the program................................................................................................................ 6
modify_evict process
A VHDL process used within the thesis to handle updates of memory blocks when a block has been evicted from the cache............................................................................................................... 25
modify_write_hit process
A VHDL process used within the thesis to handle updates of memory blocks when a block is modified for the first time in a write-once cache. ..................................................................................... 25
modify_write_miss process
A VHDL process used within the thesis to handle the update of memory blocks when a write-miss occurs in the write-once cache for the first write of the referenced block.................................................. 25
ms
Millisecond. A unit of measure that measures 10^-3 seconds ........................................................................... 2
multi-level inclusion property of second-level caches
The inclusion principle of the memory hierarchy that requires the second-level cache to contain all of the data that appears in the first-level cache............................................................................................................. 91
N
no-write-allocate
A cache design option in which, on a write-miss, the referenced block is modified directly in main memory without the block being bought into the cache first. ............................................................................. 10
ns
Nanosecond. A unit of measure that measures 10^-9 seconds ........................................................................... 2
NUMBER OF SETS
A VHDL constant storing the number of sets in the cache.................................................................................................................. 28
n-way set associative
A notation used to denote the number of blocks within a set. N represents the number of blocks in each set of the cache. .................................................................................................................. 6
O
Owned Exclusively state
A state used in the Berkeley Ownership protocol that indicates that the owning cache holds the only cached copy of the block. Updates can occur locally without first informing the other caches........ 111
Owned NonExclusively state
A state used in the Berkeley Ownership protocol that indicates that other caches have a copy of the cache block and must be informed as to any changes made locally in a cache that contains the block ...... 111
ownership-based multiprocessor cache consistency protocol
A cache coherency protocol in which a processor must own a block of memory prior to being allowed to modify its contents................................................................. 111
P

vi
package
A term used in VHDL to describe a file that contains frequently used functions and parameters. Instead of typing the same code in every file, the code is typed once, the function is given a name, and the package is instantiated in all files that will use the function......................................................... 18

pages
The unit of transfer between external disks and main memory........................................................................................................ 3

Papa's cache coherence protocol
A cache coherence protocol whose goal is to reduce bus traffic and thus decrease the wait time that a processor must wait prior to accessing the bus........................................................................................................ 104

PE
Processing Element.................................................................................................................................................................................. 106

pseudo-random replacement algorithm
A replacement algorithm used within the set-associative cache implementations of this thesis. A block is randomly chosen, however, some guidelines are followed............................................................................. 16

Q
queue
A VHDL type used to describe the LRU array or FIFO queue........................................................................................................... 29
queue_entry
A VHDL sub-type used to describe an entry in either the LRU array or FIFO queue........................................................................ 28

R
random replacement algorithm
A replacement method in which a block in the cache is randomly chosen to be evicted when the cache is full................................................................. 15

Read Broadcast (RB) cache coherence protocol
A cache coherence protocol which is based on the write once protocol, but uses the bus broadcast capabilities more efficiently for both data and event broadcasting........................................................................................................ 105

read test
A test-bench program designed to verify the read operation of the cache implementations............................................................... 54

Read Write Broadcast (RWB) cache coherence protocol
A cache coherence protocol, similar to the RB protocol, in which all of the caches read the data on the bus on both bus reads and bus writes........................................................................................................ 108

read_miss_from_cache
An input signal to main memory that receives the fact that a read-miss occurred within the cache.............................................................. 23

read_miss_out
An output signal that is sent to main memory which indicates that a read-miss occurred in the cache............................................................ 33

read_not_write
An input signal indicating whether a read or a write occurred in the program executed by the processor................................................................. 33

read_or_write process
A VHDL process used within the cache implementations of the thesis to perform either a read or a write of a block in the cache........................................................................................................ 34

read_write_miss process
A VHDL process used within the cache implementations of the thesis to report read-misses, write-misses, and updates to main memory........................................................................................................ 35

readable state
A state used with the RB protocol that indicates that the contents of the cache block are valid and consistent with memory........................................................................................................ 106

Read-For-Ownership operation
A bus operation used in the Berkeley Ownership protocol that is similar to a normal read, except that the requesting cache becomes the exclusive owner after the read completes, and all other caches invalidate any matching entries ........................................................................................................ 112
Read-Shared operation
A bus operation used in the Berkeley Ownership protocol that is a conventional read that gives the cache an UnOwned copy of the block ................................................................. 112

read-write-write-read (RWWR) test
A test-bench program designed to verify the operation of the cache implementations when reads are followed by writes to different locations, which are in turn followed by writes to the blocks that were first read, which are finally followed by reads of the blocks that were first written to ...................... 56

requested word first
critical word first ...................................................................................... See

reserved state
A state in the write-once cache in which the block has been locally modified exactly once and the results of this modification have been reported to main memory ................................................................. 8

S
segments
The unit of storage of information in the back-up storage device .................................................. 3

sequential locality
A fundamental property of the memory hierarchy which states that the execution of a program tends to follow a certain sequential order ................................................................. 3

set associative cache
A cache mapping strategy in which a referenced block is first mapped to a set and then can reside anywhere within the bounds of that set ...................................................... 6

set_associative_add
A VHDL process used within the cache implementations of the thesis to handle the adding of a block to a fully-associative mapped cache .................................................. 36

SET_CACHE_BLOCK_ADDRESS_BITS
A VHDL constant storing the number of address bits to use in a set-associative mapped cache ............ 28

SET_CACHE_BLOCK_SIZE
A VHDL constant storing the size of a block in a set-associative mapped cache ................................. 28

set_cache_block
A VHDL sub-type used to describe a cache block in a set-associative mapped cache ....................... 28

set_cache_unit
A VHDL type used to describe a set-associative mapped cache ................................................. 29

shared configuration
A configuration used in the RB protocol that implies that the shared, read-only variable Y is in the readable state in all caches that contain it. This allows any cache to read the data value associated with Y and be ensured that it is receiving the most up-to-date value .................................................. 106

shared-memory multiprocessors
A computer system that consists of at least two independent processor modules executing either a small task of a larger program, or completely independent programs. All of the processors make references to instructions and data that reside in a main memory module that is shared among the processors .. 95

shared-unmodified state
A state used in Papa's protocol that indicates that some other cache(s) may have this block and that the data in the block is consistent with that in main memory .................................................. 104

snoopy cache coherence protocol
A cache coherence protocol that requires that the responsibility of maintaining cache coherence is distributed among the local caches .................................................. 100

snoopy cache controller
A bus-watching mechanism used in shared-memory multiprocessors that watches the communication bus for all actions affecting shared data ........................................ 98

spatial locality
A fundamental property of the memory hierarchy which states that a process tends to access items whose addresses are near each other .................................................. 3
SRAM

Static Random Access Memory. A design technology using static memory cells in the design of a memory unit. In a static memory cell, the contents are retained until power is eliminated, and the contents stored in the memory cell are lost. Usually used with the design of a cache memory unit.

standard logic vector

A VHDL term used to describe an array of bits in the form of a binary number. Each bit in the array can be individually accessed.

store back

See write back cache

store through

See write through cache

sub-block placement

A cache improvement strategy in which a cache block is divided into several smaller blocks, called sub-blocks, and a valid bit is associated with each of these sub-blocks.

subtype

A VHDL term used to describe a user-defined variable type that will help describe another user-defined type.

sum test

A test-bench program designed as a sample application that could be run on the processor. This program finds the cumulative sum of all the data values in main memory.

Synopsys

A synthesizer that accepts VHDL code and returns a gate-level schematic performing the functions of the VHDL code.

T

t bytes or TB

A unit of measure that measures 2^40, or 1099511627776 bytes.

temporal locality

A fundamental property of the memory hierarchy which states that recently referenced items tend to be referenced again in the near future.

thrashing

When the upper-level of memory is much smaller than what is needed for a program, causing the CPU to run close to the lower-level memory speed, since that is where most of the data referenced resides.

transfer bandwidth

The rate at which information is transferred between levels i and i+1 of the memory hierarchy.

type

A VHDL term used to describe user-defined variables that help the programmer implement the goals of his/her program.

U

unit of transfer

The grain size for a data transfer from level i to i+1.

UnOwned state

A state used in the Berkeley Ownership protocol that indicates that several caches may have copies of this block. The block contains valid data that is possibly shared among other caches.

V

valid state

A state in the write-once cache in which the cache block contains data which has been read from main memory, but has not yet been modified.

VHDL

VHSIC Hardware Description Language. An industry standard language used to describe hardware from the abstract to the concrete level.

VHSIC
Very High Speed Integrated Circuit program. The predecessor to VHDL................................. 17

**victim cache**
A small fully-associative cache placed between the main cache and main memory that contains only
blocks that are evicted from the cache, in order to give them a "second chance" to remain in the cache
prior to being reported in main memory........................................... 82

W

**wrapped fetch**
critical word first ........................................................................... See
write around .......................................................................................... See no-write-allocate

**write back cache coherence protocol**
A cache coherence protocol in which the contents of the cache block are written to main memory only
when the block is requested by another CPU and its contents have been modified.......................... 100

write buffer
A device added to the design of the cache which stores blocks that need to be written to main memory so
that the CPU can resume the execution of its program................................................................. 8

**write once cache coherence protocol**
A cache coherence protocol that requires the first write to any cache entry to be written through to main
memory, using the write through protocol, and any subsequent write to be reported to memory after
the block is evicted from the cache, using the write back protocol........................................... 101

Write operation
A bus operation used in the Berkeley Ownership protocol that is a conventional write that causes main
memory to be updated and all cached copies to be invalidated......................................................... 112

write stall
When the CPU is halted due to the cache having to continuously write to main memory .................. 8

write test
A test-bench program designed to verify the write operation of the cache implementations.............. 54

**write through cache coherence protocol**
A cache coherence protocol in which all cache updates are reported to main memory .................. 100

**WRITE_BACK_CACHE_TYPE**
A VHDL package used within the thesis to store the parameters used within the implementations of a
write-back cache ...................................................................................... 27

**write_hit_address_modify**
An output signal sent to main memory that contains the address of a block that is being modified for the
first time as the result of a write-hit in a write-once cache.......................................................... 43

**write_hit_address_modify_from_cache**
An input signal to main memory that is used with the write-once cache implementations. This signal
receives the address of the block that has been modified if the block was first read into the cache, and
later modified, indicating that the first write to this block occurred as the result of a cache write-hit. 23

**write_hit_data_modify**
An output signal sent to main memory that contains the data associated with the block addressed by the
write_hit_address_modify signal .................................................................................. 43

**write_hit_data_modify_from_cache**
An input signal that receives the data corresponding to the address indicated by the
write_hit_address_modify_from_cache ......................................................... 23

**write_miss_address_modify**
An output signal sent to main memory that contains the address of a block that is being modified for the
first time as a result of a write-miss in a write-once cache ......................................................... 43

**write_miss_address_modify_from_cache**
An input signal that is used with the write-once cache implementations. This signal receives the address
of a block referenced by a write, however, the block does not yet exist within the cache, indicating
that a write-miss has occurred within the cache ......................................................... 23

**write_miss_data_modify**

x
An output signal sent to main memory that contains the data associated with the block addressed by the
\texttt{write\_miss\_address\_modify} signal. ................................................................. 44
\texttt{write\_miss\_data\_modify\_from\_cache}
An input signal that receives the data corresponding to the address indicated by
\texttt{write\_miss\_address\_modify\_from\_cache} ................................................................. 23
\texttt{write\_miss\_from\_cache}
An input signal to main memory that receives the fact that a write-miss occurred within the cache...... 23
\texttt{write\_miss\_out}
An output signal that is sent to main memory which indicates that a write-miss occurred in the cache... 33
\texttt{WRITE\_ONCE\_CACHE\_TYPE}
A VHDL package used within the thesis to store the parameters used within the implementations of a
write-once cache ........................................................................................................ 27
\texttt{WRITE\_THROUGH\_CACHE\_TYPE}
A VHDL package used within the thesis to store the parameters used within the implementations of a
write-through cache ................................................................................................ 27
\texttt{write-allocate}
A cache design option in which, on a write-miss, the needed block is bought into the cache and modified
without main memory being informed of the modification........................................... 9
\texttt{write-back cache}
A cache design in which information on a write is written only to the cache ....................... 8
\texttt{Write-For-Invalidation operation}
A bus operation used in the Berkeley Ownership protocol that is a quick version of the conventional
write, but does not report the modification of the data value to main memory .................. 112
\texttt{write-miss}
An event that occurs in the cache when the required data block needs to be modified, but is not resident
in the cache ............................................................................................................... 9
\texttt{write-once cache}
A cache design in which the first write to a block in the cache is reported to main memory, but all
subsequent writes to this block are not reported to main memory until the block is evicted from the
cache ......................................................................................................................... 8
\texttt{write-through cache}
A cache design in which information on a write is written to both the cache and to main memory .......... 7
\texttt{Write-Without-Invalidation operation}
A bus operation in the Berkeley Ownership protocol that causes main memory to be updated with the
new value, but any cached copies are kept valid ................................................................ 112
1 The Memory Hierarchy

When discussing the interactions between a cache and main memory, it is important to understand the concept of the memory hierarchy and the properties that justify the different levels that the hierarchy is broken into. Once the memory hierarchy is explained, and its rules are presented, not only can one better understand the communications between the cache and main memory, but those of the entire memory structure.

The memory hierarchy basically consists of five levels: the registers in the Central Processing Unit (CPU), the cache, the main memory, a disk storage device, and backup units, such as magnetic tapes. These levels are shown in Figure 1.

![Figure 1. Design of a Five-Level Memory Hierarchy [Hwang1993, p.189]](image)

There are also five parameters that help characterize the levels: the access time (ti), memory size (si), cost per byte (ci), the transfer bandwidth (bi), and the unit of transfer (xi). The access time is the total time it takes to access the CPU from the ith level of the memory hierarchy. The memory size refers to the number of bytes in level i of the memory hierarchy. The cost of the ith level is usually estimated as the cost per byte, or ci. The bandwidth is the rate at which information is transferred between levels i and i+1. Finally, the unit of transfer refers to the grain size for a data transfer from level i to i+1. As a general rule, the memory devices at a lower level in the memory hierarchy pyramid are faster to access, smaller in size,
more expensive per byte, have a higher bandwidth, and use a smaller unit of transfer as compared with those at a higher level of the memory hierarchy pyramid [Hwang1993, p. 188]. In symbolic terms, we have the following:

\[
\begin{align*}
& t_{i-1} < t_i \quad \text{access time increases as one moves up the memory hierarchy pyramid} \\
& s_{i-1} < s_i \quad \text{size increases as one moves up the memory hierarchy pyramid} \\
& c_{i-1} > c_i \quad \text{cost per bit decreases as one moves up the memory hierarchy pyramid} \\
& b_{i-1} > b_i \quad \text{bandwidth decreases as one moves up the memory hierarchy pyramid} \\
& x_{i-1} < x_i \quad \text{unit of transfer increases as one moves up the memory hierarchy pyramid}
\end{align*}
\]

These relationships are better understood by the values depicted in Table 1.

<table>
<thead>
<tr>
<th>Memory Level Characteristics</th>
<th>Level 0 CPU Registers</th>
<th>Level 1 Cache</th>
<th>Level 2 Main Memory</th>
<th>Level 3 Disk Storage</th>
<th>Level 4 Tape Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device technology</td>
<td>ECL</td>
<td>256K-bit SRAM</td>
<td>4M-bit DRAM</td>
<td>1-Gbyte magnetic disk unit</td>
<td>5-Gbyte magnetic tape unit</td>
</tr>
<tr>
<td>Access time, ( t_i )</td>
<td>10 ns</td>
<td>25-40 ns</td>
<td>60-100 ns</td>
<td>12-20 ms</td>
<td>2-20 min (search time)</td>
</tr>
<tr>
<td>Capacity, ( s_i ) (in bytes)</td>
<td>512 bytes</td>
<td>128 Kbytes</td>
<td>512 Mbytes</td>
<td>60-228 Gbytes</td>
<td>512 Gbytes-2 Tbytes</td>
</tr>
<tr>
<td>Cost, ( c_i ) (in cents/KB)</td>
<td>18,000</td>
<td>72</td>
<td>5.6</td>
<td>0.23</td>
<td>0.01</td>
</tr>
<tr>
<td>Bandwidth, ( b_i ) (in MB/s)</td>
<td>400-800</td>
<td>250-400</td>
<td>80-133</td>
<td>3 to 5</td>
<td>0.18-0.23</td>
</tr>
<tr>
<td>Unit of transfer, ( x_i )</td>
<td>4-8 bytes per word</td>
<td>32 bytes per block</td>
<td>0.5-1 Kbytes per page</td>
<td>5-512 Kbytes per file</td>
<td>Backup storage</td>
</tr>
<tr>
<td>Allocation Management</td>
<td>Compiler Assignment</td>
<td>Hardware control</td>
<td>Operating system</td>
<td>Operating system/user</td>
<td>Operating system user</td>
</tr>
</tbody>
</table>

Table 1. Memory Characteristics of a Typical Mainframe [Hwang1993, p. 190]

1.1 *Inclusion, Coherence, and Locality*

When discussing issues within the memory hierarchy, it is important to understand the three fundamental concepts of *inclusion*, *coherence*, and *locality*. In the following sections, it is assumed that the cache memory is the lowest level \( M_0 \) and communicates directly with the CPU and its registers, labeled \( M_0 \). The highest level is labeled \( M_n \), and contains all of the information words stored in the memory hierarchy, as explained in the inclusion principle below.
1.1.1 Inclusion

The inclusion property is stated as $M_1 \subset M_2 \subset M_3 \ldots \subset M_n$. This property implies that all information items are originally stored in the highest level $M_n$. During the execution of a program, portions, or subsets of $M_n$ are copied into $M_{n-1}$, and portions of $M_{n-1}$ are further copied to level $M_{n-2}$. Thus, if a word is found in level $M_i$, it will also be found in level $M_{i+1}$, $M_{i+2}$, and so on up the chain, until the highest level $M_n$ is reached. A word miss occurs when a word is searched for in level $M_i$, but is not found. If a word miss occurs in level $M_i$, it also means that a word miss occurred in all lower levels $M_{i-1}$, $M_{i-2}$, ... $M_1$.

Another concept associated with inclusion in the memory hierarchy is the method of information transfer between two levels of the hierarchy. The CPU and cache communicate through words (typically 4 or 8 bytes each). Due to the inclusion principle, the size of a cache block must then be bigger than the size of the memory word, and is typically 32 bytes, or 8 words. The cache and main memory communicate through blocks. The main memory is divided into pages (typically 128 bytes). Pages are the units of information transfer between the disk and main memory. At the highest level of the memory hierarchy, the pages of the main memory are stored as segments in the back-up storage device. These terms, and the principle of inclusion, are illustrated in Figure 2.

1.1.2 Coherence

The coherence property requires that copies of the same information item at higher levels of the memory hierarchy be consistent. This implies that if a word is modified in the cache, for example, copies of that word must be updated either immediately (write-through method) or eventually (write-back method) at all higher levels of the memory hierarchy.

1.1.3 Locality

The memory hierarchy was developed based on a behavior characteristic observed within the CPU that suggests that most of the information accessed tends to be clustered in certain regions of time, space, and ordering. Due to this behavior, we have the three dimensions of the locality principle: temporal, spatial, and sequential. Temporal locality refers to the recently referenced items, and their high likelihood of being referenced again in the near future. Spatial locality refers to the tendency of a process to access items whose addresses are near one another. Sequential locality refers to the execution of a program that follows a certain sequential order.
Figure 2. The Inclusion Property and Data Transfers Between Adjacent Levels of a Memory Hierarchy. [Hwang1993, page 191]

In designing a certain level of the memory hierarchy, the above dimensions of locality offer some suggestions to an effective design. The temporal locality dimension would lead to the popular use of the Least Recently Used (LRU) replacement algorithm and would help determine the size of memory at successive levels of the memory hierarchy. The spatial locality dimension assists in determining the size of the unit of data transfer between two adjacent levels in the hierarchy. The sequential locality dimension helps determine the grain size for optimal scheduling.
2 Introduction to Cache Memory

Cache memory is a small, fast, memory unit that is usually placed between the CPU and the physical memory. It typically stores the most recently used instructions and/or data under the assumption that these instructions/data will be used again shortly. Since instructions are rarely written to, this thesis will solely deal with accesses to data residing in the cache.

The cache memory is faster to access than the physical memory. As can be seen in Table 2, as one moves further and further away from the CPU, the size of the memory storage unit increases as well as the access time. Thus, memory units close to the CPU, such as the internal registers and the cache, require less time to access than the physical memory and external disk storage devices.

<table>
<thead>
<tr>
<th>Level Called</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&lt; 4 MB</td>
<td>&lt; 4 GB</td>
<td>&gt; 1 GB</td>
</tr>
<tr>
<td>Implementation Technology</td>
<td>Custom memory with multiple ports, CMOS or BiCMOS</td>
<td>On-chip or off-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>Magnetic disk</td>
</tr>
<tr>
<td>Access time (in ns)</td>
<td>2-5</td>
<td>3-10</td>
<td>80-400</td>
<td>5000000</td>
</tr>
<tr>
<td>Bandwidth (in MB/sec)</td>
<td>4000-32000</td>
<td>800-5000</td>
<td>400-2000</td>
<td>4-32</td>
</tr>
<tr>
<td>Managed by</td>
<td>Compiler</td>
<td>Hardware</td>
<td>Operating System</td>
<td>Operating System and User</td>
</tr>
<tr>
<td>Backed by</td>
<td>Cache</td>
<td>Main Memory</td>
<td>Disk</td>
<td>Tape</td>
</tr>
</tbody>
</table>

Table 2. The Increase of Access Time and Decrease in Bandwidth as One Moves Away from the CPU [Hennessy1996, p.41]

2.1 Types of Cache Memories

Within the cache memory, there must be a way for the CPU to know where the necessary information resides. This enforces a mapping on the data/instruction blocks in the cache. There are three general formats for the mapping of a block into the cache: direct-mapped, fully-associative mapped, and set-associative mapped.
2.1.1 Direct Mapped
In a direct-mapped cache, each block has only one place that it can go. Thus, when the CPU needs a certain data block, there is only one place that it could possibly reside in the cache. If it is not there, then the needed block must be fetched from the higher level of the memory hierarchy, and the block that was previously in this position in the cache must be evicted. In order to determine where the block resides in the cache, the block frame address is divided (modulo) by the number of blocks in the cache. The result of this integer division gives the position in the cache where the requested data block would reside.

2.1.2 Fully Associative
In a fully-associative cache, there are no restrictions as to where the block can be placed. When the CPU needs a certain data block, it must check each block that resides in the cache to determine if the required information is present in the cache. A block is only evicted from a fully-associative cache if the cache is full.

2.1.3 Set Associative
In a set-associative cache, the data block is restricted to a certain set of places. A set consists of a group of two or more blocks in the cache. When placing a block from main memory into the cache, the block is first mapped to a set, and then the block is free to go anywhere within that set. Thus, set-associative mapping combines the features of both direct-mapping and fully-associative mapping in that the block is directly mapped to a set, and then is fully associative within that set. To determine the set that the block should be placed in, the block frame address is divided (modulo) by the number of sets that are in the cache. The result of this integer division gives the set number (starting from zero and going until one minus the number of sets) that the requested block would be mapped to. A cache is said to be \textit{n-way set associative} if there are \(n\) blocks in each set.

2.1.4 The Effect of Associativity on the Cache Performance
Most caches today are set-associative, and increasing the degree of associativity has the effect of decreasing the miss-ratio since more blocks are allowed into the cache before one needs to be evicted in order to make more room in a set [Smith1987]. The highest miss ratios are thus observed in direct mapping, in which there is only one block that can be mapped to each set. Two-way associativity is slightly better by allowing two blocks within each set, and, as assumed, three-way associativity is better than two-way [Smith1987]. Eventually a point is reached where further increases in the associativity has no effect on reducing the miss ratio.
Increasing the degree of associativity also has some disadvantages. First of all, the normal parallel implementation of a cache requires that the number of comparators and data readout paths equal the degree of associativity. As the degree of associativity is thus increased, the hardware costs of implementing it increases and becomes very expensive [Smith1987]. A study on increasing the degree of associativity within a cache was performed in this thesis and is discussed on page 76.

2.2 Finding a Block Within the Cache
When the CPU needs a certain memory block, it should first check the cache to see if the block resides there. Accessing the needed block from the cache memory will be faster than retrieving it from a higher level of the memory hierarchy.

To determine if the block resides in the cache, each block in the cache includes an address tag that gives the block frame address. Each appropriate tag in the cache is checked in order to see if the block it corresponds to contains the information needed. These tags can be checked in parallel to speed up the memory access. Note that in a direct-mapped cache, only one block needs to be checked, while in a set-associative cache, all blocks within the set need to be checked. The worse case is with a fully-associative cache, where all the blocks in the cache must be checked to determine if they contain the requested information. However, once the requested block is found, the remaining blocks do not need to be checked.

2.3 Writing to the Cache
When the CPU needs to modify the value of a data block, the requested block should first be checked to see if it is in the cache. If it is, three schemes exist for writing data to the cache: write-through, write-back, and write-once. These schemes also represent the three different cache architectures that were implemented in this thesis.

2.3.1 Write-Through
In a write-through cache (also known as store through), the information is written both to the block in the cache and to the corresponding block in main memory. In implementing this strategy, both the cache and main memory contain the updated value. This is important when working with external input and output devices, or in a multiprocessor environment [Hennessy1996, p. 380].
One problem with the write-through cache is that the CPU could have to wait for all of the writes to main memory to finish before continuing with the execution of its program. When the CPU is halted due to the cache having to write the updated data value to main memory, a write stall has occurred. To reduce the effect of write stalls, a write buffer is usually included in the design of a cache. The write buffer stores blocks that need to be written to main memory so that the CPU can resume the execution of its program while the write buffer writes its contents back to main memory in parallel with the execution of the CPU. Write buffers of different sizes have been implemented in various cache designs, however, their implementation is beyond the scope of this thesis.

2.3.2 Write-Back
In a write-back cache (also known as copy back, or store back), the information is written only to the block in the cache. Thus, the modified value will be written to main memory only when this block is evicted from the cache, if it ever is. While this modified block still resides in the cache, it is referred to as a dirty block, meaning that the block has been modified while it was in the cache, but its contents have not yet been written back to main memory. The status of a block is usually determined by a dirty bit which tells whether the cache data block is clean (the contents of the cache are the same as that in memory) or dirty (the contents of the cache differ from that in memory). If the block is clean, there is no need to report its contents to the higher level of memory when it is evicted since its contents have not been changed. The dirty bit feature is especially attractive to multiprocessors since it speeds up the memory access by not requiring every write to go directly to the main memory [Hennessy1996, p. 380].

2.3.3 Write-Once
The write-once protocol, proposed by John Goodman [Goodman1983], combines the write-through and write-back protocols into one, in order to get the benefits of both. The write-once protocol requires the first write to any cache entry to be written through to main memory, using the write-through protocol. Any subsequent write to that cache block will be done locally in the cache, but the modifications will only be written to main memory after the block is evicted from the cache, hence the use of the write-back protocol.

To implement the write-once protocol, two bits are associated with each cache block. The two bits distinguish among the four states that a cache block may reside in: invalid, valid, reserved, or dirty. In the invalid state, the cache block contains no data. In the valid state, the cache block contains data which has been read from main memory and has not yet been modified. Hence, the cache and main memory are consistent with respect to this block. The reserved state indicates that the block has been locally modified exactly once since its entry into the cache, and the results of the write have been transferred to main memory. Thus once again, the cache and main memory are consistent with respect to this block. The final
state, dirty, indicates that the cache block has been modified more than once since it has been brought into the cache, and the latest change has not yet been transferred to main memory. In this state, the cache and main memory are inconsistent with respect to this cache block.

2.3.4 Comparing the Strategies
The write-through protocol has been the most common approach since it is somewhat simpler to implement than the write-back protocol, and also since it is never necessary to report anything to memory after a block is evicted from the cache. Another advantage of the write-through protocol is that the cache and main memory are always consistent with each other, whereas in the write-back protocol, the most up-to-date copy may reside in the cache. This advantage may be useful in shared-memory multiprocessors, discussed on page 95.

Despite the above advantages of the write-through protocol over the write-back protocol, the write-back protocol is starting to become very popular due to its prevention of the bandwidth bottleneck of the write-through protocol [Smith1987]. Since not every write to the cache is immediately broadcast to main memory in the write-back protocol, as is done in the write-through protocol, considerable cache to main memory bandwidth is preserved, thus making the write-back protocol preferable in shared-memory multiprocessors in order to reduce the traffic to main memory from each processor. However, the write-back protocol suffers from the cache coherence problem, described later in this thesis. Hence shared-memory multiprocessors would want a write-through cache in order to keep the cache and memory consistent. Multiprocessor cache coherence protocols will be discussed on page 96.

2.3.5 Writing to the Cache on a Cache Write Miss
The algorithms presented above only work when the memory block needed is in the cache. When the desired block is not in the cache, a write-miss has occurred, and the cache has two options of dealing with it: write allocate or no-write allocate.

2.3.5.1 Write-Allocate
In a write-allocate cache (also known as fetch on write), the needed block is loaded into the cache from the main memory, and the new data value is written to the block while it is in the cache. The modified data value is not reported to main memory until the block is removed from the cache and written back to main memory. This scheme is usually used with the write-back method since the two strategies are very similar.
2.3.5.2 No-Write-Allocate

In a no-write-allocate cache (also known as write around), the block is modified directly in main memory and not first brought into the cache. This scheme is generally used with the write-through cache scheme since the two strategies are very similar.

2.4 Sources of Cache Misses

If a needed block is not found within a cache, a cache-miss occurs and the needed block must be obtained from main memory. It is important to investigate the reasons for cache-misses occurring so that future cache designs can incorporate these scenarios into the designs of more optimal cache architectures. The three main sources of cache-misses are known as The Three C's: compulsory, capacity, and conflict.

2.4.1 Compulsory Cache-Misses

A compulsory cache-miss means that the very first access to the needed block results in a miss. Under this type of cache-miss, the needed block must be brought into the cache from main memory. This type of cache-miss is also known as cold start miss or first reference miss.

2.4.2 Capacity Cache-Misses

During the execution of a large program or process, it may be impossible for the cache to contain all the blocks needed during its execution. Therefore, a capacity miss occurs when the needed block has been recently evicted from the cache to make room for another block and now must be placed back in the cache since it is needed again.

2.4.3 Conflict Cache Misses

Conflict misses only occur in set-associative or direct-mapped caches, since these are the only cache architectures in which a block is evicted in order to make room for another block in the set. Thus, if a set-associative or direct-mapped placement strategy is used, conflict misses, in addition to compulsory and capacity misses, will occur since too many blocks will be mapped to a set, or a particular position. Conflict misses are also known as collision misses.
2.4.4 The Overall Effect of The Three C's on the Design of Cache Memory

To determine the overall effect of the Three C's on a cache memory, a simulation was performed on a cache with 32-byte blocks and using an LRU replacement scheme on a DECstation 5000 computer. The results of the simulation are reported in [Hennessy1996, p. 391] and are shown in Table 3.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Degree of Associativity</th>
<th>Total Miss Rate</th>
<th>Compulsory</th>
<th>Capacity</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>1-way</td>
<td>0.133</td>
<td>0.002</td>
<td>1%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>2-way</td>
<td>0.105</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>4-way</td>
<td>0.095</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>8-way</td>
<td>0.087</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>2 KB</td>
<td>1-way</td>
<td>0.098</td>
<td>0.002</td>
<td>2%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>2-way</td>
<td>0.076</td>
<td>0.002</td>
<td>2%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>4-way</td>
<td>0.064</td>
<td>0.002</td>
<td>3%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>8-way</td>
<td>0.054</td>
<td>0.002</td>
<td>4%</td>
<td>0.044</td>
</tr>
<tr>
<td>4 KB</td>
<td>1-way</td>
<td>0.072</td>
<td>0.002</td>
<td>3%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>2-way</td>
<td>0.057</td>
<td>0.002</td>
<td>3%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>4-way</td>
<td>0.049</td>
<td>0.002</td>
<td>4%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>8-way</td>
<td>0.039</td>
<td>0.002</td>
<td>5%</td>
<td>0.031</td>
</tr>
<tr>
<td>8 KB</td>
<td>1-way</td>
<td>0.046</td>
<td>0.002</td>
<td>4%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>2-way</td>
<td>0.038</td>
<td>0.002</td>
<td>5%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>4-way</td>
<td>0.035</td>
<td>0.002</td>
<td>5%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>8-way</td>
<td>0.029</td>
<td>0.002</td>
<td>5%</td>
<td>0.023</td>
</tr>
<tr>
<td>16 KB</td>
<td>1-way</td>
<td>0.029</td>
<td>0.002</td>
<td>7%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>2-way</td>
<td>0.022</td>
<td>0.002</td>
<td>9%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>4-way</td>
<td>0.020</td>
<td>0.002</td>
<td>10%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>8-way</td>
<td>0.018</td>
<td>0.002</td>
<td>10%</td>
<td>0.015</td>
</tr>
<tr>
<td>32 KB</td>
<td>1-way</td>
<td>0.020</td>
<td>0.002</td>
<td>10%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>2-way</td>
<td>0.014</td>
<td>0.002</td>
<td>14%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>4-way</td>
<td>0.013</td>
<td>0.002</td>
<td>15%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>8-way</td>
<td>0.013</td>
<td>0.002</td>
<td>15%</td>
<td>0.010</td>
</tr>
<tr>
<td>64 KB</td>
<td>1-way</td>
<td>0.014</td>
<td>0.002</td>
<td>14%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>2-way</td>
<td>0.010</td>
<td>0.002</td>
<td>20%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>4-way</td>
<td>0.009</td>
<td>0.002</td>
<td>21%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>8-way</td>
<td>0.009</td>
<td>0.002</td>
<td>22%</td>
<td>0.007</td>
</tr>
<tr>
<td>128 KB</td>
<td>1-way</td>
<td>0.010</td>
<td>0.002</td>
<td>20%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>2-way</td>
<td>0.007</td>
<td>0.002</td>
<td>29%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>4-way</td>
<td>0.006</td>
<td>0.002</td>
<td>31%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>8-way</td>
<td>0.006</td>
<td>0.002</td>
<td>31%</td>
<td>0.004</td>
</tr>
</tbody>
</table>

Table 3. The Total Miss Rate for Each Cache Size and Percentage of Each According to the Three C's [Hennessy1996, p. 391]
From the table, one can observe that summing the percentages of the three components of the miss-rate yields 100%. In addition, compulsory misses are independent of the cache size, while capacity misses decrease as the cache size increases. Another important observation is that the results of this simulation support the $2:1$ cache rule of thumb which states that a direct-mapped cache of size $N$ has about the same miss-rate as a 2-way set-associative cache of size $N/2$ [Hennessy1996, p.391].

2.4.5 Reducing the Effect of The Three C's

It is important to note that although the fully-associative placement strategy does not suffer from conflict cache misses, the high degree of associativity in the fully-associative cache can become very expensive in hardware, and may even slow down the cache access time, thus degrading the overall processor performance [Hennessy1996, p. 392]. The access time will be slower in a fully-associative mapped cache since the cache must search through all of its entries in order to determine if the requested block is present, as opposed to a limited number of blocks, as in the direct-mapped and set-associative mapped strategies.

In order to reduce the capacity misses, one may invest in buying larger cache memory chips in order to avoid thrashing. Thrashing occurs when the lower-level of the memory hierarchy is much smaller than what is needed for a program, causing the machine to run close to the higher-level memory speed, since that is where most of the data required by the processor will reside.

As the cache size increases, the miss-ratio will decrease since more data is capable of being stored in the cache, and the odds are in favor of the requested block residing in the cache [Smith1993]. However, a large cache tends to be slow, so a tradeoff exists between the speed of the cache and the cache miss ratio.

In designing a cache, a design target miss ratio is used in order to achieve a rough estimate as to the expected miss ratio as a function of the cache size. Smith proposes some sample design target miss ratios in [Smith1993], which are outlined in Table 4. From the values in the table, Smith proves that the miss ratio tends to drop roughly as $\frac{1}{\sqrt{C}}$, where $C$ is the cache size. Thus quadrupling the cache size will cut the miss ratio in half, at the expense of a slower cache. In Table 4, line size and cache block size are synonymous with the line size being measured in bits, cache size in kilobytes (KB), and the miss ratio is a ratio of the number of cache-misses to the number of memory references in the program being executed on the processor.
Compulsory misses are independent of the cache size, however, larger block sizes can reduce compulsory misses, as suggested by the principle of spatial locality, since more data will be stored in a cache block. However, larger block sizes also reduce the number of blocks that can be placed in the cache. This reduction of the number of blocks that may be stored in the cache may increase the number of conflict misses, and thus degrade the overall processor performance [Hennessy1996, p. 393].

The effect of increasing the block size is shown in Table 5. In Table 5, the actual miss rate versus the block size for five different-sized caches are shown. Note that for a 1 KB cache, all block sizes 64-bytes and larger have a higher than miss rate than the 32-byte block, which shows that increasing the size of the cache block actually increases the miss rate. Using the example shown in Table 5, it would take a 256 KB cache with a cache block size of 256-bytes to reduce the miss rate, since this is the only type of cache in the table in which the miss rate is less than or equal to all smaller block sizes.

<table>
<thead>
<tr>
<th>Line Size</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.717</td>
<td>0.556</td>
<td>0.500</td>
<td>0.722</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0.686</td>
<td>0.488</td>
<td>0.400</td>
<td>0.410</td>
<td>0.672</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0.674</td>
<td>0.467</td>
<td>0.350</td>
<td>0.330</td>
<td>0.400</td>
<td>0.630</td>
</tr>
<tr>
<td>256</td>
<td>0.643</td>
<td>0.420</td>
<td>0.300</td>
<td>0.258</td>
<td>0.276</td>
<td>0.386</td>
</tr>
<tr>
<td>512</td>
<td>0.596</td>
<td>0.390</td>
<td>0.270</td>
<td>0.216</td>
<td>0.197</td>
<td>0.257</td>
</tr>
<tr>
<td>1024</td>
<td>0.473</td>
<td>0.309</td>
<td>0.210</td>
<td>0.162</td>
<td>0.137</td>
<td>0.151</td>
</tr>
<tr>
<td>2048</td>
<td>0.405</td>
<td>0.258</td>
<td>0.170</td>
<td>0.124</td>
<td>0.098</td>
<td>0.093</td>
</tr>
<tr>
<td>4096</td>
<td>0.329</td>
<td>0.193</td>
<td>0.120</td>
<td>0.082</td>
<td>0.059</td>
<td>0.050</td>
</tr>
<tr>
<td>8192</td>
<td>0.232</td>
<td>0.135</td>
<td>0.080</td>
<td>0.050</td>
<td>0.033</td>
<td>0.025</td>
</tr>
<tr>
<td>16384</td>
<td>0.182</td>
<td>0.103</td>
<td>0.060</td>
<td>0.036</td>
<td>0.023</td>
<td>0.016</td>
</tr>
<tr>
<td>32768</td>
<td>0.124</td>
<td>0.070</td>
<td>0.040</td>
<td>0.024</td>
<td>0.014</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Table 4. Design Target Miss Ratios for a Unified Cache [Smith1987].

<table>
<thead>
<tr>
<th>Block Size</th>
<th>1 KB</th>
<th>4 KB</th>
<th>16 KB</th>
<th>64 KB</th>
<th>256 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>15.05%</td>
<td>8.57%</td>
<td>3.94%</td>
<td>2.04%</td>
<td>1.09%</td>
</tr>
<tr>
<td>32 bytes</td>
<td>13.34%</td>
<td>7.24%</td>
<td>2.87%</td>
<td>1.35%</td>
<td>0.70%</td>
</tr>
<tr>
<td>64 bytes</td>
<td>13.76%</td>
<td>7.00%</td>
<td>2.64%</td>
<td>1.06%</td>
<td>0.51%</td>
</tr>
<tr>
<td>128 bytes</td>
<td>16.64%</td>
<td>7.78%</td>
<td>2.77%</td>
<td>1.02%</td>
<td>0.49%</td>
</tr>
<tr>
<td>256 bytes</td>
<td>22.01%</td>
<td>9.51%</td>
<td>3.29%</td>
<td>1.15%</td>
<td>0.49%</td>
</tr>
</tbody>
</table>

Table 5. Actual Miss Rate Versus Block Size for Five Different-Sized Caches [Hennessy1996, p. 394]

The main goal of the cache designer in using larger cache blocks, and in reducing the effect of The Three C's in general, is to reduce both the miss-rate and the miss penalty. The selection of the block size depends on the latency and bandwidth of the higher-level memory; a main memory with high latency and high
bandwidth encourages larger block sizes in the cache since the cache will get many more bytes per miss for a small increase in miss penalty. Conversely, a main memory with low latency and low bandwidth encourages small cache block sizes since there is little time saved from transferring a larger block—twice the miss penalty of a small block may be close to the miss penalty of a block twice the size—and the larger number of small blocks may reduce conflict misses [Hennessy1996, p. 395].

Additional methods for reducing the effects of the Three C’s are discussed in the section on improving the cache performance, starting on page 75.

2.5 Replacing a Block on a Cache-Miss
When the CPU looks in the cache and determines that the needed data block is not there, this required block must be brought into the cache from main memory. If the cache is full, a block must be evicted, reporting its data contents to main memory if it is dirty. This is called a cache-miss since the CPU attempted to find the needed block in the cache, but did not ("missed").

It makes the most sense that the first blocks that should leave the cache are the ones that are invalid, since the data in these blocks are no longer consistent with those in the higher levels of the memory hierarchy. Keeping these "dirty" blocks in the cache not only wastes space, it also creates the hazard of the CPU reading the incorrect data values and using them in its program, thus producing erroneous results.

Assuming there are no invalid blocks in the cache, various schemes have been suggested for the removal of blocks from the cache. In a direct-mapped cache, there is only one place that a block can go. Therefore, if the necessary data is not in this position of the cache, the block that was previously there is removed, and the new block is written into this position. The case of a direct-mapped cache is the easiest to deal with since there is only one place that the block can reside, and hence only one block that is candidate to be removed from the cache.

For both fully- and set-associative caches, different algorithms have been proposed. These algorithms—random replacement, least recently used (LRU), first in first out (FIFO), and pseudo-random—are discussed in the following sections.
2.5.1 Random Replacement
In the random replacement algorithm, a block is randomly chosen to be removed from the cache and written back to main memory, if its contents are dirty. Though this scheme seems as if it would provide bad results, it has been shown to outperform the First in First Out (FIFO) strategy [Hennessy1990, p. 412].

The random replacement algorithm was not implemented in this thesis since there is no "random" function that generates a random number in VHDL. Due to this inconvenience, the pseudo-random replacement algorithm was developed and implemented instead.

2.5.2 Least Recently Used (LRU)
The random replacement scheme suffers from the fact that the block removed could contain information that will be referenced again soon. In an LRU replacement cache, the block replaced is the one that has been unused for the longest amount of time. In order to keep track of which blocks have been used when, special hardware must be used for each block. This hardware records when a block has been referenced, and how long it has been since the block was last referenced. When the cache becomes full and a block needs to be evicted from the cache, the block that has not been referenced in the longest amount of time is removed from the cache. This extra hardware will become expensive as the number of blocks in the cache increases.

2.5.3 First In First Out (FIFO)
As the number of blocks in the cache increases, the LRU algorithm becomes too expensive to implement. The FIFO replacement scheme takes away some of the incurring cost. In this scheme, a queue is kept of all the memory references as they occur. The block removed from the cache is the block that was referenced N references ago, regardless of how recently it has been referenced, where N is the size of the queue. The value of N depends on the available hardware to implement the queue needed to support this replacement method. In this thesis, the size of the queue is the same size as the cache. Though this replacement technique is generally cheaper to implement than LRU, its performance has been proven inferior to that of the random replacement method [Hennessy1996, p. 379].

2.5.4 Pseudo-Random Replacement
In the set-associative cache implementations, it is difficult to manage a FIFO queue or LRU array for each set that the cache may be divided into. In addition, since the number of sets is a varying parameter in this thesis, it is not known in advance how many FIFO queues or LRU arrays need to be managed. Therefore, a
pseudo-random replacement algorithm is used for each of the set-associative mapped cache implementations of this thesis. In a pseudo-random replacement algorithm, a block is chosen from the set at random, however some guidelines are followed.

In the write-through cache, none of the blocks in the set will be invalid, therefore one of the blocks must be randomly chosen to be evicted from the cache. In this thesis, the block residing at the end of the set is always evicted. In other words, if the set consist of four blocks, the data in the fourth block is always overwritten by the new incoming data, regardless of how recently this block was last referenced.

In the write-back cache, the set is first searched for invalid blocks, since it is important to re-establish the coherency between the cache and main memory as soon as possible. The first invalid block found in the set, traversing it from the beginning of the set to the end, is evicted from the cache, and the new data is written into this position. If no invalid blocks are found in the set, the block residing at the end of the set is overwritten with the new data, as explained in the write-through cache.

In the write-once cache, the set is first searched for invalid blocks. since, once again, it is important to re-establish the coherency between the cache and main memory. The first invalid block found in the set, again traversing it from the beginning of the set to the end, is evicted, and the new data contents are written into this position. If no invalid blocks are found in the set, the set is traversed from beginning to end searching for reserved blocks. The first reserved block that is found is evicted from the cache, and the new incoming data is written into this position. If no reserved blocks are found in the set, the block residing at the end of the set is evicted, and the new data is written into this position, as explained in the write-through cache.

2.5.5 Comparing the Replacement Schemes
Table 6 shows the comparison of the LRU and random replacement schemes for different size caches and set-associativity. From the table, it can be seen that for large cache sizes, there is very little difference (if any) between the LRU and random replacement methods.

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

Table 6. Miss Rates Comparing LRU to Random Replacement for Several Cache Sizes and Associativities [Hennessy1996, p. 379]
3 Implementing Uniprocessor Cache Memories in VHDL

The main purpose of the thesis was to implement the uniprocessor cache strategies described in the chapter starting on page 5, perform test-benches on them to verify that the implementations were correct, and to perform data analysis on them to determine which implementation yields the lowest miss ratio, and hence the best performance. This chapter will discuss the VHDL implementation of the cache architectures, and why this programming language was chosen for accomplishing the goals of this thesis.

3.1 What is VHDL?

VHSIC Hardware Description Language, or VHDL, is an industry standard language used to describe hardware from the abstract to the concrete level [Perry1994]. It developed as an adaptation of the already existing very high speed integrated circuit (VHSIC) program that was funded by the Department of Defense in the late 1970's and early 1980's. VHSIC was used to design the next generation of integrated circuits, however, its users were designing complex circuits, and these circuit designers found the VHSIC tools inadequate to design these complex circuits. The tools used within VHSIC were mostly based on the gate level, and thus creating a complex circuit, or hardware device, consisting of hundreds of thousands of gates, was a challenging task using these gate level tools. Therefore, these designers were in need of a new hardware description standard.

VHDL was proposed as a standard in 1981, and the two main goals of the standard programming language were to provide a language that could describe the complex circuits being built, and to provide a standard that all personnel involved in the design and fabrication of complex circuits and hardware could understand [Perry1994]. In addition, VHDL would allow the description of a system to consist of a structure of subsystems, with further explanations of how each of the subsystems were interconnected. Furthermore, familiar programming language constructs, similar to those used in High Level Languages such as C and Java, were to be used in VHDL to describe the functional behavior of these complex circuits. Also, the design of a system could be simulated before it was manufactured. Doing this would allow circuit designers to compare alternative designs and test for correctness without delay, and without spending exorbitant amounts of money on prototypes [Ashenden1996].

In 1987, after many revisions, VHDL became the Institute of Electrical and Electronics Engineers (IEEE) standard IEEE 1076-1987. After standard revisions, and listening to the comments users had of VHDL
1987, a new version of VHDL was issued in 1993, VHDL-93. VHDL-93 was used in all of the implementations and compilations used in this thesis.

### 3.2 Why Use VHDL to Implement Cache Memory?

The design of cache memories entails many things occurring at the same time. The occurrence of a write-miss, for example, requires that the requested data block be searched for, and brought in, from main memory, and then modified in the cache (or in both the cache and main memory, as in the write-once cache architecture). Hence, in order to create a software simulation that implements all of the main features of a cache memory, the language used must be capable of handling many concurrent processes.

VHDL was chosen mainly because of its capability to have many processes running simultaneously. In addition, VHDL directly models a hardware circuit, which a cache memory is. By directly modeling the cache hardware in VHDL, any design changes made to existing VHDL implementations can be synthesized and routed using synthesizers, such as Synopsys. The use of a synthesizer eliminates the need to first prototype the design in hardware, which wastes money and delays the shipment of the product to the market.

### 3.3 Description of Special Functions Used Throughout the VHDL Implementations

To ease in the implementations of the cache implementations, it was helpful to create a VHDL package that would be instantiated in all of the VHDL files. This package, appropriately called THESIS_FUNCTIONS, contains the VHDL code to perform the functions log_base_2, integer_to_lv, and lv_to_integer.

#### 3.3.1 Log_Base_2

The log_base_2 function takes a non-negative integer as input and returns the number of bits needed in order to express that non-negative integer in binary. A FOR loop is used to iterate among all non-negative integers, from zero to the non-negative integer input itself, until the integer 2 raised to the iterated non-negative integer is greater than or equal to the non-negative integer supplied as the input to the function. For example, assume the non-negative integer 17 is the input to the function. The function starts with the zero, and realizes that $2^0$ is 1, which is less than 17, so the function continues, testing the number 1. $2^1$ is
2, which is also less than 17, so again the function continues, now testing the non-negative integer 2.

Eventually, the number 5 is tested, and since $2^4$ is 32, which is greater than 17, the IF loop is entered and the non-negative integer 5 is returned as the result of the function. Since a RETURN statement has been reached, the function is complete, and the next statement in the processor’s program is executed.

3.3.2 Integer_to_lv

The integer_to_lv function takes two non-negative integers as input parameters—the number to convert to a logic vector, and the length of the logic vector that should be returned—and returns a standard logic vector, of the length provided, expressing the supplied non-negative integer as a binary number. In many cases, the result of the log_base_2 function will be supplied as the length parameter of the integer_to_lv function, therefore no other assertion is necessary to confirm that the length provided to the integer_to_lv is sufficient to correctly express the supplied non-negative integer in binary.

The integer_to_lv function first initializes a standard logic vector, of the length supplied, to all zeros. Then, starting with the least significant bit of the standard logic vector, the non-negative integer is divided by two, and if the remainder is zero, a zero is written into the least significant bit of the standard logic vector, however, this does not modify the existing standard logic vector since all bits of the vector have already been initialized to zeros. If the remainder of the division is a one, a one is written into the least significant bit of the standard logic vector. The new non-negative integer is then divided by two, and the remainder, either a zero or a one, is written into the next most significant bit of the standard logic vector. This process is continued until either the quotient part of the division is zero, or the maximum logic vector length is reached, whichever occurs first. At that point, the standard logic vector is returned as the result of the function.

3.3.3 lv_to_integer

The lv_to_integer function accepts a standard logic vector that represents a binary number, and converts the vector into its non-negative integer equivalent. The function traverses the vector from most-significant to least-significant bit until all of the bits of the vector have been investigated. The lv_to_integer function first initializes the result variable to zero. Next, the function multiplies this value by two, since the vector is being traversed from most-significant to least-significant, and then looks at the value of the bit being investigated. If this bit is a 1, one is added to the temporary result value. Otherwise, the result value is not modified and the function repeats the process on the next bit.
As an example of the performance of the lv_to_integer function, consider the standard logic vector 1011, which represents the decimal value eleven. Table 7, shows the iterations used to convert the standard logic vector into an integer with the lv_to_integer function.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Index Value</th>
<th>Index Value * 2</th>
<th>Bit Value</th>
<th>Final Index Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First (Most Sig.)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Second</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Third</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Fourth (Least Sig.)</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 7. Description of the Performance of the lv_to_integer Function

In the first step, the index value has been initialized to zero, thus multiplying this value by two causes no change. Next, the most significant bit is investigated and found to be a one, therefore a one is added to the index value, which is the temporary result, and the index value is now 1.

This process is repeated with the next bit. Now, the index value is one, so when the index value is multiplied by two, the new index value is two. The second bit value from the left is investigated and is found to be a zero, therefore no modification to the index value is made.

This process is repeated for the remaining two bits in the standard logic vector, and the final index value is found to be eleven, which corresponds to the decimal equivalent of the vector 1011.

### 3.4 VHDL Implementation of a Memory Unit

The memory unit should be capable of providing data to a cache and also to update the contents of a memory block that was modified within the cache. The memory unit must be able to recognize the above two actions in all of the cache architectures implemented. Since different cache architectures use different signals to report the occurrence of reads and writes, different input signals to main memory exist, as well as different processes performing the same basic procedures on the memory unit, the only difference being in the input signals used.
3.4.1 Description of the Constants and Types Used Throughout the VHDL Implementation of a Memory Unit

To aide in the VHDL implementation of a memory unit, the MEMORY_TYPE package was created and is instantiated in all files that use a memory unit. The constants that are defined in the MEMORY_TYPE package are listed and described in Table 8.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_SIZE</td>
<td>The size of the memory unit in megabytes</td>
<td>Integer</td>
<td>Varies</td>
</tr>
<tr>
<td>MEMORY_BLOCK_SIZE</td>
<td>The size of a memory block</td>
<td>Integer</td>
<td>Varies</td>
</tr>
<tr>
<td>MEMORY_BLOCK_ADDRESS_BITS</td>
<td>The number of bits needed to reference all of the blocks in main memory</td>
<td>Integer</td>
<td>( \log_{2} (\text{MEMORY_SIZE}) + 20 - 3 )</td>
</tr>
<tr>
<td>MEMORY_BLOCK_DATA_BITS</td>
<td>The number of bits in a main memory block that are used to store data</td>
<td>Integer</td>
<td>MEMORY_BLOCK_SIZE</td>
</tr>
<tr>
<td>memory_block</td>
<td>Sub-type used to define a memory block</td>
<td>Standard logic vector</td>
<td>Range is ( (\text{MEMORY_BLOCK_SIZE} - 1 \text{ downto 0}) )</td>
</tr>
<tr>
<td>memory_unit</td>
<td>Type used to define a unit of memory</td>
<td>Array</td>
<td>Bounds are ( (2^{(\text{MEMORY_BLOCK_ADDRESS_BITS} - 1 \text{ downto 0})} )</td>
</tr>
</tbody>
</table>

Table 8. Description of the Constants and Types Used in the VHDL Implementation of a Memory Unit

The package contains constants describing the memory size, the memory block size, the number of bits used for addressing, and the number of bits used for data. The memory size and memory block size are non-negative integers that can be modified within this package. Modification of these values affect all files that have this package instantiated in it, provided that the above mentioned files are recompiled prior to their simulation.

The number of bits used for memory block addressing is computed using the \( \log_{2} \) function with the memory size as its input. Since the memory size is in units of megabytes \( (2^{20}) \), the result of the \( \log_{2} \) function is added to the number twenty. In addition, each block of main memory stores 64 bits of data (eight bytes), therefore three bits must be removed from the addressing of the individual blocks so that they can be used to reference one of the eight sub-blocks that each memory location is divided into. The final formula to compute the number of bits used in the addressing of main memory is shown in Equation 1.
\[ \log_{\text{base}}_2(\text{MEMORY\_SIZE}) + 20 - 3 \]

Equation 1. The number of bits needed for addressing main memory

The result of Equation 1 will indicate how many bits are necessary to address all of the blocks in the memory unit.

The number of data bits is fixed at 64 and should not be modified. When a block of memory is requested by the cache, all 64 data bits are returned. It is up to the user to determine which of the eight sub-blocks are used in the specific program that is being executed on the processor.

In addition to the above constant values, the package also contains a subtype used to describe a memory block and a type used to describe the memory unit. These types use the above constants as bounds on their size, so modifications to the above parameters will also affect the memory_block and memory_unit types.

3.4.2 Description of Signals Used Within the Memory Unit Implementation

To implement the memory unit in VHDL, several signals were used to receive information from the caches, or to return information requested by the caches. Since different cache architectures use different signals to report read- and write-misses, the memory unit must be able to recognize all of these signals and determine which operation was issued by the cache. These signals are listed and described in Table 9.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description of Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_in_mem</td>
<td>an input signal controlled by the processor that denotes which address has been referenced by the processor. This signal is an integer</td>
</tr>
<tr>
<td>data_in_mem</td>
<td>an input signal controlled by the processor that denotes the data to write to the cache block. If an event occurs on this signal, it can be assumed that a write has been issued by the processor. This signal is a standard logic vector of length 64 bits</td>
</tr>
<tr>
<td>address_to_evict_from_cache</td>
<td>an input signal that receives the address that was evicted from the cache. In the event that the data contents of the evicted block differ from that of main memory, main memory must be updated. This signal is an integer</td>
</tr>
<tr>
<td>data_to_evict_from_cache</td>
<td>an input signal that receives the data corresponding to the evicted cache block whose address is address_to_evict_from_cache. This signal is a standard logic vector of length 64 bits</td>
</tr>
<tr>
<td>Signal Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>write_hit_address_modify_from_cache</strong></td>
<td>an input signal that is used with the write-once cache implementations. This signal receives the address of the block that has been modified if the block was first read into the cache, and later modified, indicating that the first write to this block occurred as the result of a cache write-hit. In the write-once protocol, the first write to a block that exists in the cache is relayed to main memory, but all subsequent writes are not reported to main memory until after the block has been evicted from the cache. This signal is an integer.</td>
</tr>
<tr>
<td><strong>write_hit_data_modify_from_cache</strong></td>
<td>an input signal that receives the data corresponding to the address indicated by <strong>write_hit_address_modify_from_cache</strong>. This signal is a standard logic vector of length 64 bits</td>
</tr>
<tr>
<td><strong>write_miss_address_modify_from_cache</strong></td>
<td>an input signal that is used with the write-once cache implementations. This signal receives the address of a block referenced by a write, however, the block does not yet exist within the cache, indicating that a write-miss has occurred within the cache. This signal is an integer.</td>
</tr>
<tr>
<td><strong>write_miss_data_modify_from_cache</strong></td>
<td>an input signal that receives the data corresponding to the address indicated by <strong>write_miss_address_modify_from_cache</strong>. This signal is a standard logic vector of length 64 bits</td>
</tr>
<tr>
<td><strong>read_miss_from_cache</strong></td>
<td>an input signal that receives the fact that a read-miss occurred within the cache. This signal is a standard logic bit, where a one indicates that a read-miss occurred and a zero indicates that a read-miss did not occur.</td>
</tr>
<tr>
<td><strong>write_miss_from_cache</strong></td>
<td>an input signal that receives the fact that a write-miss occurred within the cache. This signal is a standard logic bit, where a one indicates that a write-miss occurred and a zero indicates that a write-miss did not occur.</td>
</tr>
<tr>
<td><strong>init</strong></td>
<td>an input signal controlled by the processor that initializes the contents of main memory. The initialization process, described on page 24, is sensitive to this signal, and causes all memory data bits to be initialized to their starting values. This signal is a standard logic bit, with a one indicating that the initialization procedure should be executed and a zero indicating that it should not be executed.</td>
</tr>
<tr>
<td><strong>memory_dump</strong></td>
<td>an input signal controlled by the processor that writes the contents of main memory to an output file. This signal is a standard logic bit</td>
</tr>
<tr>
<td><strong>data_out_from_mem</strong></td>
<td>an output signal that returns the data of the corresponding address supplied to memory by the <strong>address_in_mem</strong> signal. This signal is a standard logic vector of length 64 bits</td>
</tr>
</tbody>
</table>

**Table 9. Description of the Signals used in the VHDL Implementation of a Memory Unit**

The signals described in Table 9, together with the processes described below, are used to handle the main memory operations that are requested by the caches and processors.
3.4.3 Initialization Process

The initialization process is sensitive to the init signal, which is toggled by the processor. When an event occurs on the init signal, and its final value is a one, the initialization process is executed. In this process, the memory unit data values are initialized to their starting values. In this thesis, the initialization process assigns the data value equivalent to the location of each memory block in main memory. For example, the memory block in location eleven, will have its data block initialized to the value eleven. The block in location fifteen will have its data portion initialized to fifteen, and so on.

3.4.4 Memory_Access Process

The memory_access process is sensitive to the read_miss_from_cache, write_miss_from_cache, address_in_mem, data_in_mem, write_hit_data_modify_from_cache, and write_miss_data_modify_from_cache signals, all of which receive information from the cache.

On a read-miss from the cache, the data portion of the requested memory block is returned to the cache which requested it via the data_out_from_mem signal, and the number of memory accesses is increased. Since main memory is the lowest level of the memory hierarchy implemented in this thesis, all data that would ever be needed by the processor is guaranteed to reside in main memory, as stated in the inclusion principle. Therefore, no read-misses can occur in main memory.

On a write-miss, if there was no event on the write_miss_data_modify_from_cache signal, and the contents of the data_in_mem signal are defined (i.e. the signal does not contain all ‘U’), then the data contents of the memory block are updated to contain the new data value specified by the data_in_mem signal, and the number of memory accesses is increased. It is necessary for this process to isolate a write-miss from the write-once cache, indicated by an event on the write_miss_data_modify_from_cache signal, since there is a separate process, the modify_write_hit process explained on page 25, that is responsible for handling write-misses from the write-once cache.

On a main memory update from a write-through cache, the data contents of the block specified by the address_in_mem signal is modified to contain the new value specified by the data_in_mem signal, and the number of memory accesses is increased. The write-through cache is the only architecture that would cause this portion of the memory_access process to be executed since, with the exception of the first write to a block in the write-once cache architecture, the write-through cache is the only cache architecture in which every write is reported directly to main memory.
3.4.5 Modify_Evict Process

The `modify_evict` process is sensitive to the `address_to_evict_from_cache` and `data_to_evict_from_cache` signals, both of which receive their information from the cache. This process assumes that a dirty block has been evicted from the cache, and the corresponding memory block must be updated in main memory. This process is primarily used with the write-back and write-once cache implementations, since these are the only two cache implementations in which cache data blocks may be inconsistent with those in main memory. The process modifies the data portion of the memory block indicated by the `address_to_evict_from_cache` with the data supplied by the `data_to_evict_from_cache` signal. In addition, the number of memory accesses is increased.

3.4.6 Modify_Write_Hit Process

The `modify_write_hit` process is sensitive to the `write_hit_address_modify_from_cache` and `write_hit_data_modify_from_cache` signals. This process assumes that a write-once cache is used, and that a cache block that was previously read into the cache has been updated for the first time, thus causing a write-hit in the cache. Since the first write in a write-once cache is reported to main memory, the address and data modified in the cache must be supplied to main memory so that main memory may be kept consistent with the cache. This process modifies the block in main memory indicated by the `write_hit_address_modify_from_cache` signal to contain the data indicated by the `write_hit_data_modify_from_cache` signal in order to preserve the coherency between main memory and the cache. In addition, the number of memory accesses is increased.

3.4.7 Modify_Write_Miss Process

The `modify_write_miss` process is sensitive to the `write_miss_address_modify_from_cache` and `write_miss_data_modify_from_cache` signals. Similar to the `modify_write_hit` process, this process is also used with the write-once cache implementations, but assumes that a write is being issued to a block that does not yet exist in the cache. The block must first be read into the cache using the `memory_access` process explained above, and then, since the first write of all cache blocks in a write-once cache is reported to main memory, the data contents of the block in main memory are modified.

This process modifies the contents of the block indicated by the `write_miss_address_modify_from_cache` signal to contain the data value indicated by the `write_miss_data_modify_from_cache` signal. In
addition, the number of memory accesses is increased by two since the first memory access to read the block into the write-once cache was not recorded, as discussed in the section describing the memory_access process on page 24.

3.4.8 Dump Process
The dump process is sensitive to the memory_dump signal, controlled by the processor, and is used to write the contents of main memory to a file for analysis purposes. During the execution of test-bench programs, frequent cache dumps are required in order to compare the execution of the test-bench program on various cache architectures.

3.5 VHDL Implementation of the FIFO Replacement Algorithm
The FIFO replacement algorithm means that the first block brought into the cache is the first one to be evicted, regardless of how recent the block was referenced. The use of this replacement algorithm may suffer from poor performance factors since the temporal locality property of the memory hierarchy states that recently referenced items are likely to be referenced again in the near future. By evicting a block that was recently used, the cache performance may be degraded under the assumptions implied by the temporal locality principle, since after the block was evicted from the cache on instruction N, it will need to be read back into the cache on instruction N+1, thus increasing the number of memory accesses, and degrading the performance.

To implement the FIFO replacement algorithm, any time a block is brought into the cache, its address is added to the tail of the FIFO queue. When the queue gets full, it is implied that the cache is also full since the FIFO queue is the same size as the cache. When a new block is brought into the full cache, there is no room to add it to the cache, nor the FIFO queue, so the block corresponding to the address located at the head of the FIFO queue is written over by the new data. The FIFO queue head is then advanced one level, and the new address added to the cache becomes the tail of the queue.

3.6 VHDL Implementation of the LRU Replacement Algorithm
The LRU replacement algorithm means that the block that has not been used in the longest time is evicted from the cache. The use of this replacement algorithm seems to be an improvement over the use of the FIFO algorithm presented in the previous section since the temporal locality property of the memory hierarchy is satisfied. By removing the block that has not been used in the longest period of time and retaining those that have, the more recently used blocks remain in the cache while those that have not been used lately are removed to make room for new blocks. By evicting a block that has not been used in
Meanwhile, the cache performance may improve since now, under the assumptions implied by the temporal locality principle, the blocks that are more likely to be used in the near future now reside in the cache. In addition, the chances that the block just evicted will be used again is very small, again under the assumptions implied by the temporal locality principle.

To implement the LRU replacement algorithm, any time a block is brought into the cache, its address is added to the end of the LRU array. Addresses at the beginning of the LRU array are the least recently used addresses, and the address at the end of the array is the one that was most recently used. When the array gets full, it is implied that the cache is also full since the LRU array is the same size as the cache. When a new block is brought into the full cache, there is no room to add it to the cache, nor the LRU array, so the block corresponding to the address located at the beginning of the LRU array is written over by the new data. The contents of the LRU array are then pushed to the front of the array one level, and the new address is added to the end of the LRU array to become the most recently used address.

### 3.7 VHDL Implementation of the Cache Architectures

The implementation of all of the cache architectures should be capable of handling read and write operations requested by the processor, as well as references to data that does not exist in the cache. In order to evaluate the affects of different cache mapping strategies, the cache architectures were implemented in three different ways: direct mapped, fully-associative mapped, and set-associative mapped. Also, in order to analyze the affect of the replacement strategy on the cache implementations, within the fully associative mapped implementations, two different replacement strategies were implemented: First-In-First-Out (FIFO) and Least-Recently-Used (LRU).

#### 3.7.1 Description of the Constants and Types Used Throughout the VHDL Implementation of a Cache

To aide in the VHDL implementation of a cache architecture, the WRITE_THROUGH_CACHE_TYPE, WRITE_BACK_CACHE_TYPE, and WRITE_ONCE_CACHE_TYPE packages were created and are instantiated in files that describe the implementation of one of the cache architectures. Only the appropriate package needs to be instantiated with the appropriate cache type (i.e. the WRITE_THROUGH_CACHE_TYPE package is instantiated with files describing or using the implementation of a write-through cache). All of the packages contain essentially the same constants and types. the only difference being the number of status bits used. These signals, sub-types, and types are described in Table 10.
<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE_SIZE</td>
<td>the cache size in units of kilobytes</td>
<td>Integer</td>
<td>Varies</td>
</tr>
<tr>
<td>BLOCKS_PER_SET</td>
<td>the number of blocks per set, or the degree of associativity</td>
<td>Integer</td>
<td>Varies</td>
</tr>
<tr>
<td>NUMBER_OF_SETS</td>
<td>the number of sets in the cache</td>
<td>Integer</td>
<td>$\frac{\log_2(CACHE_SIZE) + 10 - 3}{BLOCKS_PER_SET}$</td>
</tr>
<tr>
<td>CACHE_BLOCK_STATUS_BITS</td>
<td>the number of status bits to use with the cache architecture</td>
<td>Integer</td>
<td>0 for write-through cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 for write-back cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 for write-once cache</td>
</tr>
<tr>
<td>FULLY_CACHE_BLOCK_ADDRESS_BITS</td>
<td>the number of address bits to use in a fully-associative mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_ADDRESS_BITS</td>
</tr>
<tr>
<td>DIRECT_CACHE BLOCK_ADDRESS_BITS</td>
<td>the number of address bits to use in a direct-mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_ADDRESS_BITS - (log_2(CACHE_SIZE) + 10 - 3)</td>
</tr>
<tr>
<td>SET_CACHE BLOCK_ADDRESS_BITS</td>
<td>the number of address bits to use in a set-associative mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_ADDRESS_BITS - log_2(NUMBER_OF_SETS)</td>
</tr>
<tr>
<td>CACHE_BLOCK_DATA_BITS</td>
<td>the number of bits used for data</td>
<td>Integer</td>
<td>MEMORY_BLOCK_DATA_BITS</td>
</tr>
<tr>
<td>FULLY_CACHE_BLOCK_SIZE</td>
<td>the size of a block in a fully-associative mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_DATA_BITS + FULLY_CACHE_BLOCK_ADDRESS_BITS + CACHE_BLOCK_STATUS_BITS</td>
</tr>
<tr>
<td>DIRECT_CACHE_BLOCK_SIZE</td>
<td>the size of a block in a direct-mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_DATA_BITS + DIRECT_CACHE_BLOCK_ADDRESS_BITS + CACHE_BLOCK_STATUS_BITS</td>
</tr>
<tr>
<td>SET_CACHE_BLOCK_SIZE</td>
<td>the size of a block in a set-associative mapped cache</td>
<td>Integer</td>
<td>MEMORY_BLOCK_DATA_BITS + SET_CACHE_BLOCK_ADDRESS_BITS + CACHE_BLOCK_STATUS_BITS</td>
</tr>
<tr>
<td>fully_cache_block</td>
<td>sub-type used to describe a cache block in a fully-associative mapped cache</td>
<td>Standard logic vector</td>
<td>Range from (FULLY_CACHE_BLOCK_SIZE - 1 downto 0)</td>
</tr>
<tr>
<td>direct_cache_block</td>
<td>sub-type used to describe a cache block in a direct-mapped cache</td>
<td>Standard logic vector</td>
<td>Range from (DIRECT_CACHE_BLOCK_SIZE - 1 downto 0)</td>
</tr>
<tr>
<td>set_cache_block</td>
<td>sub-type used to describe a cache block in a set-associative mapped cache</td>
<td>Standard logic vector</td>
<td>Range from (SET_CACHE_BLOCK_SIZE - 1 downto 0)</td>
</tr>
<tr>
<td>queue_entry</td>
<td>sub-type used to describe an entry in either the LRU array or FIFO queue</td>
<td>Integer</td>
<td>Varies</td>
</tr>
</tbody>
</table>
Table 10. Description of the Constants and Types Used in the VHDL Implementation of a Cache

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Array Type</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>type used to describe the LRU array or FIFO queue</td>
<td>Array of integers</td>
<td>(2^{\log_{base_2}(CACHE_SIZE)+10^3} - 1) downto 0</td>
</tr>
<tr>
<td>fully_cache_unit</td>
<td>type used to describe a fully-associative mapped cache</td>
<td>Array of fully_cache_blocks</td>
<td>(2^{\log_{base_2}(CACHE_SIZE)+10^3} - 1) downto 0</td>
</tr>
<tr>
<td>direct_cache_unit</td>
<td>type used to describe a direct-mapped cache</td>
<td>Array of direct_cache_blocks</td>
<td>(2^{\log_{base_2}(CACHE_SIZE)+10^3} - 1) downto 0</td>
</tr>
<tr>
<td>set_cache_unit</td>
<td>type used to describe a set-associative mapped cache</td>
<td>Array of set_cache_blocks</td>
<td>(2^{\log_{base_2}(CACHE_SIZE)+10^3} - 1) downto 0</td>
</tr>
</tbody>
</table>

The cache size and number of blocks per set are non-negative integers than can be modified within this package. It is important that the number of blocks per set be a divisor of the cache size so that all sets in a set-associative cache will have the same number of blocks, and so that the cache will be divided up evenly among these sets. Modification of these values affect all files that have this package instantiated in it, provided that the above mentioned files are recompiled prior to their simulation.

The number of sets used within the cache is found by performing the non-negative integer division between the cache size and the number of blocks per set. The result, a non-negative integer, dictates the number of sets that the cache can be divided into, under the restrictions of the cache size and the number of blocks per set.

No status bits are required for the write-through cache since all writes to data blocks in the write-through are reported to main memory. Therefore, data blocks in the write-through cache are never inconsistent with those in main memory, and all blocks in the write-through cache are valid. Thus, when the cache is full an a new block must be brought in, the block to be “evicted” can simply be written over without reporting the “evicted” block’s data contents to main memory.

The number of status bits used in a write-back cache should only be one, since there are only two states that a block in a write-back cache can be in: clean (consistent with the main memory) or dirty (inconsistent with main memory). In this thesis, only one status bit is used within the write-back cache implementations. A value of one indicates a clean block while a zero in the status bit indicates a dirty block.
The number of status bits used in a write-once cache should be two, since there are four states that a block in a write-once cache can be in: valid (consistent with main memory), reserved (block has been written to only once while it has been in the cache, and the modification has been reported to main memory), dirty (the block has been written to more than once since it has been brought into the cache, and hence its contents are inconsistent with those of main memory), and invalid. Throughout this thesis, two status bits are used to distinguish among the four states described above, with the bit patterns 11, 10, 01, and 00 used to denote the valid, reserved, dirty, and invalid states respectfully. The invalid state is generally not used within this thesis, however, with three mandatory states, two bits are required and hence the fourth bit pattern, 00, must be given a name.

The number of bits used for cache block addressing varies among the cache architectures. In the fully-associative mapped cache, a block could be placed anywhere. Therefore, there is no implicit information that can be assumed about a block in the cache. Thus, the number of address bits required in a fully-associative cache is the same as the number of address bits needed to reference the block in main memory.

In a direct-mapped cache, there is only one place that a block can reside in the cache. When the data residing in a certain address block is requested by the processor, a direct-mapped cache divides the requested address by the number of blocks in the cache. The remainder of this integer division denotes the position that the requested address block should reside in the cache, if it has been previously read into it. This number is then used as an array index to retrieve the information from the cache.

What distinguishes between two addresses that could be mapped to the same position in a direct-mapped cache is the quotient of the integer division performed above—between the address and the number of blocks the direct-mapped cache can hold. Therefore, only this part of the address needs to be stored in the cache since the rest of the address is inferred by the block’s position in the cache. Thus, the formula for the number of bits required for addressing in a direct-mapped cache is the number of address bits needed to address a block in main memory (MEMORY_BLOCK_ADDRESS_BITS), minus the number of bits required to distinguish among the blocks in the cache (log_base_2(CACHE_SIZE)+10), minus three bits which are used to choose one of the eight sub-blocks that the data portion of the cache block is divided into. The equation is written symbolically as Equation 2.

\[
\text{MEMORY BLOCK ADDRESS BITS} - (\log_{2}(\text{CACHE SIZE})+10-3)
\]

Equation 2. Address Bits Needed in a Direct-Mapped Cache
In a set-associative cache, the cache is divided up into sets, and a block that is mapped into that set could reside anywhere within it. Therefore, when the data from an address block is requested by the processor, this address must be divided by the number of sets that are in the cache. The remainder of this integer division indicates the set that the requested block could reside in. This integer is used as an index in the cache array in order to find the starting block for the appropriate set in the cache. Each block in the set must be investigated to see if its address portion matches that of the address requested since the requested block could reside anywhere within the set.

What distinguishes among the many blocks that could be mapped to the same set, is the quotient part of the division performed above—between the address and the number of sets the cache has been divided into. Hence, only the quotient which distinguishes among the many addresses that could be stored within the same set needs to be stored with the data portion of the block in the cache. The number of bits required for set-associative addressing is thus calculated as the number of bits required to reference the address block in main memory (MEMORY_BLOCK_ADDRESS_BITS), minus the number of bits required to distinguish among the sets that the cache has been divided into \((\log_{2}(\text{NUMBER_OF_SETS}))\). Symbolically, this formula is written as Equation 3.

\[
\text{MEMORY_BLOCK_ADDRESS_BITS} - \log_{2}(\text{NUMBER_OF_SETS})
\]

Equation 3. Address Bits Needed in the Set-Associative Mapped Cache

The number of data bits used in the cache is equivalent to the number of data bits used within the blocks of main memory. Therefore, in this thesis, whenever a block is transferred from main memory to the cache, all 64 bits are written into the cache, and thus all 64 bits of the data portion of the block are available to the program being executed on the processor. It is up to the designer of the program to be executed on the processor to extract a particular byte from the eight sub-blocks available from the 64 bits returned.

The cache block size is determined by the number of address bits, the number of data bits, and the number of status bits, where necessary. Since the number of address bits differs among the three different mapping schemes, and since only the write-back and write-once cache architectures use status bits, the size of a cache block will differ among the cache architectures and among the different mapping strategies, as shown in Table 10.
3.7.2 Description of the VHDL Implementation of a Write-Through Cache Using No-Write-Allocate on a Write-Miss

The implementations of the write-through cache architectures are described below. The general performance of the write-through cache implementations, however, are the same. On a read, if the requested block does not exist within the cache, the block is read into the cache from main memory, and its information is accessed. On a write, however, if the requested block does not exist in the cache, the block is not brought into main memory prior to its modification. Instead, the new data is written directly to main memory, since all write-misses to blocks not within the cache are reported directly to main memory in a write-through environment. This method is referred to as no-write-allocate and is used in order to decrease the number of memory accesses required on a cache miss. If its opposite, a write-allocate scheme, was used instead, the requested block would first be read in (one memory access) and then modified. Since all writes in a write-through cache are sent to main memory, another memory access (the second one) will occur in order to update main memory with the new data value.

With the no-write-allocate scheme, only one memory access is required in order to send the new data to the corresponding address in main memory. The second memory access (the one due to bringing the block into the cache) does not occur unless the block is needed to satisfy a read-miss in the cache. If this read-miss never happens, time is saved by never having brought the block into the cache.

There are four different write-through cache implementations investigated in this thesis: direct-mapped, fully-associative mapped using FIFO replacement, fully-associative mapped using LRU replacement, and set-associative mapped. Most of the processes used among these four implementations are the same, however, there were some differences. Each of the processes used in the implementation, in addition the signals used in the communication between the cache an main memory, are described below, and the differences in each process among the four different write-through cache implementations are highlighted.

3.7.2.1 Description of the Signals Used Within the Write-Through Cache Implementations

To implement a write-through cache in VHDL, several signals were used to communicate information to either the processor or main memory. Since the processor can only handle one request at a time, the cache will either be in the read mode or the write mode at any specific instant in time. Therefore, only a certain number of the signals described below will actually be in use at any given instant in time.
• **read_not_write**: an input signal indicating whether a read or a write occurred in the program executed by the processor. The signal is one to indicate a read and zero to indicate a write. This signal is a standard logic bit.

• **address_in_cache**: an input signal indicating the address referenced by the processor. This signal is an integer.

• **data_in_cache**: an input signal indicating the data associated with the address specified by the address_in_cache signal. This signal is a standard logic vector of length 64 bits.

• **data_in_from_mem**: an input signal that receives the data contents from main memory of the corresponding block in main memory that is indicated by the address_in_cache signal. An event on this signal implies that a read-miss occurred in the cache. This signal is a standard logic vector of length 64 bits.

• **cache_dump**: an input signal controlled by the processor that requests that the contents of the cache, and the FIFO queue or LRU array where applicable, be written, or "dumped," to a file. This signal is a standard logic bit, and a value of one indicates that the contents of the cache, and also the contents of either the FIFO queue or LRU array, be written to an output file for analysis.

• **data_read_out_cache**: an output signal that returns the requested data contents to the processor. This signal is a standard logic vector of length 64 bits.

• **address_needed_in_cache**: an output signal that is sent to main memory which contains the address that was referenced by the processor, but does not exist in the cache. An event on this signal indicates that a miss occurred in the cache, and the required block must be brought into the cache from main memory. This signal is an integer.

• **data_to_write_from_cache**: an output signal that is sent to main memory which contains the data to write to a block in main memory when an update must be performed. This update will either be a result of a write in the cache, or of a write-miss in the cache. This signal is a standard logic vector of length 64 bits.

• **read_miss_out**: an output signal that is sent to main memory which indicates that a read-miss occurred in the cache. This signal is a standard logic bit, where a value of one indicates that a read-miss occurred in the cache.

• **write_miss_out**: an output signal that is sent to main memory which indicates that a write-miss occurred in the cache. This signal is a standard logic bit, where a value of one indicates that a write-miss occurred in the cache.
3.7.2.2 Read_or_Write Process

The read_or_write process is sensitive to the read_not_write, address_in_cache, and data_in_cache signals, all of which are controlled by the processor. This process is used to perform either a read or a write of a block in the cache, whichever operation is indicated by the read_not_write signal.

In the direct-mapped cache, the appropriate index in the cache is found first. This is performed by computing the non-negative integer equivalent of the address supplied in the address_in_cache signal and dividing this non-negative integer by the size of the cache. This final result is the position in the cache where the desired block should reside. If the address requested corresponds to the integer equivalent of the address portion of this block multiplied by the number of blocks in the cache, and added to the index, the information is returned to the processor via the data_read_out_cache signal if a read was issued, or modified and reported to main memory if a write was issued. The equation used to compare the addresses is shown symbolically in Equation 4.

\[ \text{lv_to_integer(Address portion)\times(2\times(log_{base\_2}(\text{CACHE\_SIZE})+10-3))}\times\text{position of block in cache} \]

Equation 4. Converting the Address Portion of a Cache Block in a Direct-Mapped Cache to Compare to the Requested Address

If the block is not found, a read- or write-miss occurs, and is indicated by the internal read_miss and write_miss signals.

In a fully-associative cache, the entire contents of the cache must be searched, since the referenced block may exist anywhere. If the addressed block is found, its data value is read and returned to the processor via the data_read_out_cache signal if a read was issued, or modified and reported to main memory if a write was issued. In addition, in the fully-associative mapped cache implementation using the LRU replacement algorithm, the corresponding address is removed from its previous location in the LRU array and placed at the end to become the new most recently used address. All other addresses above this position in the LRU array are then pushed down one level to become least recently used. If the block is not found in the cache, a read- or write-miss occurs, and this miss is indicated by the internal read_miss and write_miss signals.

In the set-associative cache, the appropriate set must be accessed and searched to see if it contains the requested address. The appropriate set is found by taking the address supplied in the address_in_cache signal and dividing this non-negative integer by the number of sets in the cache. This final result is used to find the set in the cache where the desired block should reside. The entire contents of this set are searched, starting at the index calculated above, and continuing BLOCKS_PER_SET-1 blocks in the cache. For example, if there are four blocks in a set, and the address referenced maps to the first set, blocks zero
through three will be checked in the cache to see if their address matches that indicated by the address_in_cache signal.

If an address portion of a block in this set multiplied by the number of sets in the cache (NUMBER_OF_SETS), and added to the index corresponds to that of the address expressed in the address_in_cache signal, the information is returned to the processor via the data_read_out_cache signal if a read was issued, or modified and reported to main memory if a write was issued. The formula used to convert the address portion of the cache block to use in comparing the two addresses is shown symbolically in Equation 5.

\[ lv\_to\_integer(\text{Address portion}) \times \text{NUMBER\_OF\_SETS} + \text{position of block in the cache} \]

**Equation 5. Converting the Address Portion of a Block in a Set-Associative Cache to Compare to the Requested Address**

If the block is not found, a read- or write-miss occurs, and this miss is indicated by the internal read_miss and write_miss signals.

### 3.7.2.3 Read_Write_Miss Process

The read_write_miss process is sensitive to the address_in_cache, data_in_cache, and read_not_write signals, which are controlled by the processor. This process is responsible for reporting a read-miss, write-miss, or an update to main memory. If the internal read_miss signal is a one, the write_miss signal is a zero, and the read_not_write signal is a one, a read-miss occurred in the cache, and the needed block must be brought into the cache. This read-miss occurrence is reported to main memory by placing the referenced address on the address_needed_in_cache output signal, and indicating a read-miss to main memory by making the output read_miss_out signal a one and the write_miss_out signal a zero. The toggling of these signals will cause main memory to return the data contents of the corresponding block.

If the internal read_miss signal is a zero, the write_miss signal is a one, and the read_not_write signal a zero, a write-miss occurred in the cache. Since a no-write-allocate scheme is used, the needed block is not brought into the cache, however, the data modification must still be reported to main memory. This is done by placing the corresponding address on the address_needed_in_cache signal, the data to write on the data_to_write_from_cache signal, and indicating a write-miss to main memory by placing a value of zero on the output read_miss_out signal and a value of one on the output write_miss_out signal. The toggling
of these signals will cause main memory to modify the data portion of the indicated address with that indicated by the data_to_write_from_cache signal.

If the internal read_miss and write_miss signals are both a zero value, and the read_not_write signal is also a zero, a write-hit occurred in the cache. Since all writes in a write-through cache are immediately reported to main memory, the address modified and the new data value must be relayed to main memory. This is done by placing the address modified on the address_needed_in_cache output signal line, the new data contents on the data_to_write_from_cache signal line, and indicating that a hit occurred in the cache by placing a zero value on both the read_miss_out and write_miss_out output signal lines. The toggling of these signals will cause main memory to modify the data contents corresponding the address indicated by the address_needed_in_cache signal.

### 3.7.2.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes

The direct_mapped_add, fully_associaive_add, and set_associative_add processes are sensitive to the read_not_write, address_in_cache, and data_in_cache signals controlled by the processor, and the data_in_from_mem signal which receives its information from main memory. The only time a block is added to a write-through cache is on a read-miss. In all other cases (i.e. write-misses and all hits) the referenced block is either in the cache, as in a read- or write- hit, or solely accessed within main memory, as in a write-miss. Therefore, if the internal read_miss signal has a value of one, the internal write_miss signal has a value of zero, and data has arrived from main memory, indicated by an event occurring on the data_in_from_mem signal, a block is found in the cache with which to store the incoming data. In the direct-mapped cache, the appropriate index within the cache is found, and both the address and data portions of this block are overwritten.

In a fully-associative mapped cache, the first free block discovered, indicated by the first bit (and in fact all of the bits) in the block being undefined, is used to store the new information added to the cache. If a free block is found, the new address and data are written to this location, and the address is added to the tail of the FIFO queue, or end of the LRU array.

If no free blocks are discovered in the fully-associative mapped cache, the head of the FIFO queue or the first entry in the LRU array is read and assigned to the internal variable address_to_evict. The contents of the cache are then searched until the block corresponding to this address is found. Once found, this block is overwritten with the new address and data supplied by the address_in_cache and data_in_from_mem signals.

36
signals, respectively. The new address is then added to the tail of the FIFO queue, and the head of the FIFO queue is advanced one level if the FIFO replacement algorithm is used. If the LRU replacement algorithm is used instead, the address evicted is removed from the first position in the LRU array, all other addresses are pushed down one level to become least recently used, and the new address is added to the end of the LRU array to become the new most recently used address.

In the set-associative cache, the first free block discovered within the set is used to store the incoming data. If no free blocks are discovered in the set, the first block in the set is always replaced with the incoming information. This is a pseudo-random replacement algorithm that is used with the set-associative mapped cache implementation since, keeping track of FIFO queues or LRU arrays for each set will be difficult to do.

Recall, that a read-miss is the only event that requires a block to be brought into a write-through cache. Read- and write-hits access the data directly from the cache, with the modified data being reported to main memory in the event of a write-hit. A write-miss requires that the new data be written directly to main memory, without the block first being brought into the cache, since a no-write-allocate scheme is used. In addition, overwriting the contents of a block in the cache does not affect coherency issues or loss of data, since in a write-through cache, the cache and main memory are always consistent with each other.

### 3.7.2.5 Dump Process

The dump process is sensitive to the cache_dump signal, controlled by the processor, and is used to write the contents of the cache, and the FIFO queue or LRU array in the fully-associative write-through cache implementations, to a file for analysis purposes. During the execution of test-bench programs, frequent cache dumps are required in order to compare the execution of the test-bench program on various cache architectures.

### 3.7.3 Description of the VHDL Implementation of a Write-Back Cache Using Write-Allocate on a Write-Miss

The implementation of a write-back cache architecture is described below. On a read, if the requested block does not exist within the cache, the block is read into the cache and its information is accessed. On a write, if the requested block does not exist in the cache, the block is brought into the cache and then modified without notifying main memory. This modification will be reported to main memory when, and
if, the block is evicted from the cache. Hence, a memory access is saved, however, the coherency between the cache and main memory is jeopardized.

This method is referred to as a write-allocate scheme and is generally used with write-back caches, since in a write back cache, modifications in the cache are not reported to main memory until after the block has been evicted from the cache. The same strategy is used in the write-allocate scheme; the needed block is brought into the cache and modified, however, the modifications are not reported to main memory until after the cache block is evicted.

This method differs from the no-write-allocate scheme used in the write-through cache since in the no-write-allocate scheme, coherency is preserved at the possible expense of a second memory access to bring the block into the cache when it is accessed again in the near future— an appropriate assumption to make based on the implications of the principle of temporal locality. In the write-allocate scheme, coherency is sacrificed in order to reduce the number of accesses to main memory. This tradeoff will be explored more in the remainder of the thesis as test-benches are used to analyze the different cache strategies.

3.7.3.1 Description of the Signals Used Within the Write-Back Cache Implementations

To implement a write-back cache in VHDL, several signals were used to communicate information to either the processor or main memory. Since the processor can only handle one request at a time, the cache will either be in the read mode or the write mode at any specific instant in time. Therefore, only a certain number of the signals used in the VHDL implementations will be used at any given instant of time.

The signals used to communicate information between main memory and the write-back cache are the same as those used in the write-through cache implementation, however, the write-back cache uses one additional signal. The address_to_evict signal is an output signal sent to main memory which contains the address of a dirty block that was evicted from the cache. An event on this signal indicates that a dirty block was evicted from the cache, and its data contents, indicated by the data_to_write_from_cache signal, must be written to the corresponding block in main memory. This signal is an integer.
3.7.3.2 Read_or_Write Process

The read_or_write process is sensitive to the read_not_write, address_in_cache, and data_in_cache signals, all of which are controlled by the processor. This process is used to perform either a read or a write of a block in the cache, whichever operation is indicated by the read_not_write signal.

In the direct-mapped cache, the appropriate index in the cache is found. This is performed by computing the non-negative integer equivalent of the address supplied in the address_in_cache signal and dividing this non-negative integer by the size of the cache. This final result is the position in the cache where the desired block should reside. If the address requested corresponds to the integer equivalent of the address portion of this block multiplied by the number of blocks in the cache, and added to the index, the information is returned to the processor via the data_read_out_cache signal if a read was issued, or modified if a write was issued. This formula was shown in Equation 4. In addition, the block now becomes dirty, if it was not already, since it has been modified while it was in the cache, and its contents are no longer consistent with those in main memory. The dirty status of the block is indicated by making the status bit for the block a zero. If the block is not found, a read- or write-miss occurs, and is indicated by the internal read-miss and write-miss signals.

In a fully-associative cache, the entire contents of the cache must be searched, since the referenced block may exist anywhere. If the addressed block is found, its data value is read and returned to the processor via the data_read_out_cache signal if a read was issued, or modified if a write was issued. In addition, in the fully-associative mapped cache implementation using the LRU replacement algorithm, the corresponding address is removed from its previous location in the LRU array and placed at the end to become the new most recently used address. All other addresses in the LRU array are then pushed down one level to become least recently used. Also, the status bit of the cache block must be changed to a zero, to mark a dirty block, if a write was issued by the processor. If the block is not found, a read- or write-miss occurs, and is indicated by the internal read_miss and write_miss signals.

In the set-associative cache, the appropriate set must be accessed and searched to see if it contains the requested address. The appropriate set is found by taking the address supplied in the address_in_cache signal and dividing this non-negative integer by the number of sets in the cache. This final result is used to find the set in the cache where the desired block should reside. The entire contents of this set are searched, starting at the index calculated above, and continuing BLOCKS_PER_SET-1 blocks in the cache. For example, if there are four blocks in a set, and the address referenced maps to the first set, blocks zero through three will be checked in the cache to see if their address matches that indicated by the address_in_cache signal.
If an address portion of a block in this set multiplied by the number of sets in the cache (NUMBER_OF_SETS) and added to the index corresponds to that of the address expressed in the address_in_cache signal, the information is returned to the processor via the data_read_out_cache signal if a read was issued, or modified if a write was issued. This formula was shown in Equation 5. In addition, the block must be marked as dirty if a write was issued by the processor. If the block is not found, a read-or write-miss occurs, and is indicated by the internal read_miss and write_miss signals.

3.7.3.3 Read_Write_Miss Process
The read_write_miss process is sensitive to the address_in_cache, data_in_cache, and read_not_write signals, which are controlled by the processor. This process is responsible for reporting a read-miss or write-miss to main memory. If the internal read_miss signal is a one, the write_miss signal is a zero, and the read_not_write signal is a one, a read-miss occurred in the cache, and the needed block must be brought into the cache. This read-miss occurrence is reported to main memory by placing the referenced address on the address_needed_in_cache output signal, and indicating a read-miss to main memory by making the output read_miss_out signal a one and the write_miss_out signal a zero. The toggling of these signals will cause main memory to return the data contents of the corresponding block.

If the internal read_miss signal is a zero, the write_miss signal is a one, and the read_not_write signal a zero, a write-miss occurred in the cache. Since a write-allocate scheme is used, the needed block is brought into the cache and the data is modified in the cache, without informing main memory of the change to the data contents. This is done by placing the corresponding address on the address_needed_in_cache signal, and indicating a write-miss to main memory by placing a value of zero on the output read_miss_out signal and a value of one on the output write_miss_out signal. The toggling of these signals will cause main memory to modify the data portion of the indicated address with that indicated by the data_to_write_from_cache signal.

3.7.3.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes
The direct_mapped_add, fully_associative_add, and set_associative_add processes are sensitive to the read_not_write, address_in_cache, and data_in_cache signals controlled by the processor, and the data_in_from_mem, which receives its information from main memory. If the internal read_miss signal has a value of one, the internal write_miss signal has a value of zero, and data has arrived from main memory, indicated by an event occurring on the data_in_from_mem signal, a block is found in the cache...
with which to store the incoming data. In the direct-mapped cache, the appropriate index within the cache is found, and the status bit of this block is checked. If the block is clean, indicated by a status bit of one, or the block is unused, indicated by an undefined bit ('U'), both the address and data portions of this block are overwritten since the block contents are clean, and thus consistent with main memory. Thus, overwriting the contents of this block does not affect coherency issues or loss of data, since the cache and main memory are consistent with respect to this block. In addition, the status bit of the new block is made a one, since the block has just been read into the cache and not modified, hence its data contents match those in memory, and the block is clean.

If the block is dirty, indicated by a zero for the status bit, its contents must be reported to main memory so the coherency between main memory and the cache can be re-achieved with respect to this block. This update is performed by placing the evicted address on the address_to_evict output signal line, and its corresponding data on the data_to_write_from_cache output signal line. The new information is then written into this block, making the status bit a one, since the data contents of the block have just been read and not modified, hence its contents are the same as those in main memory and the block is clean.

In a fully-associative mapped cache, the first free block discovered, indicated by the first bit (and in fact all of the bits) in the block being undefined, is used to store the new block in the cache. If a free block is found, the new address and data are written to this location, the status bit is made a one to indicate a clean block, and the address is added to the tail of the FIFO queue if a FIFO replacement algorithm is used, or to the end of LRU array if the LRU replacement algorithm is used.

If no free blocks are discovered in the cache, the head of the FIFO queue or first entry of the LRU array is read and assigned to the internal variable address_to_evict. The contents of the cache are then searched until the block corresponding to this address is found. Once found, if this block is clean, it is overwritten with the new address and data supplied by the address_in_cache and data_in_from_mem signals, respectively, and the new cache block is marked as clean. The new address is then added to the tail of the FIFO queue and the head of the FIFO is advanced on level if the FIFO replacement algorithm is used. In the LRU replacement algorithm, the first entry in the LRU array is deleted, the remaining addresses in the array are pushed down one level to become least recently used, and the new address is added to the end of the array. If the evicted block is dirty, its contents are reported to main memory prior to the block being overwritten in the cache.

In a set-associative mapped cache, the first free block discovered within the set, indicated by the first bit (and in fact all of the bits) in the block being undefined is used as a free block. If a free block is found, the
new address and data are written to this location, with the status bit made a one since the block has just
been read into the cache, and not modified, hence its contents are clean with respect to main memory.

If no free blocks are discovered in the set, the first dirty block discovered within the set is evicted since the
coherency between main memory and the cache should be preserved as soon as possible. If no dirty blocks
exist in the set, the first block in the set is written over by the new information. This is a pseudo-random
replacement strategy that is used within the set associative cache implementations since managing separate
FIFO queues or LRU arrays for each set will be too difficult.

In a write-back cache architecture, a block is also added to the cache on a write-miss, since a write-allocate
scheme is used. A procedure similar to the one discussed above for a read-miss is followed for adding a
block to the cache on a write-miss, however, for a write-miss, the status bit of the new block is made a
zero, since the block is modified right away, and hence its contents differ from those in main memory.

3.7.3.5 Dump Process
The dump process is sensitive to the cache_dump signal, controlled by the processor, and is used to write
the contents of the cache, and FIFO queue or LRU array where applicable, to a file for analysis purposes.
During the execution of test-bench programs, frequent cache dumps are required in order to compare the
execution of the test-bench program on various cache architectures.

3.7.4 Description of the VHDL Implementation of a Write-Once Cache
The implementation of the write-once cache architectures are described below. On a read, if the requested
block does not exist within the cache, the block is read into the cache, its status bits are assigned the value
11 (valid) and its information is accessed. On a write, if the requested block does not exist in the cache, the
block is brought into the cache, modified, and the status bits are assigned the value 10 (reserved). Since this
modification is the first for this block in the cache, its data contents must be reported to main memory so
that coherency can be maintained between the cache and main memory. Recall, that in a write-once cache,
the first write to any block in the cache is reported to main memory with respect to this block. Main
memory does not receive notice of data modifications to this block due to subsequent writes in the
processor’s program until after the block has been evicted from the cache, if it is evicted at all.

This method differs from the no-write-allocate scheme used in the write-through cache since in the no-
write-allocate scheme, coherency is preserved at the possible expense of a second memory access to bring
the block into the cache when it is accessed again in the near future— an appropriate assumption to make based on the implications of the principle of temporal locality. The method used in the write-once cache also differs from the write-allocate scheme used in write-back caches, where coherency is sacrificed in order to reduce the number of accesses to main memory. The write-once cache seems to combine the benefit of added coherency from the no-write-allocate scheme by reporting the first write to main memory, with the benefit of fewer memory accesses of the write-allocate scheme by not reporting subsequent writes to main memory until the block has been evicted from the cache. The tradeoffs among these three schemes will be explored more in the remainder of the thesis as test-benches are used to analyze the different cache strategies.

3.7.4.1 Description of Signals Used Within the Implementation of a Write-Once Cache

To implement a write-once cache in VHDL, several signals were used to communicate information to either the processor or main memory. Since the processor can only handle one request at a time, the cache will either be in the read mode or the write mode at any specific instant in time. Therefore, only a certain number of the signals described below will actually be in use at any given instant in time. The signals used in the implementation of the write-once cache are the same as those used in the write-through and write-back caches, however, the following are exclusive to the write-once cache:

- **data_to_evict**: an output signal sent to main memory that contains the data corresponding to the evicted block whose address is **address_to_evict**. An event on this signal indicates that a dirty block was evicted from the cache. This signal is a standard logic vector of length 64 bits.

- **write_hit_address_modify**: an output signal sent to main memory that contains the address of a block that is being modified for the first time. In a write-once cache, the first write to any block in the cache is reported to main memory while all subsequent writes are not known to main memory until after the block has been evicted from the cache. An event on this signal implies that a block that was previously read into the cache has been modified. This means that a write-hit has occurred, and hence the name of this signal. This signal is an integer.

- **write_hit_data_modify**: an output signal sent to main memory that contains the data associated with the block addressed by the **write_hit_address_modify** signal. This signal is a standard logic vector of length 64 bits.

- **write_miss_address_modify**: an output signal sent to main memory that contains the address of a block that is being modified for the first time. An event on this signal implies that a block is being
brought into the cache and then being modified. This means that a write-miss has occurred, and hence the name of this signal. This signal is an integer.

- **write_miss_data_modify**: an output signal sent to main memory that contains the data associated with the block addressed by the write_miss_address_modify signal. This signal is a standard logic vector of length 64 bits.

### 3.7.4.2 Read_or_Write Process

The read_or_write process is sensitive to the read_not_write, address_in_cache, and data_in_cache signals, all of which are controlled by the processor. This process is used to perform either a read or a write of a block in the cache, whichever operation is indicated by the read_not_write signal.

In a direct-mapped cache, the appropriate index in the cache is found. This is performed by computing the non-negative integer equivalent of the address supplied in the address_in_cache signal and dividing this non-negative integer by the size of the cache. This final result is the position in the cache where the desired block should reside. If the address in this desired block multiplied by the number of blocks in the cache and added to the index calculated above corresponds to that of the address expressed in the address_in_cache signal, the information is returned to the processor via the data_read_out_cache if a read was issued, or modified if a write was issued. This formula was shown in Equation 4. In addition, if the block status was previously valid (11), it now becomes reserved (10), and the new data contents must be reported to main memory. If the block was previously reserved (10), it becomes dirty (01), however, the new data is not reported to main memory. If the block was already dirty (01), it remain dirty. If the block was written into a previously unused block, its status bits are made 11. If the block is not found in the cache, a read- or write-miss occurs, and is indicated by the internal read_miss and write_miss signals.

In a fully-associative cache, the referenced block may exist anywhere in the cache. If the addressed block is found, its data value is accessed and returned to the processor via the data_read_out_cache signal if a read was issued, or modified if a write was issued. On a write, if the block status was previously valid (11), it now becomes reserved (10), and the new data contents must be reported to main memory. If the block was previously reserved (10), it becomes dirty (01), however, the new data is not reported to main memory. If the block was already dirty (01), it remains dirty and only the data portion of the cache block is modified. In addition, in the fully-associative cache using the LRU replacement algorithm, the corresponding address is removed from its previous location in the LRU array and placed at the end to become the new most recently used address. All addresses above this position in the LRU array are then pushed down one level to become least recently used. If the block is not found in the cache, a read-miss
occurs, and is indicated by placing a one on the internal read_miss signal and a zero on the internal write_miss signal.

In the set-associative cache, the appropriate set must be accessed and searched to see if it contains the requested address. The appropriate set is found by taking the address supplied in the address_in_cache signal and dividing this non-negative integer by the number of sets in the cache. This final result is used to find the set in the cache where the desired block should reside. The entire contents of this set are searched, starting at the index calculated above, and continuing BLOCKS_PER_SET-1 blocks in the cache. For example, if there are four blocks in a set, and the address referenced maps to the first set, blocks zero through three will be checked in the cache to see if their address matches that indicated by the address_in_cache signal.

If an address portion of a block in this set multiplied by the number of sets in the cache (NUMBER_OF_SETS) and added to the index calculated above corresponds to that of the address expressed in the address_in_cache signal, the information is returned to the processor via the data_read_out_cache signal if a read was issued, or modified if a write was issued. This formula was shown in Equation 5. On a write, if the block was valid (11), the block now becomes reserved (10), and the data modification is reported to main memory. If the cache block was reserved (10), it now becomes dirty (01), although, the data modification is not reported to main memory until the block is evicted from the cache. A block that was previously dirty (01), remains dirty, and its data contents are overwritten by the new data in the data_in_cache signal. If the block is not found in the set, a read- or write-miss occurs, and is indicated by the internal read_miss and write_miss signals.

3.7.4.3 Read_Write_Miss Process

The read_write_miss process is sensitive to the address_in_cache, data_in_cache, and read_not_write signals, which are controlled by the processor. This process is responsible for reporting a read-miss or write-miss to main memory. If the internal read_miss signal is a one, the write_miss signal is a zero, and the read_not_write signal is a one, a read-miss occurred in the cache, and the needed block must be brought into the cache. This read-miss occurrence is reported to main memory by placing the referenced address on the address_needed_in_cache output signal, and indicating a read-miss to main memory by making the output read_miss_out signal a one and the write_miss_out signal a zero. The toggling of these signals will cause main memory to return the data contents of the corresponding block.
If the internal read_miss signal is a zero, the write_miss signal is a one, and the read_not_write signal a zero, a write-miss occurred in the cache. The needed block is brought into the cache and the data is modified in the cache. Since this write is the first to this block, its modified data value must be reported to main memory. This is done by placing the corresponding address on the address_needed_in_cache signal, the modified data on data_to_write_from_cache signal line, and indicating a write-miss to main memory by placing a value of zero on the output read_miss_out signal and a value of one on the output write_miss_out signal. The toggling of these signals will cause main memory to modify the data contents corresponding to the address indicated by the address_needed_in_cache signal.

3.7.4.4 Direct_Mapped_Add, Fully_Associative_Add, and Set_Associative_Add Processes

The direct_mapped_add, fully_associative_add, and set_associative_add process is sensitive to the read_not_write, address_in_cache, and data_in_cache signals controlled by the processor, and the data_in_from_mem signal, which receives its information from main memory. If the internal read_miss signal has a value of one, and the internal write_miss signal has a value of zero, a block is found within the cache in which to store the incoming data. In the direct-mapped cache, the appropriate index within the cache is found, and the status bits of the block residing there, if there is one, is checked. If the block is valid (11), reserved (10), or unused (all bits are 'U'), its contents match those in main memory, so both the address and data portions of this block are overwritten since the block contents are consistent with those of main memory, and overwriting the contents of this block does not affect coherency issues or loss of data. In addition, the status bits of the new block are made (11), to indicate a valid block, since the block has just been read into the cache and not modified.

If the block is dirty, indicated by the status bit pattern 01, its contents must be reported to main memory so the coherency between main memory and the cache can be re-achieved with respect to this block. This update is performed by placing the evicted address on the address_to_evict output signal line, and its corresponding data on the data_to_evict output signal line. The new data is then written into this block, making the status bits 11 to indicate a valid block, since the data contents of the block have just been read and not modified.

In a fully-associative mapped cache, the first free block discovered, indicated by the first bit (and in fact all of the bits) in the block being undefined, is used to store the new block in the cache. If a free block is found, the new address and data are written to this location, the status bits are made 11 to indicate a valid
block, and the address is added to the tail of the FIFO queue if the FIFO replacement algorithm is used, or the end of the LRU array if the LRU replacement algorithm is used.

If no free blocks are discovered in the cache, the head of the FIFO queue or the first entry of the LRU array is read and assigned to the internal variable `address_to_remove`. The contents of the cache are then searched until the block corresponding to this address is found. Once found, if this block is valid or reserved, it is overwritten with the new address and data supplied by the `address_in_cache` and `data_in_from_mem` signals, respectively, and the new cache block is marked as valid. The new address is then added to the tail of the FIFO queue and the head of the FIFO queue is advanced one level if the FIFO replacement algorithm is used. In the LRU replacement algorithm, the address evicted is removed from the first position in the LRU array, all other addresses in the LRU are pushed down one level to become least recently used, and the new address is added to the end of the array to become the new most recently used address. If the evicted block is dirty, its contents are reported to main memory prior to the block being overwritten in the cache.

In a set-associative cache, the first free block discovered within the set is used to store the new information. If a free block is found, the new address and data are written to this location, and the status bits are made 11 since the block has just been read into the cache, and not modified.

If no free blocks are discovered in the set, the first dirty block discovered within the set is evicted from the cache since the coherency between main memory and the cache should be re-established as soon as possible. If no dirty blocks are found within the set, the first reserved block discovered within the set is evicted from the cache. If no reserved blocks are found within the set, the first block in the set is evicted from the cache, and the new information is written into this location.

In a write-once cache, blocks are added to the cache on write-misses as well. An approach similar to the one used for read-misses is used to add blocks to the cache on a write-miss, however, the status bits of the new block are marked as 10, to indicate a reserved block, since the block has been modified once since it has been brought into the cache. In addition, since this is the first write to this block, its data modification must be reported to main memory.

### 3.7.4.5 Dump Process

The `dump` process is sensitive to the `cache_dump` signal, controlled by the processor, and is used to write the contents of the cache, and FIFO queue or LRU array where applicable, to a file for analysis purposes.
During the execution of test-bench programs, frequent cache dumps are required in order to compare the execution of the test-bench program on various cache architectures.
4 Cache and Processor Performance Parameters

The addition of cache memory is known to reduce the time needed to access the main memory, however, there are other parameters that the addition of cache memory affects. These parameters are discussed in this section.

4.1 CPU time

The CPU time is divided between the total number of clock cycles that the CPU spends executing its program and the time that is spent waiting for memory accesses to return with the necessary data, as shown in Equation 6.

\[
CPU\ time = (CPU\ execution\ cycles + Memory\ stall\ cycles) \times Clock\ cycle\ time
\]


In Equation 6, the number of memory stalls indicates the time that the CPU spends waiting for main memory to return with the necessary data. This occurs when either a read- or a write-miss occurs in the program that is being executed on the processor, as shown in Equation 7.

\[
Memory\ stall\ cycles = \frac{Reads}{Program} \times Read\ miss\ rate \times Read\ miss\ penalty
\]

\[
+ \frac{Writes}{Program} \times Write\ miss\ rate \times Write\ miss\ penalty
\]

Equation 7. Memory Stall Cycles per Read-Miss and Write-Miss Separately [Hennessy1996, p. 386]

Sometimes, designers assume that all memory stalls are due to the cache [Hennessey1990, p. 416]. This assumption may be true for most machines, however, not for all. For those where it isn’t true, the cache dominates the stalls that are not exclusively due to the cache, therefore it is safe to assume that all memory stalls are due to the cache.

In the memory stall cycle equation, the reads and the writes can be combined together to form an equation in terms of the number of memory accesses per program. This equation can be used to give a general
measurement of the miss rate per total number of memory accesses, instead of per read and write access separately, as shown in Equation 8.

\[
\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}
\]

Equation 8. Memory Stall Cycles per Memory Access [Hennessy1996, p. 386]

The instruction count (IC) can be factored out from the execution time and memory stall cycles to obtain a CPU time formula in terms of memory accesses per instruction, the miss rate, and the miss penalty, as shown in Equation 9.

\[
\text{CPU time} = IC \times \left( CPL_{\text{execution}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time}
\]

Equation 9. CPU Time in Terms of Memory Accesses per Instruction [Hennessy1996, p. 386]

Some designers prefer to measure the miss rate in terms of misses per instructions, as shown in Equation 10, instead of misses per memory reference [Hennessey1990, p. 417]. The advantage of using this formula is that it is independent of the hardware implementation, however, it is architecture dependent, and hence most designers use this formula when analyzing computers of the same family, such as the Motorola 68000.

\[
\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate}
\]

Equation 10. Miss Rate in Terms of Misses per Instruction [Hennessy1996, p. 386]

Using Equation 10, the final equation for the CPU time becomes

\[
\text{CPU time} = IC \times \left( CPL_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}
\]

Equation 11. CPU Time Using the Miss Rate in Terms of Misses per Instruction [Hennessy1996, p. 386]
4.2 **Average Memory Access Time**

The average memory access time is a measure of the hit time, the miss rate and the miss penalty, as shown in Equation 12.

\[
\text{Average memory access time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]


Equation 12 can also be broken down into the memory references due to instructions and those due to referencing data from the main memory, as shown in Equation 13.

\[
\text{Average memory access time} = \% \text{instructions} \times (\text{hit time} + \text{instruction miss rate} \times \text{miss penalty}) + \\
\% \text{data} \times (\text{hit time} + \text{data miss rate} \times \text{miss penalty})
\]

Equation 13. Average Memory Access Time Broken Down into Memory Accesses Due to Instruction and Memory Accesses Due to Data [Hennessy1996, p. 385]

Throughout this thesis, we will be concerned only with references to data, since instructions are rarely, if ever, modified while in the cache. By focusing on the behavior of the data values in the cache, more emphasis can be placed on the coherency issues, which is the purpose of this thesis.

### 4.3 Keeping Track of the Memory Accesses in the VHDL Implementations

In order to record the number of times main memory was accessed, a counter is kept in the memory unit implementation that counts the number of times the memory unit was accessed. The counter is incremented on a memory read, a memory write, a memory update from a write-through cache implementation, a memory update due to an evicted block, a memory update due to a write-hit in a write-once cache implementation, and a memory update due to a write-miss in a write-once cache implementation. The counter is printed for analysis in the dump process of each cache implementation.

### 4.4 Keeping Track of the Cache Performance

In order to calculate the performance of each cache implementation, counters are used to keep track of write-hits, write-misses, read-hits, and read-misses in each cache implementation. These parameters are
printed to the analysis file as part of the dump process of the cache implementations. Together with the number of memory accesses, these parameters will help evaluate the performance of each of the cache architectures implemented in this thesis.
In order to verify that the cache implementations were correctly modeled in VHDL, four test-bench programs were created, each one verifying a different function of a cache. Each test-bench program was executed on each cache implementation created in this thesis, and data was accumulated to analyze the performance of each test-bench program on each cache architecture. In the following sections, each test-bench program will be explained, and its performance on each of the cache architectures will be reported. For each test-bench, the size of main memory was 1 MB, the size of the cache was 512 KB, and there were four blocks in each set, thus dividing the cache into 16384 sets. Although the size of main memory is 1 MB, which is equivalent to $2^{20}$, or 1048576 bytes, eight bytes are combined to form one block consisting of 64 bits of data. Since each block consists of 64 data bits, or eight bytes, there are actually $\frac{2^{20}}{2^3} = 2^{17}$, or 131072, blocks in main memory.

Similarly, although the size of the cache is 512 KB, the size of the data portion must be equivalent to the block size of main memory. Therefore, if the memory block consists of eight bytes combined to form one 64-bit data block, the cache must also use 64-bit data blocks. Hence, there are $\frac{2^{19}}{2^3} = 2^{16}$, or 65536 blocks in the cache, each of which has a 64-bit data block that is divided into eight bytes. It is up to the program designer to extract the correct byte from the eight available bytes that are returned to him/her from referencing the data block.

Finally, although fully-associative caches are not very popular on the market due to their inferior performance to the other architectures, their implementations and simulations were performed in this thesis in order to compare them to those of the other architectures. It should be noted though that the time required to simulate a fully-associative cache is many times larger than the time required to simulate a direct-mapped cache or set-associative mapped cache. There were many fully-associative cache simulations that were left running for 72 hours or more on a Hewlett-Packard Visualize J282 UNIX Workstation, which uses a 180 MHz processor, and 1 GB of RAM.
5.1 The Read Test

The read test was designed to verify the read operation of the cache implementations. This test first initializes main memory, and then reads each block of main memory into the cache. The program uses the MEMORY_SIZE constant so that all the blocks in main memory will be read. Table 11 contains the data collected when the read test was executed on all of the cache implementations of this thesis.

<table>
<thead>
<tr>
<th>Cache Architecture</th>
<th>Memory Accesses</th>
<th>Read-Hits</th>
<th>Read-Misses</th>
<th>Write-Hits</th>
<th>Write-Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-Back Direct</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Back Full FIFO</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Back Full LRU</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Back Set</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Once Direct</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Once Full FIFO</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Once Full LRU</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Once Set</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Through Direct</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Through Full FIFO</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Through Full LRU</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Write-Through Set</td>
<td>131072</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 11. Data Analysis of the Read Test

As can be seen from Table 11, all of the cache implementations required 131072 memory accesses and 131072 read-misses since 131072 blocks are being read into the cache. Since each read is for a different block of main memory, each one results in a read-miss in the cache. There are no modifications made to the data contents of the cache block, so no additional memory accesses are needed when the block is evicted from the cache, because the cache and main memory are always coherent with each other.

The results obtained from this test verify that the read operation works for all of the cache implementations created in this thesis, and further verify that the counters for the memory and read-misses work correctly.

5.2 The Write Test

The write test was designed to verify the write operation of the cache implementations. This test first initializes main memory, and then modifies the data contents of each block of main memory. The program uses the MEMORY_SIZE constant so that all the blocks in main memory will be written to. Table 12 contains the data collected when the write test was executed on all of the cache implementations of this thesis.
### Table 12. Data Analysis of the Write Test

<table>
<thead>
<tr>
<th>Cache Architecture</th>
<th>Memory Accesses</th>
<th>Read-Hits</th>
<th>Read-Misses</th>
<th>Write-Hits</th>
<th>Write-Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-Back Direct</td>
<td>196608</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Back Full FIFO</td>
<td>196608</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Back Full LRU</td>
<td>196608</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Back Set</td>
<td>196608</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Once Direct</td>
<td>262144</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Once Full FIFO</td>
<td>262144</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Once Full LRU</td>
<td>262144</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Once Set</td>
<td>262144</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Through Direct</td>
<td>131072</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Through Full FIFO</td>
<td>131072</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Through Full LRU</td>
<td>131072</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
<tr>
<td>Write-Through Set</td>
<td>131072</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131072</td>
</tr>
</tbody>
</table>

As can be seen in Table 12, all of the cache implementations result in 131072 write-misses since each write is to a different block, and none of the blocks exist in the cache prior to the write being issued to it. This value verifies that the write-miss counter works correctly. The only thing that differs among the cache implementations is the number of memory accesses required to successfully execute the write test program. These differences are discussed in the following sections.

#### 5.2.1 Write-Back Cache Implementations

All of the write-back cache implementations required 196608 memory accesses to successfully execute the write test test-bench program. Recall that in a write-back cache, on a write-miss, the needed block is brought into the cache and then modified in the cache—without main memory being notified of the modification. Hence, after the first 65536 blocks are read into the cache and then modified, all of the blocks in the cache are dirty, and the cache is now full. On the next write, a block must be evicted from the cache. Since the evicted block is dirty, its contents must be reported to main memory so that the coherency between main memory and the cache can be re-established. This update causes one memory access. After the evicted block is updated in main memory, the new block must be read into the cache, thus causing another memory access. A total of 131072 blocks must be read into the cache prior to their modifications, and 65536 blocks must be updated in main memory after they are evicted from the cache, resulting in a total of 196608 memory accesses, as reported in Table 12.
In terms of coherency, the cache and main memory are not coherent with respect to the final 65536 blocks that were modified. This is due to the fact that in a write-back cache, main memory is not notified of a modification to a block in the cache until after the block is evicted from the cache, if it ever is.

5.2.2 Write-Once Cache Implementations

In the write-once cache implementations, each block is brought into the cache, causing one memory access, and then modified. This modification causes a second memory access, since in a write-once cache implementation, the first write to any block in the cache is reported to main memory. Thus, two memory accesses are required on each write. No memory access is required when a block is evicted from the cache since the evicted block is reserved, and hence its modified contents have been reported to main memory. Since there is a total of 131072 write operations performed in the write test, 262144 memory accesses are required, as reported in Table 12. In addition, since all of the writes are reported to main memory, the write-once cache is always consistent with main memory throughout the execution of the write test.

5.2.3 Write-Through Cache Implementations

In all of the write-through cache implementations, the write test required 131072 memory accesses. Recall that in a write-through cache, on a write-miss, main memory is directly supplied with the correct data, and the block is not brought into the cache. Since every write in the write test is to a different block, and none of the blocks previously exist in the cache, only one memory access is required for each write in order to write the new data into the appropriate block in main memory. Since there are 131072 write operations performed in the write test, 131072 memory accesses are required to provide main memory with the correct data, as reported in Table 12. In addition, since no block enters the cache, there are no coherency issues between the cache and main memory. In fact, all blocks in the cache remain undefined since no information ever entered the cache. This fact differentiates the execution of the write test on the write-once cache from the write-through cache since in the former, the last 65536 blocks that were modified reside in the cache, while in the latter, none of the blocks reside in the cache.

5.3 The Read-Write-Write-Read (RWWR) Test

The read-write-write-read (RWWR) test was designed to verify the operation of the cache implementations when reads are followed by writes to different locations, which are in turn followed by writes to the blocks that were first read, which are finally followed by reads of the blocks that were first written to. This test first initializes main memory, and then reads 10000 addresses (addresses 0-9999 in the first iteration) of
main memory into the cache. This read is followed by a write to 10000 different locations (addresses 10000-19999 in the first iteration) of main memory. This write is followed by a write to the first set of blocks that were read into the cache (blocks 0-9999 of the first iteration). This second write is then followed by a read of the first set of blocks that were written to (blocks 10000-19999 of the first iteration). This entire process is repeated five times, so that a total of 100000 blocks are read and modified. Table 13 contains the data collected when the RWWR test was executed on all of the cache implementations of this thesis. The following sections explain how the data presented in Table 13 was calculated.

<table>
<thead>
<tr>
<th>Cache Architecture</th>
<th>Memory Accesses</th>
<th>Read-Hits</th>
<th>Read-Misses</th>
<th>Write-Hits</th>
<th>Write-Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-Back Direct</td>
<td>134464</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Back Full FIFO</td>
<td>134464</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Back Full LRU</td>
<td>134464</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Back Set</td>
<td>134464</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Once Direct</td>
<td>200000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Once Full FIFO</td>
<td>200000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Once Full LRU</td>
<td>200000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Once Set</td>
<td>200000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Through Direct</td>
<td>200000</td>
<td>0</td>
<td>10000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Through Full FIFO</td>
<td>200000</td>
<td>0</td>
<td>10000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Through Full LRU</td>
<td>200000</td>
<td>0</td>
<td>10000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>Write-Through Set</td>
<td>200000</td>
<td>0</td>
<td>10000</td>
<td>50000</td>
<td>50000</td>
</tr>
</tbody>
</table>

Table 13. Data Analysis of the RWWR Test

5.3.1 Write-Back Cache Implementations

In all of the write-back cache implementations, the RWWR test required 134464 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 14.

In the write-back cache, every time a write-miss occurs, the required block is first brought into the cache and then modified, following the write-allocate scheme used with write-back caches. This occurs on the first write of each of the five iterations, thus causing a total of 50000 memory accesses— 1 access to read each of the ten thousand blocks into the cache, for each of the five iterations. When the cache becomes full after the 65536th block has been read into the cache, different blocks from the cache must be evicted in order to make room for the incoming data. Since all of the blocks being evicted are dirty, their contents must be reported to main memory. This causes 34464 memory accesses— 4464 from the first read of the 60000-69999 iteration, and then 10000 each for the remaining three iterations. The remaining 50000
memory accesses are due to the 10000 read-misses that occur as a result of the first read within each iteration.

It is also important to note that the write-back cache and main memory are only consistent with respect to the first 34464 blocks that were brought into the cache, since these are the only blocks that have been evicted from the cache, and are hence the only blocks whose modified contents have been reported to main memory.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Block Numbers Accessed</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Read</td>
<td>0-9999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read blocks</td>
</tr>
<tr>
<td>1st Write</td>
<td>10000-19999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read blocks into cache prior to writing to it</td>
</tr>
<tr>
<td>2nd Write</td>
<td>0-9999</td>
<td>10000 write-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd Read</td>
<td>10000-19999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>20000-29999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read blocks</td>
</tr>
<tr>
<td>1st Write</td>
<td>30000-39999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block into cache prior to writing to it</td>
</tr>
<tr>
<td>2nd Write</td>
<td>20000-29999</td>
<td>10000 read-misses</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd Read</td>
<td>30000-39999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>40000-49999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read blocks</td>
</tr>
<tr>
<td>1st Write</td>
<td>50000-59999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block into cache prior to writing to it</td>
</tr>
<tr>
<td>2nd Write</td>
<td>40000-49999</td>
<td>10000 read-misses</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd Read</td>
<td>50000-59999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>60000-69999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read blocks 4464 accesses to update evicted blocks</td>
</tr>
<tr>
<td>1st Write</td>
<td>70000-79999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read blocks 10000 accesses to update evicted blocks</td>
</tr>
<tr>
<td>2nd Write</td>
<td>60000-69999</td>
<td>10000 write-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd Read</td>
<td>70000-79999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>80000-89999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read blocks into cache 10000 accesses to update evicted blocks</td>
</tr>
<tr>
<td>1st Write</td>
<td>90000-99999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block into cache 10000 accesses to update evicted blocks</td>
</tr>
<tr>
<td>2nd Write</td>
<td>80000-89999</td>
<td>10000 write-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd Read</td>
<td>90000-99999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>Read-Hits: 50000</strong></td>
<td><strong>Read-Misses: 50000</strong></td>
<td><strong>Write-Hits: 50000</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Write-Misses: 50000</strong></td>
<td></td>
<td><strong>Total Accesses: 134464</strong></td>
</tr>
</tbody>
</table>

Table 14. Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Back Cache Implementations
5.3.2 Write-Once Cache Implementations

In all of the write-once cache implementations, the RWWR test required 200000 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 15.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Block Numbers Accessed</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Read</td>
<td>0-9999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>10000-19999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>0-9999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>10000-19999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>20000-29999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>30000-39999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>20000-29999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>30000-29999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>40000-49999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>50000-59999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10000 accesses to update main</td>
</tr>
<tr>
<td>2nd Write</td>
<td>40000-49999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>50000-59999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>60000-69999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>70000-79999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>60000-69999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>70000-79999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>1st Read</td>
<td>80000-89999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>90000-99999</td>
<td>10000 write-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>80000-89999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>90000-99999</td>
<td>10000 read-hits</td>
<td>Idle</td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td>Read-Hits: 50000</td>
<td>Total Accesses: 200000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-Misses: 50000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write-Hits: 50000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write-Misses: 50000</td>
<td></td>
</tr>
</tbody>
</table>

Table 15. Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Once Cache Implementations

As shown in Table 15, 200000 memory accesses are needed every time a write is performed on a block in the cache, since in a write-once cache environment, the data modified on the first write to a block in the cache is reported to main memory. Hence, the block must be brought into the cache (one memory access) and then modified, reporting the modified data contents to main memory (second memory access). There
are a total of 50000 write-misses in the write-once cache executing the RWWR test, therefore 100000 memory accesses are required as a result of these write-misses. The remaining 100000 memory accesses are due to the 100000 read-misses that occur in order to bring the 10000 necessary blocks into the cache. Therefore, a grand total of 200000 memory accesses are required in the execution of the RWWR test on a write-once cache. It is important to note, that since all of the writes are reported to main memory, main memory and the cache are always consistent with each other throughout the execution of the RWWR test.

In addition, it is important to note that since only one write is performed on each block, the modified contents of each block have already been reported to main memory. Therefore, when a block is evicted from the cache, there is no need to report its modified contents to main memory.

5.3.3 Write-Through Cache Implementations

In all of the write-through cache implementations, the RWWR test required 200000 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 16.

As shown in Table 16, 10000 memory accesses are needed every time a write is performed on a block in the cache, since in a write-through cache environment, the modified data is reported directly to main memory. In detail, on a write-hit, the data is modified in both the cache and main memory, and on a write-miss, the block is modified only in main memory, without bringing the required block into the cache. This follows the no-write-allocate scheme that is employed with the write-through cache implementations of this thesis.

Since there are 100000 write operations in the RWWR test, 100000 memory accesses are required to satisfy these write operations. The remaining 100000 memory accesses are due to bringing in the 100000 blocks into the cache as a result of the 100000 read-misses that occur. The increase in the number of read-misses in the write-through cache, as compared to the write-back and write-once cache implementations, is due to the fact that the required blocks are not brought into the cache on the first write of each iteration, as they are in the other two cache implementations. Thus, when these blocks are referenced again in the second read of each iteration, a read-miss occurs for each one.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Block Numbers Accessed</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Read</td>
<td>0-9999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>10000-19999</td>
<td>10000 write-misses</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>0-9999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>10000-19999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Read</td>
<td>20000-29999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>30000-39999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>2nd Write</td>
<td>20000-29999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>30000-29999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Read</td>
<td>40000-49999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>50000-59999</td>
<td>10000 write-misses</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>40000-49999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>50000-59999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Read</td>
<td>60000-69999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>70000-79999</td>
<td>10000 write-misses</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>60000-69999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>70000-79999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Read</td>
<td>80000-89999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>1st Write</td>
<td>90000-99999</td>
<td>10000 write-misses</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Write</td>
<td>80000-89999</td>
<td>10000 write-hits</td>
<td>10000 accesses to update memory</td>
</tr>
<tr>
<td>2nd Read</td>
<td>90000-99999</td>
<td>10000 read-misses</td>
<td>10000 accesses to read block</td>
</tr>
<tr>
<td>Write-Hits: 50000</td>
<td>Read-Hits: 0</td>
<td>Read-Misses: 50000</td>
<td>Total Accesses: 200000</td>
</tr>
</tbody>
</table>

Table 16. Description of the Cache and Main Memory Performance for the RWWR Test on the Write-Through Cache Implementations

5.4 **The Sum Test**

The sum test was designed as a sample application that could be run on the processor. This test was performed last, after all of the basic operations had been verified, and intensive testing was needed on the cache implementations. This test first initializes main memory, and then clears the sum block, which is always the last block in main memory. The program then finds the sum of the remaining blocks of main memory by first reading in the sum block and converting the sum into an integer. Then, the next block to add is read into the cache, and its data is converted into an integer. Finally, the sum of these two integers is computed and written back into the sum block. The program uses the MEMORY_SIZE constant so that all the blocks in main memory will be used in computing the sum, except, of course, the previous data value that was stored in the last block of main memory, which was lost to form the sum block. Table 17 contains the data collected when the sum test was executed on all of the cache implementations of this thesis. As can be seen in Table 17, the data collected for the sum test differs for each of the cache implementations, and these differences are discussed in the following sections.
Before discussing the results of the sum test, it is important to note that the maximum integer allowed in VHDL is $2^{14}$, or $2^{31}$. During the execution of the sum test on the implementations of this thesis, this maximum value was achieved and then surpassed, yielding an incorrect final value for the cumulative sum. Since this thesis is more concerned with the main memory and cache accesses than with the correct addition of several numbers, the problem was not resolved since its solution does not effect the number of main memory or cache accesses. However, if in the future, the sum test is to be executed to obtain correct summation values, a solution to this maximum integer problem should be found.

### 5.4.1 Write-Back Direct Cache Implementation and Fully-Associative Cache Implementation with FIFO Replacement Algorithm

In both the direct-mapped and fully-associative mapped write-back cache using the FIFO replacement algorithm, the sum test required 131074 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 18.

As can be seen in Table 18, both the direct-mapped and fully-associative mapped with FIFO replacement cache implementations require the eviction of the sum block from the cache when the cache becomes full on the 65536th addition. The sum block is evicted in the direct-mapped cache since the sum block and the 65536th block in main memory are both mapped to the same position in the cache. The sum block is evicted in the fully-associative mapped cache with FIFO replacement algorithm because the sum block was the first block brought into the cache, and under the rules of the FIFO replacement algorithm, the first
block brought into the cache is the first one to be evicted from it, regardless of how recently it was last referenced.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 65535th</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 65535th</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to update evicted sum</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 131071th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 131071th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td>Totals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Hits: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Misses: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Hits: 131070</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Misses: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total: 131074 accesses</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 18. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Direct-Mapped and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation

On the 65536th addition, the sum block is first read, causing a read-hit, and its data value is converted into an integer. The next block to add is then read, causing a read-miss, however the cache is full so a block must be evicted from the cache. As discussed above, the sum block is evicted from the cache, and since the sum block is dirty, its contents must be updated in main memory, thus causing one memory access. After the sum block has been evicted, the new data block is read into the cache, causing a second memory access, and its data value is converted into an integer. The sum of the two integers previously calculated is computed, and the sum is written back to the sum block, however this write causes a write-miss, since the sum block was just evicted. This write-miss causes another memory access to occur, for a total of three memory accesses, and the sum block is read back into the cache and modified.

It is important to note that at the end of the execution of the sum program on the write-back cache, the cache is the only level of the memory hierarchy that contains the correct final sum of all of the data values in main memory. The value of the cumulative sum was written to main memory when the sum block was evicted from the cache on the 65536th addition, however, this value is not the correct final sum. Main
memory will not get the correct final sum until after the sum block is evicted from the cache, which does not occur in a write-back cache implementation when the sum program is only executed once.

5.4.2 Write-Back Fully-Associative Cache Implementation with LRU Replacement Algorithm

In the fully-associative mapped write-back cache using the LRU replacement algorithm, the sum test required 131072 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 19.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total: 131072 accesses</td>
</tr>
<tr>
<td></td>
<td>Totals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Hits: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Misses: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Hits: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Misses: 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 19. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation

Unlike the direct-mapped cache and the fully-associative mapped cache with FIFO replacement, the fully-associative cache with LRU replacement does not require the eviction of the sum block from the cache. Recall that in the LRU replacement algorithm, the block evicted is the one that was used the least recently, or, in other words, the block that has not been used in the longest period of time. In the sum test, this block is the one corresponding to the first address block that was read into the block, which was used for the first addition, but has not been referenced since. Since no modification has been made to this block, its eviction does not require an update to main memory, so the new block simply replaces the information previously stored for this address block.

It is important to note that if the LRU replacement algorithm is used in a write-back cache executing the sum program, main memory will never have the correct sum value since the sum block is never evicted from the cache. Therefore, coherency is never preserved between the cache and main memory with respect to the sum block. This may be an important factor if write-back caches using the LRU replacement algorithm are used in a multiprocessor environment, where coherency between cache and main memory, and among the caches, is key to the correct and efficient implementation of a program.

64
5.4.3 Write-Back Set-Associative Cache Implementation

In the set-associative mapped write-back cache, the sum test required 131080 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 20.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Write the Sum</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>81921st to 98303rd Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81921st to 98303rd Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Write-Hits: 131067</td>
<td>Read-Hits: 131071</td>
<td>Total: 131080 accesses</td>
<td></td>
</tr>
<tr>
<td>Write-Misses: 5</td>
<td>Read-Misses: 131071</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 20. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Back Set-Associative Mapped Cache Implementation

As can be seen in Table 20, the write-back set-associative cache requires the sum block to be evicted on the 65536th addition, and then every 16384 additions afterwards. This is because the cache is divided into
16384 sets, and after the cache becomes full on the 65536th addition, a block must be evicted from the cache on the next read-miss in order to make room for the incoming data. Since the write-back set-associative cache implements a pseudo-random replacement algorithm in which a dirty block is the first to be evicted, and since the sum block is dirty after every addition, the sum block is evicted every time the cache is full, and its contents must be updated in main memory, requiring another memory access.

It is important to note that main memory will not have the final result of the cumulative sum of all the data values in main memory, however, it will have the most recent update compared to the direct and fully-associative mapped write-back caches, since it was last updated on the 114688th addition, while the other cache implementations update main memory for the final time on the 65536th addition.

5.4.4 Write-Once Direct Cache Implementation and Fully-Associative Cache Implementation with FIFO Replacement Algorithm

In both the direct-mapped and fully-associative mapped write-once cache using the FIFO replacement algorithm, the sum test required 131076 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 21.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 access to update</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 for evicted sum block</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 access to update</td>
</tr>
<tr>
<td>65537th to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65537th to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td></td>
<td>Total: 131076 accesses</td>
</tr>
<tr>
<td></td>
<td>Totals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Hits: 131070</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Misses: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Hits: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Misses: 131071</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 21. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Direct-Mapped Cache Implementation and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation

66
As can be seen in Table 21, the performance of these two cache implementations on a write-once cache is very similar to that of the write-back cache. The main difference is in the number of memory accesses. Recall that the first write to a block in a write-once cache is reported to main memory, hence the additional memory access in the beginning of the program when the sum block is cleared. The sum block is first brought into the cache (one memory access) and then written to. This write causes a second memory access to occur since it is the first write to the block. The additional modifications to the sum block that occur later in the program are not reported to main memory until the sum block is evicted from the cache on the 65536th addition.

On the 65536th addition, the sum block is first read, causing a read-hit, and its data value is converted into an integer. The next block to add is then read into the cache (one memory access), however, the cache is full, so a block must be evicted from the cache. As explained in the write-back cache implementations, the sum block is candidate for eviction, however, since the sum block is dirty, its contents must be updated in main memory, causing a second memory access. After this update, the sum is calculated and is written back to the sum block, however, this write causes a write-miss to occur in the cache since the sum block was just evicted. When the sum block is read back into the cache (the third memory access) it is modified, and since this is the first write to the block, the data modification is reported to main memory, causing a fourth memory access.

It is important to note that just like the write-back cache implementations, main memory does not have the final value of the cumulative sum. The last update that was reported to main memory was on the 65536th addition, after the sum block was written to. This is different from the write-back direct-mapped and fully-associative with FIFO replacement caches since the last value written to the sum block in main memory of the write-back caches is the result of the first 65535 additions, while the last value written to main memory in the write-once caches is the result of the first 65536 additions. Hence, the write-once direct-mapped or fully-associative mapped with FIFO replacement implementation has the most recent value of the cumulative sum.

5.4.5 Write-Once Fully-Associative Cache Implementation with LRU Replacement Algorithm

In the fully-associative mapped write-once cache using the LRU replacement algorithm, the sum test required 131073 memory accesses. This is one greater than the number of accesses required for the write-back fully-associative cache using LRU replacement algorithm, since the sum block must be updated in
main memory after the sum block is cleared in the initialization step. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 22.

It is important to note that if the LRU replacement algorithm is used in a write-once cache executing the sum program, main memory will never have the correct sum value since the sum block is never evicted from the cache. Therefore, coherency is never preserved between the cache and main memory with respect to the sum block. This may be an important factor if write-once caches using the LRU replacement algorithm are used in a multiprocessor environment, where coherency between cache and main memory, and among the caches, is key to the correct and efficient implementation of a program.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 access to update</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total: 131073 accesses</td>
</tr>
</tbody>
</table>

Table 22. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation

5.4.6 Write-Once Set-Associative Cache Implementation

In the set-associative mapped write-once cache, the sum test required 131085 memory accesses. This is five more memory accesses than the number required for the write-back cache. The extra memory accesses are due to the need to update main memory when the sum block is read back into the cache and modified on a write-miss. This occurs on the 65536th, 81920th, 98304th, and 114688th additions. An update of main memory is also required on the initialization step, when the sum block is read into the cache and cleared. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 23.
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 accesses to update</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>1st to 65535th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>65537th to 81919th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>81921st to 98303rd Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81921st to 98303rd Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>81921st to 98303rd Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>98305th to 114687th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read block</td>
</tr>
<tr>
<td>114689th to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td></td>
<td>Total: 131085 accesses</td>
</tr>
<tr>
<td></td>
<td>Read-Hits: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read-Misses: 131071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Hits: 131067</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-Misses: 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 23. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Once Set-Associative Mapped Cache Implementation

It is important to recall that main memory will not have the final result of the cumulative sum of all the data values in main memory, however, it will have the most recent update compared to the direct and fully-
associative mapped write-once caches, since it was last updated on the 114688th addition, while the other caches received their last update on the 65536th addition.

5.4.7 Write-Through Direct Cache Implementation and Fully-Associative Cache Implementation with FIFO Replacement Algorithm

In both the direct-mapped and fully-associative mapped write-through cache using the FIFO replacement algorithm, the sum test required 262144 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 24.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Main Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>1st Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>1st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>2nd Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle since no change</td>
</tr>
<tr>
<td>65536th to 65535th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th to 65535th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65537th to 65536th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>65537th to 65536th Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65538th to 65537th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td></td>
<td>262144 accesses</td>
</tr>
</tbody>
</table>

Table 24. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Through Direct-Mapped Cache Implementation and Fully-Associative Mapped with FIFO Replacement Algorithm Cache Implementation

It is important to note that the first write-hit to the sum block causes no memory access since the data value of main memory and the data value that will be written to the sum block of main memory are the same. Since the data in main memory will be modified with the same data, why bother updating it? Therefore, no memory access is recorded and the data value of main memory is kept unchanged.
The main difference between the execution of the sum program on the write-through cache implementations and the execution of the sum process on the write-back and write-once cache implementations is the extra memory access required to update main memory on each sum calculation. In addition, when the sum block is cleared during the initialization step, the block is not brought into the cache, as is done in the write-back and write-once cache implementations. This is due to the no-write-allocate scheme that is used in the write-through cache implementations, in which the required block is not brought into the cache on a write-miss. Instead, main memory is modified directly. Since the sum block is not brought into the cache on the initialization step, it must be brought into the cache on the first addition, causing another memory access to occur.

It is important to note that at all times, main memory contains the most up-to-date value of the sum, since a write-through cache is always consistent with main memory. The results of this test show that if coherency between the cache and main memory is desired, one must pay the price of more memory accesses.

5.4.8 Write-Through Fully-Associative Cache Implementation with LRU Replacement Algorithm

In the fully-associative mapped write-through cache using the LRU replacement algorithm, the sum test required 262143 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 25.

Again, main memory does not need to be modified on the first write-hit since the data value to be written to the sum block is the same value that is currently stored in the sum block of main memory. The main thing to note about the execution of the sum test on the write-through fully-associative cache with LRU replacement algorithm is the fact that the sum block is never evicted. This means that one less memory access is required in this cache implementation than in the write-through direct-mapped or fully-associative mapped with FIFO replacement algorithm, as shown by comparing Table 24 to Table 25.
Table 25. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Through Fully-Associative Mapped with LRU Replacement Algorithm Cache Implementation

5.4.9 Write-Through Set-Associative Cache Implementation

In the set-associative mapped write-through cache, the sum test required 262145 memory accesses. A breakdown of these accesses, as well as a description of how the cache parameters were calculated is presented in Table 26.

Again, main memory does not need to be modified on the first write-hit since the data value to be written to the sum block is the same value that is currently stored in the sum block of main memory. As shown in Table 26, the set-associative cache requires that the sum block be brought back into the cache after it was evicted in the 65536th, 81920th, 98304th, and 114688th additions. The difference between the write-through cache implementation and the write-back and write-once implementations is that in both the write-back and write-once implementations, the sum block is evicted when the next block to add is read in, but is brought back into the cache after the sum is computed and written to the sum block— all within the same addition in which the sum block was evicted. This whole process causes four memory accesses in the write-once cache, since the new information is read in (one memory access), the sum block is updated in main memory (second memory access), the sum block is read back into the set (third memory access), and the write to the sum block is reported to main memory (fourth memory access). In the write-back cache, this process causes three memory accesses— one to read the new information, a second to update the sum block in main memory, and a third to read the sum block back into the cache after the sum has been computed.

72
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
<th>Cache Performance</th>
<th>Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Clear Sum Block</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>1st Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>1st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>1st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd to 65535th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>2nd to 65535th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>2nd to 65535th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65536th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>65537th Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65537th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65537th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>65538th to 81919th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>65538th to 81919th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>65538th to 81919th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>81920th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>81921th Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>81921th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>81921th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>81922nd to 98303rd Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>81922nd to 98303rd Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>81922nd to 98303rd Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>98304th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>98305th Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>98305th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>98305th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>98306th to 114687th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>98306th to 114687th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>98306th to 114687th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to update</td>
</tr>
<tr>
<td>114688th Addition</td>
<td>Write the Sum</td>
<td>Write-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>114689th Addition</td>
<td>Read Sum Block</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>114689th Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>114689th Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td>114690th to 131071st Addition</td>
<td>Read Sum Block</td>
<td>Read-Hit</td>
<td>Idle</td>
</tr>
<tr>
<td>114691st to 131071st Addition</td>
<td>Read Next to Add</td>
<td>Read-Miss</td>
<td>1 access to read</td>
</tr>
<tr>
<td>114691st to 131071st Addition</td>
<td>Write the Sum</td>
<td>Write-Hit</td>
<td>1 access to update</td>
</tr>
<tr>
<td></td>
<td>Write-Hits: 131067</td>
<td>Read-Hits: 131066</td>
<td>Total: 262145</td>
</tr>
<tr>
<td></td>
<td>Write-Misses: 5</td>
<td>Read-Misses: 131076</td>
<td>accesses</td>
</tr>
</tbody>
</table>

Table 26. Description of the Cache and Main Memory Performance for the Sum Test on the Write-Through Set-Associative Mapped Cache Implementation
In the write-through cache implementation, however, the sum block is evicted from the cache and after the sum is computed, the sum block is modified in main memory only, causing one memory access. On the next addition, when the sum block is read, a read-miss occurs, and the sum block must be brought into the cache, causing a second memory access. Thus, only two memory accesses are required to handle the eviction of the sum block in the write-through cache, however, these memory accesses are spanned across two additions instead of one, as in the write-back and write-once cache implementations. Though at first glance the write-through cache looks like an improvement over the write-back and write-once caches, one must not forget that the write-through cache requires that all writes be reported to main memory, which accounts for the additional memory accesses that are required when the sum test is executed on the write-through cache.

### 5.5 Summary of the Coherency Between the Cache and Main Memory for All of the Test-Benches

The previous sections mainly focused on the number of memory accesses and the cache performance in terms of the number of read- and write misses and hits. The coherency issues were briefly mentioned, but are reported again in Table 27.

<table>
<thead>
<tr>
<th>Cache Architecture</th>
<th>Read Test</th>
<th>Write Test</th>
<th>RWWR Test</th>
<th>Sum Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Back Direct</td>
<td>Always Coherent</td>
<td>First 65536 blocks</td>
<td>First 34464 blocks</td>
<td>65535 additions</td>
</tr>
<tr>
<td>Write Back Full FIFO</td>
<td>Always Coherent</td>
<td>First 65536 blocks</td>
<td>First 34464 blocks</td>
<td>65535 additions</td>
</tr>
<tr>
<td>Write Back Full LRU</td>
<td>Always Coherent</td>
<td>First 65536 blocks</td>
<td>First 34464 blocks</td>
<td>Never</td>
</tr>
<tr>
<td>Write Back Set</td>
<td>Always Coherent</td>
<td>First 65536 blocks</td>
<td>First 34464 blocks</td>
<td>114687 additions</td>
</tr>
<tr>
<td>Write Once Direct</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>65536 addition</td>
</tr>
<tr>
<td>Write Once Full FIFO</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>65536 addition</td>
</tr>
<tr>
<td>Write Once Full LRU</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>Never</td>
</tr>
<tr>
<td>Write Once Set</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
<td>114688 additions</td>
</tr>
<tr>
<td>Write Through Direct</td>
<td>Always Coherent</td>
<td>Cache is not used</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
</tr>
<tr>
<td>Write Through Full FIFO</td>
<td>Always Coherent</td>
<td>Cache is not used</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
</tr>
<tr>
<td>Write Through Full LRU</td>
<td>Always Coherent</td>
<td>Cache is not used</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
</tr>
<tr>
<td>Write Through Set</td>
<td>Always Coherent</td>
<td>Cache is not used</td>
<td>Always Coherent</td>
<td>Always Coherent</td>
</tr>
</tbody>
</table>

Table 27. The Coherency Status of Each Test-Bench on Each Cache Architecture

As can be seen in Table 27, the write-through cache is always coherent with main memory when it is used, however, one should not forget the high number of memory accesses required to keep main memory and the write-through cache coherent. In addition, on the sum test, one should not use the LRU replacement algorithm since main memory will never receive an updated value of the sum.
6 Improving Cache Memory Performance

When designing a cache, an architect has a three-fold dilemma when he/she wants to reduce the average memory access time: increasing the block size does not improve the average access time, since the lower incurred miss rate does not offset the higher miss penalty; making the cache bigger would also make it slower, thus jeopardizing the CPU clock rate; and making the cache more associative would also make it slower, again affecting the CPU clock rate [Hennessy1990 p. 454].

Explaining this dilemma differently, if one increases the size of the block in the cache, there will be more data available in the cache, thus allowing more cache hits to occur and lowering the miss-rate. However, the cost to evict this enlarged block from the cache is much greater than the cost to evict a cache block of a smaller size, and this increase in the cost often does not outweigh the reduction in the miss-rate.

The second part of the dilemma states that if the cache is made bigger, there will be more data stored in the cache so that, once again, the cache miss-rate will be reduced. However, the time required to search through the entire cache to find the requested block, as is needed in a fully-associative mapped cache, increases, thus making the CPU wait longer in order to receive the requested data. Thus, the increase in the size of the cache does not outweigh the increase in the memory access time.

The third part of the dilemma states that if one increases the associativity of the cache, meaning to increase the number of blocks in a set, the time required to search through the entire enlarged set to retrieve the requested data increases since there are more blocks to search. This increase in time will cause the CPU to wait even longer to receive the requested data from the cache. Further increases in the associativity of the set-associative mapped cache will cause the cache to approach a fully-associative mapped strategy, which requires the cache to search through every block to see if the address corresponds to the one that was requested. Thus, the effects of increasing the associativity of the cache do not outweigh the increases in the memory access time.

Despite the dilemma discussed above, there are several things that a designer may do to improve upon the performance of a cache memory architecture. Improvements in the miss rate, miss penalty, and hit time are discussed in the following sections.
6.1 Reducing the Miss Rate by Increasing the Associativity of the Cache
Referring back to Table 3, we showed that a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2. This is known as the 2:1 cache rule of thumb. This rule shows that increasing the associativity of a cache (i.e., making a 2-way set-associative cache a 4-way set-associative cache) has little effect on reducing the miss rate. In addition, the increase in the associativity may increase the hit time, since there are now more blocks in the set that need to be searched in order to determine if the requested block is in the cache [Hennessy1996, p. 396].

The effects of increasing the associativity of the cache are shown in Table 28. In this table, a sample program was executed on cache implementations with varying sizes and degrees of associativity. The miss rates were computed for each implementation and are shown in Table 28. As can be seen by the bold numbers in the 32 KB, 64 KB, and 128 KB caches using 8-way associativity, the miss rate is higher than the 4-way set-associative cache of the same sizes. These higher miss rates show that increasing the associativity of a cache may increase the average memory access time.

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.65</td>
<td>6.60</td>
<td>6.22</td>
<td>5.44</td>
</tr>
<tr>
<td>2</td>
<td>5.90</td>
<td>4.90</td>
<td>4.62</td>
<td>4.09</td>
</tr>
<tr>
<td>4</td>
<td>4.60</td>
<td>3.95</td>
<td>3.57</td>
<td>3.19</td>
</tr>
<tr>
<td>8</td>
<td>3.30</td>
<td>3.00</td>
<td>2.87</td>
<td>2.59</td>
</tr>
<tr>
<td>16</td>
<td>2.45</td>
<td>2.20</td>
<td>2.12</td>
<td>2.04</td>
</tr>
<tr>
<td>32</td>
<td>2.00</td>
<td>1.80</td>
<td>1.77</td>
<td>1.79</td>
</tr>
<tr>
<td>64</td>
<td>1.70</td>
<td>1.60</td>
<td>1.57</td>
<td>1.59</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.42</td>
<td>1.44</td>
</tr>
</tbody>
</table>

Table 28. Average Memory Access Time for Different Cache Sizes and Associativities [Hennessy1996, p. 397]

A similar study was performed by us using the sum test and varying the degree of associativity within a fixed sized cache. By fixing the size of the cache at 512 KB, and using a 1 MB main memory unit, the BLOCKS_PER_SET constant was varied in the WRITE_BACK_CACHE_TYPE, WRITE_ONCE_CACHE_TYPE, and WRITE_THROUGH_CACHE_TYPE packages. The number of main memory accesses, read-hits, read-misses, write-hits, and write-misses were obtained for 1-way (same as direct-mapped), 2-way, 4-way, 8-way, 16-way, 32-way, 64-way, 128-way, 256-way, and 512-way set-associative caches using the write-back, write-once, and write-through architectures. This data is presented in Table 29.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Degree of Associativity</th>
<th>Memory Accesses</th>
<th>Number of Read-Hits</th>
<th>Number of Read-Misses</th>
<th>Number of Write-Hits</th>
<th>Number of Write-Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-Back</td>
<td>1-way</td>
<td>131074</td>
<td>131071</td>
<td>131071</td>
<td>131070</td>
<td>2</td>
</tr>
<tr>
<td>Write-Once</td>
<td>1-way</td>
<td>131076</td>
<td>131071</td>
<td>131071</td>
<td>131070</td>
<td>2</td>
</tr>
<tr>
<td>Write-Through</td>
<td>1-way</td>
<td>262144</td>
<td>131069</td>
<td>131073</td>
<td>131070</td>
<td>2</td>
</tr>
<tr>
<td>Write-Back</td>
<td>2-way</td>
<td>131076</td>
<td>131071</td>
<td>131071</td>
<td>131069</td>
<td>3</td>
</tr>
<tr>
<td>Write-Once</td>
<td>2-way</td>
<td>131079</td>
<td>131071</td>
<td>131071</td>
<td>131069</td>
<td>3</td>
</tr>
<tr>
<td>Write-Through</td>
<td>2-way</td>
<td>262143</td>
<td>131068</td>
<td>131074</td>
<td>131069</td>
<td>3</td>
</tr>
<tr>
<td>Write-Back</td>
<td>4-way</td>
<td>131080</td>
<td>131071</td>
<td>131071</td>
<td>131067</td>
<td>5</td>
</tr>
<tr>
<td>Write-Once</td>
<td>4-way</td>
<td>131085</td>
<td>131071</td>
<td>131071</td>
<td>131067</td>
<td>5</td>
</tr>
<tr>
<td>Write-Through</td>
<td>4-way</td>
<td>262145</td>
<td>131066</td>
<td>131076</td>
<td>131067</td>
<td>5</td>
</tr>
<tr>
<td>Write-Back</td>
<td>8-way</td>
<td>131088</td>
<td>131071</td>
<td>131071</td>
<td>131063</td>
<td>9</td>
</tr>
<tr>
<td>Write-Once</td>
<td>8-way</td>
<td>131097</td>
<td>131071</td>
<td>131071</td>
<td>131063</td>
<td>9</td>
</tr>
<tr>
<td>Write-Through</td>
<td>8-way</td>
<td>262149</td>
<td>131062</td>
<td>131080</td>
<td>131063</td>
<td>9</td>
</tr>
<tr>
<td>Write-Back</td>
<td>16-way</td>
<td>131104</td>
<td>131071</td>
<td>131071</td>
<td>131055</td>
<td>17</td>
</tr>
<tr>
<td>Write-Once</td>
<td>16-way</td>
<td>131121</td>
<td>131071</td>
<td>131071</td>
<td>131055</td>
<td>17</td>
</tr>
<tr>
<td>Write-Through</td>
<td>16-way</td>
<td>262157</td>
<td>131054</td>
<td>131088</td>
<td>131055</td>
<td>17</td>
</tr>
<tr>
<td>Write-Back</td>
<td>32-way</td>
<td>131136</td>
<td>131071</td>
<td>131071</td>
<td>131039</td>
<td>33</td>
</tr>
<tr>
<td>Write-Once</td>
<td>32-way</td>
<td>131139</td>
<td>131071</td>
<td>131071</td>
<td>131039</td>
<td>33</td>
</tr>
<tr>
<td>Write-Through</td>
<td>32-way</td>
<td>262173</td>
<td>131038</td>
<td>131104</td>
<td>131039</td>
<td>33</td>
</tr>
<tr>
<td>Write-Back</td>
<td>64-way</td>
<td>131200</td>
<td>131071</td>
<td>131071</td>
<td>131007</td>
<td>65</td>
</tr>
<tr>
<td>Write-Once</td>
<td>64-way</td>
<td>131265</td>
<td>131071</td>
<td>131071</td>
<td>131007</td>
<td>65</td>
</tr>
<tr>
<td>Write-Through</td>
<td>64-way</td>
<td>262205</td>
<td>131006</td>
<td>131136</td>
<td>131007</td>
<td>65</td>
</tr>
<tr>
<td>Write-Back</td>
<td>128-way</td>
<td>131328</td>
<td>131071</td>
<td>131071</td>
<td>130943</td>
<td>129</td>
</tr>
<tr>
<td>Write-Once</td>
<td>128-way</td>
<td>131457</td>
<td>131071</td>
<td>131071</td>
<td>130943</td>
<td>129</td>
</tr>
<tr>
<td>Write-Through</td>
<td>128-way</td>
<td>262209</td>
<td>130942</td>
<td>131200</td>
<td>130943</td>
<td>129</td>
</tr>
<tr>
<td>Write-Back</td>
<td>256-way</td>
<td>131584</td>
<td>131071</td>
<td>131071</td>
<td>130815</td>
<td>257</td>
</tr>
<tr>
<td>Write-Once</td>
<td>256-way</td>
<td>131841</td>
<td>131071</td>
<td>131071</td>
<td>130815</td>
<td>257</td>
</tr>
<tr>
<td>Write-Through</td>
<td>256-way</td>
<td>262397</td>
<td>130814</td>
<td>131328</td>
<td>130815</td>
<td>257</td>
</tr>
<tr>
<td>Write-Back</td>
<td>512-way</td>
<td>132096</td>
<td>131071</td>
<td>131071</td>
<td>130559</td>
<td>513</td>
</tr>
<tr>
<td>Write-Once</td>
<td>512-way</td>
<td>132609</td>
<td>131071</td>
<td>131071</td>
<td>130559</td>
<td>513</td>
</tr>
<tr>
<td>Write-Through</td>
<td>512-way</td>
<td>262653</td>
<td>130558</td>
<td>131584</td>
<td>130559</td>
<td>513</td>
</tr>
</tbody>
</table>

Table 29. Cache and Main Memory Performance of the Sum Test for Different Degrees of Associativity

By looking at Table 29, one can see that the number of main memory accesses increases as the degree of associativity is increased among the three cache implementations. This effect is better seen graphically in Figure 3, Figure 4, and Figure 5 which plot the number of memory accesses versus the degree of associativity for the write-back, write-once, and write-through caches respectively.
Number of Memory Accesses vs. Degree of Associativity for the Write-Back Cache

![Graph showing the relationship between number of memory accesses and degree of associativity for the Write-Back Cache.]

Figure 3. Number of Memory Accesses vs. Degree of Associativity for the Write-Back Cache

Number of Memory Accesses vs. Degree of Associativity for the Write-Once Cache

![Graph showing the relationship between number of memory accesses and degree of associativity for the Write-Once Cache.]

Figure 4. Number of Memory Accesses vs. Degree of Associativity for the Write-Once Cache
Figure 5. Number of Memory Accesses vs. Degree of Associativity for the Write-Through Cache

As can be seen from these figures, there is an increase in the number of memory accesses as the degree of associativity is increased within a fixed sized cache for all three of the cache architectures implemented in this thesis. Since the number of memory accesses increases, one would expect the overall performance of the cache to degrade as the degree of associativity is increased.

Another aspect of the cache performance is the miss ratio. Measuring the miss ratio as the number of cache misses per number of cache references, and using the data in Table 29, the miss-rates were calculated for each degree of associativity and are presented in Table 30. A graphical representation of the data is shown in Figure 6. In calculating the miss-rates, there were 393214 references to the cache in the sum test.

<table>
<thead>
<tr>
<th>Degree of Associativity</th>
<th>Write-Back</th>
<th>Write-Once</th>
<th>Write-Through</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way</td>
<td>0.3333337571907</td>
<td>0.3333337571907</td>
<td>0.333342658196</td>
</tr>
<tr>
<td>2-way</td>
<td>0.3333401150520</td>
<td>0.3333401150520</td>
<td>0.333347744485</td>
</tr>
<tr>
<td>4-way</td>
<td>0.3333452013410</td>
<td>0.3333452013410</td>
<td>0.333357917063</td>
</tr>
<tr>
<td>8-way</td>
<td>0.3333553739190</td>
<td>0.3333553739190</td>
<td>0.333378262219</td>
</tr>
<tr>
<td>16-way</td>
<td>0.3333757190740</td>
<td>0.3333757190740</td>
<td>0.333418952530</td>
</tr>
<tr>
<td>32-way</td>
<td>0.3334164093850</td>
<td>0.3334164093850</td>
<td>0.333500333152</td>
</tr>
<tr>
<td>64-way</td>
<td>0.3334977900070</td>
<td>0.3334977900070</td>
<td>0.333663094396</td>
</tr>
<tr>
<td>128-way</td>
<td>0.3336605512520</td>
<td>0.3336605512520</td>
<td>0.333988618850</td>
</tr>
<tr>
<td>256-way</td>
<td>0.3339860737410</td>
<td>0.3339860737410</td>
<td>0.334639661864</td>
</tr>
<tr>
<td>512-way</td>
<td>0.3346371187190</td>
<td>0.3346371187190</td>
<td>0.335941751820</td>
</tr>
</tbody>
</table>

Table 30. Miss Rates for the Varying Degrees of Associativity
Miss Rate Versus Degree of Associativity

Figure 6. Miss Rates vs. Degree of Associativity for all of the Cache Implementations

As shown in Figure 6, there is an increase in the miss rate as the degree of associativity increases. In addition, the write-back and write-once cache implementations yield almost the same miss rates for all of the degrees of associativity. This may provoke a designer to choose either cache architecture in order to achieve the same performance. However, if coherency is of higher importance, the write-once cache should be chosen due to its report of the first modification of a block in the cache as opposed to none in the case of the write-back cache. This tradeoff of performance versus coherency is one of the issues a cache designer must face when designing a cache.

The data presented above shows that the increase of the degree of associativity harms the performance of the cache due to the increase in main memory accesses and the miss ratio. Using the miss rates calculated in Table 30, we can compare our results to those found in [Hennessy1996] by computing the average memory access time of the caches with varying degrees of associativity. Using Equation 12, the average memory access times were computed and reported in Table 31. In calculating the average memory access time, a hit time of 1 ns was used, and a miss penalty of 10 ns was used.
Table 31. Average Memory Access Time for Varying Degrees of Associativity

<table>
<thead>
<tr>
<th>Degree of Associativity</th>
<th>Write-Back</th>
<th>Write-Once</th>
<th>Write-Through</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way</td>
<td>4.33333757191</td>
<td>4.3333757191</td>
<td>4.33342658196</td>
</tr>
<tr>
<td>2-way</td>
<td>4.3340115052</td>
<td>4.3340115052</td>
<td>4.33347744485</td>
</tr>
<tr>
<td>4-way</td>
<td>4.3345201341</td>
<td>4.3345201341</td>
<td>4.3357917063</td>
</tr>
<tr>
<td>8-way</td>
<td>4.3355373919</td>
<td>4.3355373919</td>
<td>4.3378262219</td>
</tr>
<tr>
<td>16-way</td>
<td>4.3375719074</td>
<td>4.3375719074</td>
<td>4.33418952530</td>
</tr>
<tr>
<td>32-way</td>
<td>4.33416409385</td>
<td>4.33416409385</td>
<td>4.3350333152</td>
</tr>
<tr>
<td>64-way</td>
<td>4.33497790007</td>
<td>4.33497790007</td>
<td>4.3363094396</td>
</tr>
<tr>
<td>128-way</td>
<td>4.3360551252</td>
<td>4.3360551252</td>
<td>4.33988618850</td>
</tr>
<tr>
<td>256-way</td>
<td>4.33986073741</td>
<td>4.33986073741</td>
<td>4.34639661864</td>
</tr>
<tr>
<td>512-way</td>
<td>4.34637118719</td>
<td>4.34637118719</td>
<td>4.35941751820</td>
</tr>
</tbody>
</table>

As can be seen in Table 31, the miss rate increases for increasing degrees of associativity. This can be better seen graphically in Figure 7. One point to make about the results is that the write-back and write-once cache implementations yield the same miss rates, therefore in the figure, their lines overlap.

Average Memory Access Time Versus Degree of Associativity

Figure 7. Average Memory Access Time Versus Degree of Associativity

Another aspect of the performance that is affected by the increase in the degree of associativity is the hit-time. Although this parameter could not be measured numerically in our study, we did notice that the time it took to complete the execution of the sum test increased as the degree of associativity increased. For example, the simulation of the sum test on the 2-way set-associative cache took only 3 minutes, while that of the 512-way set-associative cache took 60 minutes. Though this time would vary depending on the processing power of the machine performing the simulations, there would still be a noticeable increase in
the simulation time as the degree of associativity is increased, thus increasing the average memory access time, and lowering the performance measure of the cache.

6.2 Reducing the Miss Rate by Using Victim Caches

Using a higher degree of associativity reduces the miss rate while affecting the clock cycle and the miss penalty. In this solution, the number of conflict misses is reduced without affecting the clock rate by adding a small, fully-associative cache between the main cache and main memory, as shown in Figure 8. This small cache is referred to as a victim cache and contains only blocks that are evicted from the cache, in order to give them a "second chance" to remain in the cache prior to being reported in main memory. On a cache miss, the items in the victim cache are checked to see if they contain the requested data prior to accessing the data from the higher level of memory. If the required data is found in the victim cache, a block is removed from the main cache, using whatever replacement algorithm that is implemented, and this evicted block and the block in the victim cache are swapped. Jouppi [Jouppi990] found that a victim cache of one to five entries was effective in reducing conflict misses, especially for small, direct-mapped data caches. According to Jouppi's results, a four-entry victim cache removed 20% to 95% of the conflict misses in a 4 KB direct-mapped data cache. Obviously, these results will vary depending upon the type of program being executed.

Figure 8. Placement of the Victim Cache in the Memory Hierarchy [Hennessy1996, p. 398]

6.3 Reducing the Miss Rate by Using Hardware Prefetching of Data

Another way to reduce the miss rate without affecting the clock rate or miss penalty is to prefetch the data before it is requested by the processor. On a cache miss, two data blocks are retrieved from main memory: the requested data block, and the one that is located immediately after it in main memory. The requested block is placed into the data cache, while the prefetched block is placed in the data stream buffer. If the requested block exists in the data stream buffer from the previous cache miss, the original cache request is
canceled, the requested data block is read from the data stream, and the next instruction in the program is executed. It is important to note that at all times there is only one data block in the data stream buffer, however, several data stream buffers can be used, each one prefetching a data block from a different offset from the requested data block. Jouppi found that a single data stream buffer caught 25% of the misses from a 4 KB direct-mapped cache [Jouppi1990]. He also found that using four data stream buffers increased the data hit rate to 43%. In a similar study, Palacharla and Kessler looked at a set of scientific programs and found that eight stream buffers could capture 50% to 70% of all misses (both data and instruction) from a processor using two 64 KB four-way set-associative caches, one for data and one for instructions [Palacharla1994]. It is important to note that the prefetching of data relies on utilizing memory bandwidth that otherwise would be unused, and can actually harm the performance if it interferes with demand misses [Hennessy1996, p. 402].

6.4 Reducing the Miss Rate by Using Compiler Optimizations

The previous optimization techniques involved making changes to the hardware in order to reduce the miss rate. In this optimization technique, the compiler notices code changes that would reduce the number of cache misses. For example, many programs use nested loops in which the data is accessed in a non-sequential manner [Hennessy1996, p. 406]. Simply exchanging the nesting of the loops can make the code access the data in the order that it is stored, thus preventing the program from jumping around the cache or main memory in search of data. This technique reduces the number of misses by improving upon the spatial locality of the data since by reordering the memory accesses, the data in the cache is used maximally before it is evicted from the cache.

As an example of this type of compiler optimization, consider the following code from [Hennessy1996, p. 407]:

\[
\begin{align*}
\text{for } (j=0; \ j<100; \ j=j+1) \\
\quad \text{for } (i=0; \ i<5000; \ i=i+1) \\
\quad \quad x[i][j] &= 2 \times x[i][j];
\end{align*}
\]

This code would skip through the blocks of memory in strides of 100 words since the loop with control variable \( i \) is nested inside the loop with control variable \( j \). This means that the code loops through memory down a column, accessing a word in each block, instead of across a row, accessing all of the words in one block. By rearranging the loops, the memory would be searched horizontally, thereby accessing all of the words in a data block sequentially. Therefore, an improved version of the code above would be the following [Hennessy1996, p. 407]:

83
for (i=0; j<5000; i=i+1)
    for (j=0; i<100; j=j+1)
        x[i][j]=2*x[i][j];

In this optimized code, all of the words in the cache block are accessed prior to continuing to the next cache block. This optimization improves the cache performance without affecting the number of instructions that were executed in the program, nor the hardware.

A similar code enhancement can be performed by fusing two loops that access the same array with the same loops, but perform different computations on the common data. By fusing the two loops into one, the data can be fetched into the cache and used repeatedly before being evicted from the cache. Fusion reduces the number of cache misses by improving upon the temporal locality principle of the memory hierarchy.

As an example of fusion, consider the code below from [Hennessy1996, p. 407]:

    for (i=0; i<N; i=i+1)
        for (j=0; j<N; j=j+1)
            a[i][j]=1/b[i][j]*c[i][j];
    for (i=0; i<N; i=i+1)
        for (j=0; j<N; j=j+1)
            d[i][j]=a[i][j]+c[i][j];

In this code, the misses associated with accessing arrays a and c will be counted twice, once in the first loop and again in the second loop. Since the two loops access the same data, one can fuse the two loops together and perform the two operations within the same loop. Note that the modification of the d array in the second loop depends on the value of a, therefore, the modification of the a array must occur first in the fused loop shown below, also from [Hennessy1996, p. 407]:

    for(i=0; i<N; i=i+1)
        for(j=0; j<N; j=j+1)
        {
            a[i][j]=1/b[i][j]*c[i][j];
            d[i][j]= a[i][j]+c[i][j];
        }
In this fused loop, the second statement uses the cache accesses of the first statement. In doing this, the number of cache misses is reduced by half since only one loop exists instead of two. Note that the final results of the a and d arrays reached using the two loops are the same as those reached using the fused loop.

6.5 Reducing the Miss Penalty by Giving Priority to Read Misses Over Writes

The above methods of improving the cache performance focused on reducing the miss rate, however, the cache performance formula assures us that improvements in the miss penalty can be just as beneficial as improvements in the miss rate [Hennessy1996, p. 411]. The first method of reducing the miss penalty is by giving priority to read misses over writes.

Most write-through caches on the market today come with write buffers, which store one or more words to be written to main memory simultaneously with the execution of the processor’s program. These words are written into the write buffer when they are evicted from the cache under the replacement strategy enforced by the cache. Instead of wasting valuable processor time by immediately writing these words to main memory, the words are stored in the write buffer so that: 1) they are given a second chance to be accessed by the cache under the rules of temporal locality, and 2) they can be written to main memory at the same time that the processor is executing its program. The addition of the write buffer is the most important improvement, provided that the buffer is of the correct size. However, the addition of write buffers may complicate things since they have the updated value of a location needed on a read-miss [Hennessy1996, p.380].

One way to avoid this problem is, on a read-miss, have the processor wait until all of the entries in the write buffer have been updated in main memory. This waiting period increases the read-miss penalty since the write buffer will almost always have something in it if a write-through cache implementation is used [Hennessy1996, p. 412]. In fact, the designers of the MIPS M/1000 estimated that waiting for a four-word write buffer to empty increases the average read-miss penalty by a factor of 1.5 [Hennessy1996, p. 412].

Another solution to the read-miss write buffer problem is to check the contents of the write buffer on a read-miss, to see if the required address resides in the write buffer. If it does, then the needed data is read from the write buffer, and the processor can continue its execution. If the requested address is not in the write buffer, then if the memory system is available, the read-miss should be allowed to continue, and thus fetch the needed address from main memory. By enforcing that there are no conflicts with the information
in the write buffer and the information requested, coherency is preserved between the cache and main memory.

The miss penalty can also be reduced with read-misses in a write-back cache. If a read-miss will replace a dirty memory block, instead of writing the dirty block to memory, and reading the requested block from main memory into the cache, the processor could copy the dirty block into the write buffer, then read the requested data from main memory, and then write the dirty block into main memory. In doing this, the CPU does not have to wait as long for the read to finish, as it would if the contents of the write buffer must be emptied prior to the read-miss accessing the required information from main memory [Hennessy1996, p.412].

6.6 Reducing the Miss Penalty By Using Sub-block Placement
In some cache designs, the address tags are either too large or too slow, thus degrading the performance of the cache, or preventing it from fitting onto one chip [Hennessy1996, p. 412]. One solution is to use larger blocks, which would reduce the tag storage—since now the tags are considered part of the enlarged block and can be contained within the cache—without decreasing the amount of information one can store in the cache, since just the length of the blocks increased, not the size of the cache.

The cache block size is the unit of transfer between main memory and the cache. Changes to the block size affect the miss ratio, since as the block size increases from very small to very large, the miss ratio will initially decrease because a miss will fetch more data at one time. With more data in the cache, the chances of the data required residing in the cache increase, thus decreasing the miss rate.

As the cache block size is further increased, the miss ratio will start to increase since the probability of using the newly fetched data in the near future becomes less than the probability of using the data that was just evicted from the cache to make room for the new block [Smith1993]. As the cache block size gets even larger, a cache miss ties up the memory interface for longer periods of time, since more information must be read into the cache, and more must be read out and placed into main memory—provided that the evicted block is dirty. By hogging the cache to memory interface, other processors in a multiprocessor may be locked out of the shared memory for long periods of time, and hence may cause long, unwanted idle time within the multiprocessor [Smith1993]. More on multiprocessors are discussed on page 95.

Returning to the uniprocessor model, the increase in the cache block size is likely to increase the miss penalty. Therefore, making the cache blocks larger is a bad decision. Another solution is to use sub-block
**placement** in which a cache block is divided into several smaller blocks, called **sub-blocks**, and a valid bit is associated with each of these sub-blocks. These valid bits specify whether a part of the cache block is valid or not. Thus, a matching address tag does not guarantee a hit to the cache since the requested sub-block may be invalid. Using sub-block placement, only one sub-block needs to be read on a read-miss, instead of an entire block, thus decreasing the miss penalty. In addition, the use of the valid bits decreases the number of address tags required, as can be seen in Figure 9. The use of the valid bits can be thought of as an extra level of addressing beyond the address tag [Hennessy1996, p. 413]. If sub-block placement were not used, each of the 16 blocks represented in Figure 9 would require a separate block entry into the cache, with a full address tag associated with each one. Thus 16 address tags would be required instead of the four that are used in the sub-block placement method. In this figure, one can tell that all of the sub-blocks in the first block, which contains addresses 100-103, are valid since all of the valid bits are on. In the last block, which contains addresses 204-207, none of the valid bits are on, therefore the entire block is invalid. In the second block, a read of addresses 300 or 301 will result in hits, but reads of addresses 302 or 303 will result in misses since the valid bits are off.

![Sub-block Placement Diagram](image)

**Figure 9. The Sub-block Placement Strategy [Hennessy1996, p. 413]**

This sub-block placement method was primarily created to reduce the long miss penalty of large blocks, since now only a small part of the full block needs to be read into the cache instead of the entire block. It is important to note that this scheme does not work with write-back caches, since in the write-back cache, the only valid copy of the data may reside in the cache, and if this cache value is modified, the modification will not be reported to main memory until the block is evicted from the cache. Since this modified block
may get overwritten while it is in the cache, due to a write from another instruction, the previous modified value will be lost, and erroneous results may occur in the program being executed on the processor.

6.7 Reducing the Miss Penalty by Using Early Restart and Critical Word First Methods
The previous two methods of reducing the miss penalty required extra hardware to be added to the existing cache architecture, however, these two methods of reducing the miss penalty do so without the cost of extra hardware. Both of these methods rely on the observation that the CPU needs just one word of the block at a time. Therefore, there is no need to wait until the full block is loaded into cache before sending the requested word to the cache and waking up the CPU [Hennessy1996, p. 412].

The first of these strategies is called early restart and sends the requested word to the CPU as soon as it arrives into the cache so that the CPU may continue with its execution as soon as possible. Thus, the entire block is read into the cache in the order that it is received from main memory, and when the requested word is loaded into the cache, it is sent on to the processor, and the CPU continues its execution while reading in the rest of the block into the cache.

The second of these strategies, called critical word first (also known as wrapped fetch and requested word first), requests that the required word be read into the cache first, sent to the CPU, and then the rest of the block be read into the cache—filling in the gaps in order from the least significant word to the most significant word—while the CPU is executing its program. Both of these techniques only benefit cache designs with very large cache blocks, since the benefit is low unless the cache blocks are large [Hennessy1996, p. 413]. In addition, these techniques depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been read into the cache.

6.8 Reducing the Miss Penalty by Using Second Level Caches
The performance gap between the processors and main memory may lead a designer to ask the question “Should I make the cache faster to keep pace with the speed of the CPUs, or should I make the cache larger to overcome the widening performance gap between the CPUs and main memory?” [Hennessy1996, p. 416] The approach discussed here does both. By adding a second-level of cache to the existing memory hierarchy, the first-level cache can be small enough to match the clock cycle time of the fast CPU, while the second-level cache can be large enough to capture many accesses that would otherwise go to main memory, thereby lessening the effective miss penalty [Hennessy1996, p. 416].

88
In analyzing the effect of a second level of cache, some basic definitions need to be attained. First of all, the average memory access time is defined as the following:

$$
\text{Average memory access time}_{\text{two-level cache}} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}
$$


In Equation 14, $L1$ and $L2$ refer to the first-level and second-level caches, respectively. In addition, the miss penalty of level one is expanded upon to form the following:

$$
\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}
$$

**Equation 15. Miss Penalty of the First Level of Cache [Hennessy1996, p. 417]**

Combining Equation 14 and Equation 15 yields the following:

$$
\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})
$$


As can be seen in the above equations, the second-level miss rate is measured by the number of misses that occurred in the first-level cache. In addition to the above formulae, two terms are introduced in order to better explain the benefits of a second-level cache architecture. The local miss rate is the number of misses in the cache divided by the total number of memory accesses to the cache, or $\text{Miss rate}_{L2}$ from Equation 16 for the second-level cache [Hennessy1996, p. 417]. Another important, and related, term is the global miss rate, which is the number of misses in the cache divided by the total number of memory accesses generated by the CPU, which is $\text{Miss rate}_{L1} \times \text{Miss rate}_{L2}$ from Equation 16 for the second-level cache [Hennessy1996, p. 417]. To help explain the differences between these two terms, one can say that the global miss rate is a measure of the fraction of memory accesses that must go all the way to main memory, while the local miss rate is just the number of misses in a particular level of the two-level cache architecture.

A couple of things can be noted from the two terms mentioned above. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache [Hennessy1996, p. 418]. This fact is true because all
misses in the second-level cache must go all the way to main memory. Thus the miss rate in the second-level cache, or the local miss rate of the second-level cache, is equivalent to the global miss rate.

The second insight is that the local cache miss rate is not a good measure of secondary caches since it is a function of the miss rate of the first-level cache and hence will vary if parameters within the first-level cache are changed [Hennessy1996, p. 419]. Therefore, the global miss rate is a better performance measurement when evaluating the performance of second-level caches since it is a measure of the number of memory accesses that go straight to main memory, and does depend on how the two levels of cache are designed.

Now that the terms are better understood, we can begin to understand the benefits and weaknesses of second-level caches. In comparing the first-level cache with the second-level, the main difference is in the speed. Since the first-level cache is closest to the CPU, it only affects the clock rate of the CPU. Likewise, the speed of the second-level cache only affects the miss penalty of the first-level cache [Hennessy1996, p. 419]. These speed concepts are important in determining the size of the two levels of the cache, which is discussed next.

The size of the second-level cache should be much bigger than the first, however, if the second-level cache is only slightly larger than the first, the miss rate of the entire cache will be high, since a miss in the first-level will generally cause a miss in the second-level of cache and thus enforce the most penalty on the memory reference. By the same token, if the second-level cache is too large, then the global miss rate will be approximately equivalent to that of a single-level cache of the same size, and thus no benefits of a second-level cache will be observed. This observation leads designers to build huge second-level caches—larger than the size of main memory in older computers [Hennessy1996, p. 419]. This large sized second-level cache implies that the second-level cache will have practically no capacity misses since approximately all of the data needed will reside in the second-level cache.

Another consideration in the design of a second-level cache is in the degree of associativity. A high degree of associativity has a small impact on the second-level hit time since much of the average access time is due to the misses from the second-level cache [Hennessy1996, p. 420]. In very large caches, many capacity misses have been eliminated by allowing more blocks to be stored in the cache and many conflict misses have been eliminated by distributing the data over more blocks. Thus, when designing a large second-level cache using a direct-mapped mapping strategy, one must be concerned with the percentage of conflict misses, or how many of the cache misses are due to conflicts arising from the use of the direct-mapped mapping strategy [Hennessy1996, p. 420]. The percentage of conflict misses is important since if a high
percentage of the misses are due to conflict misses, a direct-mapped cache should not be used, since the blocks are restricted in their placement within the cache. A fully-associative mapping strategy should be used in this case, so that the blocks are free to be mapped anywhere within the cache.

In addition to the cache size and the degree of associativity, increasing the cache block size of the second-level cache can increase conflict misses within small caches since there may not be enough places to put data, thus increasing the miss-rate [Hennessy1996, p.420]. Since the second-level cache is typically huge, no capacity misses should occur. Therefore, second-level cache block sizes of 64, 128, and 256 bytes are being used in today’s designs [Hennessy1996 p. 421].

Another key consideration in the design of second-level caches is whether or not to obey the inclusion principle of the memory hierarchy and have the second-level cache contain all of the data that appears in the first-level cache. This is known as the multi-level inclusion property of second-level caches and is desirable because consistency between the caches in a multiprocessor can be determined just by checking the second-level cache [Hennessy1996, p. 421]. However, the multi-level inclusion property does have the drawback that all first-level blocks that map onto the second-level block must be replaced when the second-level cache block is either evicted or invalidated. This replacement causes a slightly higher first-level miss rate and may also cause some unneeded cache invalidations [Hennessy1996, p. 421].

To summarize the design issues of second-level caches, there are many fewer hits in the second-level cache than in the first-level, so the emphasis on the design of the second-level cache shifts to designing it to achieve fewer misses. This insight leads to larger caches with higher associativity and larger blocks, as discussed above [Hennessy1996, p. 422]. Some typical values for a second-level cache are shown in Table 32.

<table>
<thead>
<tr>
<th>Block size</th>
<th>32-256 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit time</td>
<td>6-15 clock cycles</td>
</tr>
<tr>
<td>Miss penalty</td>
<td>30-2000 clock cycles</td>
</tr>
<tr>
<td>Local miss rate</td>
<td>15%-30%</td>
</tr>
<tr>
<td>Cache size</td>
<td>256 KB - 16 MB</td>
</tr>
<tr>
<td>Block Placement</td>
<td>Direct-mapped or set-associative mapped</td>
</tr>
<tr>
<td>Block Replacement</td>
<td>Random</td>
</tr>
<tr>
<td>Write Strategy</td>
<td>Write-back</td>
</tr>
</tbody>
</table>

Table 32. Typical Values and Parameters of a Second-Level Cache [Hennessy1996, p. 471]

To conclude the discussion of two-level caches, Frank Casilio in his MS Thesis [Casio1998] proposes a second-level of cache that only stores shared data. He claims that this second level of cache removes the
need for a cache coherence protocol at the expense of adding some hardware, and making minute changes to the CPU logic.

6.9 Reducing the Hit Time By Using a Small and Simple Cache Design

In this section, we address improvements to the cache performance by decreasing the hit time. The hit time affects the clock rate of the processor, so a fast hit time is higher in importance than the average memory access time formula because it improves all of the parameters affecting the cache performance [Hennessy1996, p. 422].

The first hit time reduction comes from the notion that using the index portion of the address to read the tag in the cache, and then to compare it to the desired address is a time-consuming process [Hennessy1996, p. 422]. A small cache is suggested so that fewer checks of address tags are required on each memory reference, and also because smaller hardware is generally faster. In addition, one wants to make the cache small enough to fit on the same chip as the processor so that no extra access time is spent going off the chip to access data from the cache. To meet this suggestion, some of today’s designs keep the address tags on-chip and the actual data off-chip [Hennessy1996, p. 422]. Doing this promises a fast tag check along with a higher cache capacity, since all of the data is off-chip.

A similar suggestion is to keep the design of the cache as simple as possible by using the direct-mapped mapping strategy. The main benefit of the direct-mapped strategy is that the designer can overlap the address tag checking with the actual retrieval and transmission of the data [Hennessy1996, p. 422]. The direct-mapped strategy reduces the hit time by not requiring the cache to check every address tag in the cache to see if it matches the required address, as would be required in the fully-associative mapped cache. This greatly reduces the hit time and suggests that in order to have a fast cache, one should keep it small and simple.

6.10 Cache Optimization Summary

The techniques presented in the previous sections impact other aspects of the average memory access time formula and memory hierarchy complexity, as well as the miss rate, miss penalty, and hit time. Table 33 explains the impact the reduction techniques discussed above have on the miss rate, miss penalty, hit time, and overall hardware complexity. In the table, a + indicates a reduction or benefit to that particular aspect, a - indicates a harm, and a blank box indicates that the technique discussed does not affect that particular aspect of the memory access time equation. In addition, the hardware complexity is measured subjectively, with a 0 being the easiest to implement and a 3 being the hardest.
As can be seen in Table 33, none of the proposed suggestions for improving the cache performance help more than one category, nor do any of them help all three aspects of the average memory access time equation. In addition, one can see the tradeoffs that occur by helping one aspect and harming another. For example, as discussed on page 76, increasing the degree of associativity of the cache improves the miss rate by allowing more blocks to reside in the cache, however, the hit time is reduced since now there are more blocks that the cache must check to see if the address tags of the block and the requested address match.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss Rate</th>
<th>Miss Penalty</th>
<th>Hit Time</th>
<th>Hardware Complexity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td></td>
<td>0</td>
<td>0</td>
<td>Trivial; RS/6000 550 uses 128 bytes</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td></td>
<td>1</td>
<td>1</td>
<td>e.g. MIPS R10000 is 4-way</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td>2</td>
<td>2</td>
<td>Similar technique used in HP 7200</td>
</tr>
<tr>
<td>Hardware Prefetching of Data</td>
<td>+</td>
<td></td>
<td>2</td>
<td>2</td>
<td>Data are harder to prefetch than instructions; tried in Alpha 21064</td>
</tr>
<tr>
<td>Compiler Techniques to Reduce Cache Misses</td>
<td>+</td>
<td></td>
<td>0</td>
<td>0</td>
<td>Software is a challenge; some machines give compiler option</td>
</tr>
<tr>
<td>Giving Priority to Read-Misses Over Writes</td>
<td>+</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Trivial for uniprocessors and widely used</td>
</tr>
<tr>
<td>Sub-block Placement</td>
<td>+</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Used primarily to reduce tags</td>
</tr>
<tr>
<td>Early Restart and Critical Word First</td>
<td>+</td>
<td></td>
<td>2</td>
<td>2</td>
<td>Used in MIPS R10000, IBM 620</td>
</tr>
<tr>
<td>Second-Level Caches</td>
<td>+</td>
<td></td>
<td>2</td>
<td>2</td>
<td>Costly hardware; harder if block size of level 1 does not equal block size of level 2; widely used</td>
</tr>
<tr>
<td>Small and Simple Cache Designs</td>
<td>+</td>
<td></td>
<td>0</td>
<td>0</td>
<td>Trivial; widely used</td>
</tr>
</tbody>
</table>


The hardware complexity is a measure of how easy, or difficult it is to implement the particular technique in hardware. In most cases, the particular technique has already been implemented, and examples of its use are included in the comment section of the table. In other cases, reasons are suggested that explain why the
technique has not yet been implemented. Hopefully, future research in the design of cache memories will provide even more ways to improve upon the performance of the cache by reducing the miss rate, miss penalty, and hit time without significantly increasing the complexity of the hardware implementation.
7 Introduction to Shared-Memory Multiprocessors

A shared-memory multiprocessor consists of at least two independent processor modules executing either a small task of a large program, or completely independent programs. All of the processors make references to instructions and data that reside in a main memory module that all the processors share. There is contention in using the shared-memory resource, and because of that, the average latency in accessing shared memory tends to be longer than if each processor had its own local memory unit [Tomasevic1993]. Therefore, local caches are often added to each processor in shared-memory multiprocessors.

These private, local caches are added to each processor and are used to satisfy most of the local memory references, eliminating the need to access the shared memory unit. Therefore, the addition of private local caches in a shared-memory multiprocessor is effective in terms of increasing the memory and communication bandwidth. However, the addition of private caches introduces the cache coherence problem, explained in the next section.

7.1 The Cache Coherence Problem

According to Tomasevic and Milutinovic in [Tomasevic1993] and Censier and Feutier in [Dubois1988], a system is coherent if every read by any processor always returns the value produced by the last previous write, no matter which processor performed the write. In addition, it can be assumed that no data coherence problems exist in multiprocessors that maintain only a single copy of the data. To see this point, Dubois, in [Dubois1988], proposes an example in which there is a shared-memory multiprocessor. None of the processors have private caches, thus all data references, both reads and writes, go directly to the shared main memory. If the reads, writes, and read-modify-write cycles are atomic, meaning that the processor issuing the operation has exclusive access to that data block, and simultaneous accesses to the same data element are serialized by the hardware, the data elements can be accessed and modified in indivisible operations, and hence each access to an element will result in the most up-to-date value being retrieved. Since the most up-to-date value is always retrieved, the system is coherent, as defined by Tomasevic, Milutinovic, Censier, and Feutier.

The cache coherence problem occurs in a shared-memory multiprocessor when two or more processors each have a local copy of a data value, and one of the processors changes the data value within its local cache, as part of an instruction that is executed in its program. The data value in the other processors' caches are no longer valid, and in most cases, main memory is also inconsistent with the caches. Since the
caches are inconsistent with each other, and with memory, in terms of which unit has the valid data, a coherency problem exists.

A visual description of the cache coherency problem is shown in Table 34. In time frame 0, the data value of X is initialized to ‘1’ and is stored in main memory. Neither cache has yet made a reference to X. In time frame 1, CPU A reads the value of X from main memory and sees that its value is ‘1’. By reading the value from main memory, it is assumed that a cache miss occurred in CPU A’s cache, and hence a copy of X now resides in CPU A’s cache. In time frame 2, CPU B reads the value of X from main memory and also finds that its value is ‘1’. By reading the value from main memory, it is assumed that a cache miss occurred in CPU B’s cache as well, and hence a copy of X now resides in both CPU A’s and CPU B’s caches. Since neither cache has yet modified the value of X, both caches and main memory are consistent with each other.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache Contents of CPU A</th>
<th>Cache Contents of CPU B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 34. Visual Description of the Cache Coherency Problem [Hennessy1996, p. 655]

In time frame 3, CPU A executes a write to the value of X, thus modifying its value to ‘0’. Since CPU A’s cache uses the write-through protocol, main memory is also notified of this change to the value of X, and hence main memory and CPU A’s cache are consistent with each other. However, the write to X was local to CPU A, and hence was not observed by CPU B. Unless the modification of the value of X by CPU A is reported to CPU B, any reference to X by CPU B will cause erroneous results in future instructions executed by CPU B.

7.2 The General Categories of Solutions to the Cache Coherency Problem

Many solutions to the cache coherence problem have been suggested, both in hardware and in software. The efficiency of these solutions depends upon the system architecture parameters, and especially on the parallel programming characteristics of the system [Tomasevic1993]. The majority of these solutions are hardware based and deal with the coherency problem by dynamically recognizing inconsistency conditions.
for shared data at run time. These hardware solutions promise better performance, since the coherence overhead is generated only when actual sharing of data takes place. Also, hardware solutions are totally transparent to the programmer and user [Tomasevic1993].

Dubois proposes some general solutions to the cache coherency problem in [Dubois1988]. These general solutions are discussed in the following sections in their order of complexity, from easiest to hardest to implement.

### 7.2.1 Disallowing Private Caches

The first, and easiest, solution to the cache coherency problem is to not allow private caches to exist in the multiprocessor, and instead have all memory references access a shared cache. This requires all the processors to be connected to this shared cache by some form of processor-to-memory network. Although this solution provides cache coherence in terms of the data, and is transparent to the user, the memory conflicts are not reduced, since now there are several processors all trying to access the same data from the same place. Since there are several caches trying to read data from the same location, contention may arise when two processors try to access the same data block at the same time. Due to this scenario, an arbiter must be added to the cache architecture between the processors and the shared cache. The goal of the arbiter is to determine which processor obtains access to the cache. Since the arbiter lies in between the processors and the cache, this technique violates the rule that the processors and the cache be as close as possible. Despite this disadvantage, the memory access latency will be reduced, since now all the caches will be able to access the same data from the same shared cache.

### 7.2.2 Allowing Private Caches

The next, and somewhat more difficult, solution is to allow the existence of private caches, but to disallow the cache to contain shared writeable data, such as locks and semaphores, that are used to control critical sections in which processors modify these shared data values. The classification of which data values are cacheable and which are not is left as a task for the programmer, and thus makes this solution non-transparent to the user.

### 7.2.3 Non-Cacheable Shared Writeable Data

In this category of solutions, an improvement is made to the previous solution of making all shared data non-cacheable. If all the shared data must be accessed in main memory, the performance of the processor will be degraded due to the long access time required to access main memory. However, if just the locks,
semaphores, and other variables that are used to protect the critical section where the shared data is modified are declared as non-cacheable, then the memory access time will be reduced, thus improving the processor's performance. However, to ensure that consistency is still preserved, all the data modified while a processor is in its critical section must be invalidated in the cache when the critical section is exited, termed a cache flush. This flushing needs to be performed so that no stale data remains in the cache when the critical section is next entered. This constant flushing of the cache after every exit from a critical section degrades the processor performance by increasing the time required to perform a cache write. A possible improvement to the flushing of the entire cache is to tag the data that is accessed by the critical section being entered, so that when the critical section is exited, only those data values need to be flushed from the cache.

7.2.4 Allowing Shared Writeable Data

In this category of solutions, shared writeable data are allowed to exist in the cache, however in order to maintain their coherency, a centralized global table is used, which stores the status of each block in the cache (i.e. exclusively owned, shared, clean, or dirty) and information as to all the caches that contain copies of the data block. A cache interrogate signal is associated with each block in the cache, and any time a cache wishes to modify the contents of a data block, it must first acquire exclusive ownership of the block by propagating its request to all the other caches. This is done by the centralized global table, which sends a cache interrogate signal to all the caches that contain the requested block, in order to invalidate all other copies of the block about to be modified. Modification is then performed and all processors resume their respective programs.

The main disadvantage of this solution is the fact that the centralized global table must be accessed by all the processors, which might cause a communication bottleneck as the number of processors is increased. In addition, as the processors become faster and faster, the memory access latency becomes more and more dependent on the time required to access the centralized global table.

7.2.5 Bus-Oriented Multiprocessors

In this category of solutions to the cache coherency problem, the concept of the centralized global table is expanded upon by taking advantage of the bus that connects all processors together, and distributing the centralized global table among the processors. Cache consistency is maintained in this strategy by maintaining a bus watching mechanism, or snoopy cache controller which watches the communication bus for all actions affecting shared data. If there is a write to a shared data block, all copies of the block that
reside in other caches are either invalidated or updated, depending upon the snoopy cache coherence protocol implemented by the multiprocessor.

This class of solutions is by far the most complicated to implement since the bus interface now must watch the bus for all transactions that take place across it. The main disadvantage of this strategy is that since it uses the broadcast features of the bus, the number of processors that can be used in a multiprocessor environment using a bus-based cache coherence solution is limited by the bus bandwidth. Several snoopy cache coherence protocols are discussed on page 100.

7.2.6 Examples of Cache Coherence Solutions in Existing Multiprocessors

Table 35 shows how some of the multiprocessors on the market today are designed in order to solve the cache coherence problem. Note that the Denelcor HEP was built without allowing caches to exist, therefore no solution to the cache coherence problem is needed.

<table>
<thead>
<tr>
<th>Multiprocessor</th>
<th>Number of Processors</th>
<th>CPU Architecture</th>
<th>Cache</th>
<th>Coherence Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 3081</td>
<td>≤ 4</td>
<td>IBM 370</td>
<td>Write-back</td>
<td>Central table</td>
</tr>
<tr>
<td>Synapse N+1</td>
<td>≤ 32</td>
<td>Motorola 68000</td>
<td>Write-back</td>
<td>Distributed table/ bus watching</td>
</tr>
<tr>
<td>Denelcor HEP</td>
<td>Hundreds</td>
<td>Custom</td>
<td>No cache</td>
<td>No shared writeable data in cache</td>
</tr>
<tr>
<td>IBM RP3</td>
<td>Hundreds</td>
<td>IBM ROMP</td>
<td>Write-back</td>
<td>No shared writeable data in cache</td>
</tr>
<tr>
<td>NYU Ultracomputer</td>
<td>Hundreds</td>
<td></td>
<td>Write-back</td>
<td>No shared writeable data in cache</td>
</tr>
<tr>
<td>Encore Multimax</td>
<td>≤ 20</td>
<td>National Semiconductor 32032</td>
<td>Write-through (two processors share each cache)</td>
<td>Bus watching</td>
</tr>
<tr>
<td>Sequent Balance 8000</td>
<td>≤ 12</td>
<td>National Semiconductor 32032</td>
<td>Write-through</td>
<td>Bus watching</td>
</tr>
</tbody>
</table>

Table 35. Example Cache Coherence Solution in Existing Multiprocessors [Dubois1988]
8 Snoopy Cache Coherence Protocols for Multiprocessors

A snoopy protocol requires that the responsibility of maintaining cache coherence is distributed among the local caches. In addition, the caches must make any necessary changes to the status of its blocks during the execution of the various programs throughout the processors in the network [Tomasevic 1993].

The general design of a snoopy cache coherence protocol consists of a local controller for each processor that monitors, or “snoops”, on the shared bus. All processors must broadcast any memory request that could potentially modify the coherence state of shared blocks. The local controller then determines if the address being modified is contained within its corresponding cache. If it is, the cache block is invalidated, or other appropriate actions are taken.

In the following sections, several snoopy cache coherence protocols are discussed, and their benefits and weaknesses are also addressed. These protocols were not implemented in VHDL for this thesis, however, as discussed in the future work section later in this thesis, the VHDL code created in this thesis could be used to represent the individual processors in the multiprocessor system.

8.1 Write Through Protocol
In the write through protocol, all cache updates are written back to main memory. The other caches must invalidate any entry(ies) that contain a matching address to the one being modified. This is considered the simplest protocol, however, it is not suitable for systems consisting of more than a few processors, since substantial traffic could saturate the bus. Also, extra misses will be required in order to reload the cache blocks that have been previously invalidated once they are accessed again [Thacker 1988]. In addition, the processor performance may be degraded on the writes if it must wait until all of the writes are finished before continuing with the execution of its process [Katz 1985].

8.2 Write Back Protocol
In the write back protocol, the contents of the cache block are written to main memory only when the block is requested by another CPU, and its contents have been modified. If the contents of the cache block have not been modified locally, thus indicating that the block is clean, no write to main memory is required since the cache and main memory are coherent. This protocol has been proven superior to the write-
through protocol since only dirty blocks must be written to main memory when they are requested by another cache, instead of every block [Thacker1988].

Despite the enhanced performance, the write back protocol also suffers from a drawback in that it does not entirely solve the cache coherence problem. When a local CPU writes to blocks in its cache, the changes are not observed by the other processors until after the cache block is evicted. Hence, the other processors do not know that the address has been modified until after the contents of the cache block are written back to main memory. This problem is often solved by requiring the caches that want to write to a shared cache block to obtain exclusive ownership of the block, as illustrated in the Berkeley Ownership protocol to be discussed on page 111 [Katz1985].

8.3 Comparing Write Back to Write Through

Goodman [Goodman1983] compared the average bus traffic using the write back protocol to that of the write through protocol. He discovered that as the hit ratio of the cache approaches 100%, the write back protocol will require no bus traffic since all of the needed blocks are currently in the cache. However, the write through protocol requires at least one bus cycle for each read operation, since the previous write may have invalidated the copy in the local cache. He also proved that using the write back protocol instead of the write through protocol could reduce the bus traffic by 50%, however, the write back has more severe coherency problems than write through, as mentioned above, since even main memory does not always contain the current value of a particular data block.

8.4 Write Once Protocol

As a result of his study on the write through and write back protocols, John Goodman [Goodman1983] proposed the write once protocol which combines the write through and write back protocols into one, in order to get the benefits of both. The write once protocol requires the first write to any cache entry to be written through to main memory, using the write through protocol. This forces all other caches to invalidate their copy of the cache block, and ensures that the processor writing to the cache block contains the only legally cached, valid copy of the data block [Katz1985]. Any subsequent write to that cache block will be done locally in the cache, but the modifications will only be written to main memory after the block is requested by another processor, or when the block is evicted from the cache. Hence the use of the write back protocol.

This protocol is more difficult to implement than the write-through protocol because the snoop controller must service all external reads of the cache and perform all of the invalidations to cache entries within its
cache. Another disadvantage of this protocol is that it incurs an initial write to main memory even if the block is not shared by other processors. If the block is not shared when it is modified, there is no need to update its value in main memory until it is requested by another processor [Katz1985].

To implement the write-once protocol, two bits are associated with each cache block. The two bits distinguish among the four states that a cache block may reside in: invalid, valid, reserved, and dirty. In the invalid state, the cache block contains no data. In the valid state, the cache block contains data which has been read from the main memory and has not yet been modified. Hence, the cache and main memory are consistent with respect to this block. The reserved state indicates that the block has been locally modified exactly once since it has been brought into the cache, and the results of the modification have been reported to main memory. Thus, once again, the cache and main memory are consistent with respect to this block. The final state, dirty, indicates that the cache block has been modified more than once since it has been brought into the cache, and the latest change has not yet been reported to main memory. In this state, the cache and main memory are inconsistent with respect to this cache block.

During the course of the execution of programs on the processors in the multiprocessor system, the snoopers of each cache constantly monitor the bus to determine if the address(es) being modified match those contained within its cache. If a match is found on a write operation, the cache controller invalidates its local cache block containing the matching address. If a match is found on a read operation, nothing is done unless the matching cache block is either reserved or dirty. If the local cache block is reserved, the cache block state is changed to valid, since the data block is now shared between two caches. If the local cache block state is dirty, the local system inhibits main memory from supplying the data value to the requesting processor, and the local cache supplies the data to the requesting processor itself, since it has the most up-to-date copy. On the same bus access, or immediately after, the updated value must be written back to main memory, so that coherency can be re-achieved with respect to this block. After this process is completed, all of the caches that share this data block, as well as main memory, have the most up-to-date value of the data block, so the local cache block state is changed to valid. The above state transitions are depicted in Figure 10.

Coherency is preserved in the write once protocol since on a write, all other caches that contain a copy of the block being modified must invalidate their copy Therefore, the cache performing the write is guaranteed that it contains the only valid copy, except for the one in main memory, which will receive the result of the modified data, since the write was the first to this block while it resided in the cache [Goodman1983]. The local cache block is now marked reserved, and if it is later evicted from this local cache, without another write to it, no additional write to main memory is needed. If another write does
occur, the block will be marked dirty, and the write back protocol will be employed, preventing main memory from receiving another modified value until after the cache block is evicted from the cache, or another processor requests this block.

Figure 10. Write Once Protocol [Hwang1993 p. 354]

8.5 Comparing Write Once to Write Back and Write Through

Goodman compared his protocol to both the write through and write back protocols. According to Goodman [Goodman1983], a multiprocessor system consisting of three PDP-11 computers, each with a four-way set associative, 2048 byte cache and cache block size of 32 bytes resulted in average bus traffic measurements of 30.768\%, 17.55\%, and 17.38\% for the write through, write back, and write once protocols, respectively. Thus, Goodman's write once protocol proves to outperform both the write through and write back protocols.
8.6 *Papa’s Protocol*

Mark Papamarcos proposed a snoopy cache coherency protocol whose goal is to reduce bus traffic and thus decrease the wait time that a processor must wait prior to accessing the bus. This reduction in wait time and bus traffic will thus increase the overall processor utilization [Papamarcos1984]. Since Papamarcos does not specifically give a name for his protocol, it will be referred to as “Papa’s protocol” throughout this paper. It is important to note that Papa’s protocol is also expandable and closely tied to the miss ratio and amount of sharing among the processors within the multiprocessor.

Papa’s protocol requires that two status bits be associated with each cache block. The first bit denotes whether the cache block is shared and the second indicates if the block has been modified locally. With these two bits, four states can be associated with each cache block: invalid, exclusive-unmodified, exclusive modified, and shared-unmodified. Note that a block can never be both shared and modified, since doing so would render the data value inconsistent between two or more caches. Hence, the shared-modified state is referred to as an invalid state throughout the protocol.

The invalid state indicates that the block contains no valid data. The exclusive-unmodified state indicates that no other cache has this block, and that the data in the block is consistent with that in main memory (i.e. its contents have not been modified in the cache). The exclusive-modified state indicates that no other cache has this block and that the data in the block is inconsistent with that in main memory since the data has been locally modified. The shared-unmodified state indicates that some other cache(s) may have this block and that the data in the block is consistent with that in main memory (i.e. its contents have not been modified in the cache, nor any cache that contains this block).

Based on the above states, Papamarcos realized that an evicted cache block only needs to be written to main memory if its status is exclusive-modified, since no other state contains a modified block. If the block is both shared and modified, it becomes invalid, and all caches receive the valid data on the next read of the cache address. In addition, in order to simplify the implementation of the protocol, if a cache block is initially marked as shared-unmodified, and later on, the other caches containing the cache block evict the block from their respective caches so that only one cache now contains the block, this cache block will remain in the shared-unmodified state in the cache that still contains its value. If the data value of this block is modified later on in the program, an extra write to main memory is required because the block is marked as shared and not exclusive. This extra write is required even though the block is no longer shared. These state transitions and the others can be seen in Figure 11.
8.7 Read Broadcast Protocol

The Read Broadcast (RB) protocol, presented by Larry Rudolph et al [Rudolph1984], is based on the write once protocol, but uses the bus broadcast capabilities more efficiently for both data and event broadcasting.

The RB protocol, and the RWB protocol discussed in the next section, both make use of the following key assumptions [Rudolph1984]:

- each data item is referenced more often with a read operation than a write. This indicates that it may be desirable to increase the overhead of a write in order to optimize the performance of the reads.
- references to local data and to read-only shared data are more frequent than references to readable or writeable shared data.
- many shared variables act like local variables for moderately long periods of time.
- there is a logically single bus connecting the N processing elements and I/O devices with the shared memory.
- there is a bus arbitrator that allocates access to the bus so that only one processor is given access to the bus at any given time. All other processors must wait until the bus is free again.
• a cache is associated with each processing element (PE) and communicates both with the PE and with the shared bus.
• the caches can “listen” to the bus activity and detect the referenced address, the activity (either a read or write), and the data value being transmitted.
• the bus cycle time is no faster than the cache cycle time. Thus, each cache has time to monitor the bus and take appropriate action before the next bus cycle. Similarly, the PE cycle time should be no faster than the cache cycle time so that the PE does not spend valuable time waiting for the cache to respond to its request.
• a cache has the ability to interrupt (or kill) the current bus activity in order to replace it with one of its own. The cache is fast enough to first observe a bus action and then to interrupt it.
• both set size and bus size are assumed to be one word.

The main feature of the RB protocol is that values fetched in response to certain processor reads are broadcast to all of the caches, hence the name “Read Broadcast.” There are two status bits associated with each cache block, thus specifying one of the following three states: readable [R], invalid [I], or local [L]. The fourth state is not used.

The readable state indicates that the contents of the cache block are valid and consistent with memory. The invalid state indicates that the cache block does not contain valid data. A read operation of this cache block will cause the data to be fetched from main memory, and a write will cause the contents of the cache block to be written to main memory and, at the same time, broadcast to all other caches. The local state indicates that the data in the cache block can be read or written to locally, causing no bus activity. A write to a cache block in the local state will cause the data to be inconsistent with that of main memory, and with that of the other caches.

There are two possible configurations that a cache block may be in if the RB protocol is used. The local configuration indicates that a variable X that is local to PEi will be in state L in cache i and in state l in any other cache containing variable X. This configuration allows cache i to have exclusive ownership of X, in order to be able to modify the contents of it. The shared configuration implies that the shared, read-only variable Y is in state R in all caches that contain it. This allows any cache to read the data value associated with Y and be ensured that it is receiving the most up-to-date value.

With the above states and configurations, the following scenarios can happen [Rudolph1984]:

106
let X be in the shared configuration. A read simply fetches the cached value, causing no bus activity. A write by PEi to variable X causes the value to be updated in cache i, as well as a broadcast of the bus write. The bus write updates main memory and at the same time causes all other matching cache blocks to change into the state I.

let X be in a local configuration with cache i in the local state. Two different sub-scenarios can occur:

- X is referenced by PEi. A read of X by PEi simply fetches the cached value and a write to X just updates the cached value. No bus activity is generated for either a read or a write.
- X is referenced by PEj, where \( j \neq i \). A write updates the cached value in cache j, changes its state to I, and generates a bus write to the cache block containing X with the new value. This bus write causes all other caches to be set to state I. If PEj reads X, since X is in state I in cache j, a bus read is issued which is “seen” by all the other caches. Without intervention, the bus read will fetch the value stored in shared memory, however, cache i which is in state L, interrupts this bus read and performs its own bus write, updating main memory to contain the correct value. Cache i has the capability, as well as all of the other, to interrupt an activity on the bus, as explained in the assumptions made earlier in this section. The original bus read will be removed from the bus immediately. In addition to fetching the correct value, this bus read is noticed by all the caches, which then read the up-to-date value placed on the bus by PEi, and modifies its own cached copy of X. Since all of the caches that contain X now have the most up-to-date value, all of the caches change the state of the block to R.

Based on the above scenarios, one can realize that only cache blocks in the Local state need to be written back to main memory when they are evicted from the cache, since all other cache block states are consistent with main memory. The above scenarios, along with their associated cache state transitions, are illustrated in Figure 12.
Figure 12. RB (Read Broadcast) Protocol [Rudolph1984]

8.8 Read Write Broadcast Protocol

The Read Write Broadcast (RWB) protocol, also proposed by Rudolph et al [Rudolph1984], is an extension of the RB protocol, previously presented. Recall that in the RB protocol, the caches note the occurrence of both bus reads and bus writes, however, the data is only modified in the cache on a bus read. In the RWB protocol, all of the caches read the data on the bus on both bus reads and bus writes, hence the name.

Rudolph showed that the RWB protocol has improved performance over the RB protocol at the cost of introducing a new state and new bus action, both explained later. The RWB protocol differs from the RB
protocol in the way a cache makes the transition from a local configuration to a shared configuration. In the RB protocol, if variable X is in the local configuration, and if PEi was the last processor to modify its value, a read by any other processor, PEj, was the only operation that could change variable X to the shared configuration. In the RWB protocol, variable X will be changed to the shared configuration whenever another cache, PEj, references variable X, be it a read or a write.

In addition, in the RWB protocol, all variables are assumed to be in the local configuration until the first reference (read or write) by another processor, which changes the variable to the shared configuration. So long as the variable is in the local configuration, any read or write to it can be performed without notifying the other caches, thus reducing bus traffic.

A new state, first-write (F), and a new bus action, bus-invalidate (BI), are introduced to the design of the RB protocol in order to obtain the design of the RWB protocol. All of the state transitions that were used in the RB protocol are used again in the RWB protocol. The first write to a variable by PEi makes all caches remain in the R state except for the cache associated with PEi, which goes to the F state. Subsequent writes by PEi confirm the fact that the variable is local, and thus cache i enters state L and causes a bus-invalidate (BI) signal to be put on the bus. This signal causes all other caches containing the variable to enter the I state. This is equivalent to the local configuration explained in the RB protocol.

While the caches are all in state R, except for cache i in the F state, all reads have no effect on the state of the cache block, and valid data can be fetched from any cache. A write by some other processor, PEj, will cause cache j to go to the F state and a bus write to occur. The data written is read by all caches, and they in turn enter the R state. A read causing a cache miss generates a bus read. A bus write caused by a write-miss causes all other caches to enter the R state and the cache that initiated the write to enter the F state. These state transitions are explained in Figure 13.
Figure 13. RWB (Read and Write Broadcast) Protocol [Rudolph1984]

Legend:
- CW -> CPU write request
- CR -> CPU read request
- BW -> Bus write request
- BR -> Bus read request
- BI -> Bus invalidate request

1 -> generates a BW (write through)
2 -> interrupts bus read and supply the data from the cache
3 -> generated a BR (cache miss)
4 -> generates a BI
8.9 **Berkeley Ownership Protocol**

The Berkeley Ownership protocol, proposed by Katz et al [Katz1985], is an ownership-based multiprocessor cache consistency protocol designed to be implemented by a single chip VLSI cache controller. Its designers sought improvement over the other protocols by seeking solutions to the cache consistency problem that require a small amount of work on the system bus.

To begin with, an ownership protocol is one in which a processor must own a block of memory prior to being allowed to modify its contents. Ownership of a block is acquired through special read and write operations on the block required. By indicating the possibility of modifying a block at the time the block is read into a cache, the invalidation signals mentioned in the above protocols are avoided, thus reducing traffic on the bus. If the processor does not correctly pre-declare its intentions, extra bus traffic may be incurred [Katz1985].

The Berkeley protocol implements an ownership protocol with the owning caches inhibiting memory and owned blocks being kept in the cache. There are four states that cache blocks may be in under this protocol: Invalid, UnOwned, Owned Exclusively, and Owned NonExclusively. The Invalid state indicates that the cache block contains no useful data. The UnOwned state indicates that several caches may have copies of this block. The block contains valid data that is possibly shared among other caches. This block cannot be written to locally by any cache without the cache first acquiring ownership of the block. The Owned Exclusively state indicates that the owning cache holds the only cached copy of the block. Updates can occur locally without first informing the other caches. The Owned NonExclusively state indicates that other caches have a copy of the cache block and must be informed as to any changes made locally in a cache that contains the block.

By observing the above states, one realizes that at most, only one cache can own a block, and the owner is the only cache allowed to update the contents of the block. However, owning a block also means that the processor associated with the owning cache must provide the data contained in the block to other processors that request it. In addition, the owning cache must also update main memory when the contents of the cache block are updated, or when the cache block is evicted.

Along with the above states, the Berkeley Ownership protocol uses several bus operations in order to maintain the coherency of all of the cache blocks with those in the other caches, and with those in main memory. These bus operation are Read-Shared, Write, Read-For-Ownership, Write-For-Invalidation, and

111
Write-Without-Invalidation. The Read-Shared operation is a conventional read that gives the cache an UnOwned copy of the block. While in this state, the data contained in the cache block may be provided by the cache owner rather than by main memory. The Write operation is a conventional write that causes main memory to be updated and all cached copies to be invalidated. The Read-For-Ownership operation is similar to a normal read, except that the requesting cache becomes the exclusive owner after the read completes, and all other caches invalidate any matching entries. The Write-For-Invalidation operation is a quick version of the conventional write, but does not report the modification of the data value to main memory. The operation does cause the invalidation of other cached copies, but main memory will be updated later when the owned block is evicted from the cache, if it ever is. The Write-Without-Invalidation operation causes main memory to be updated with the new value, but any cached copies are kept valid. This operation is primarily used for evicting owned blocks from the cache, and reporting their modified contents to main memory, but is not an essential operation to execute in order to maintain the coherency of the protocol.

8.10 Comparison of the Berkeley Ownership Protocol to the Write Once Protocol
Both the Berkeley protocol and the write once protocol employ a copy back strategy in which dirty blocks are retained in the cache as long as possible. Main memory is only updated when the block is evicted from the cache. The write through part of the write once protocol places an upper bound on the total number of processor bus write transactions, since the write through strategy generates a bus write for every processor write [Katz1985]. Since the first write in the write once protocol uses write through, this write is produced even if the data is not shared, causing extra bus traffic, which increases in the write once protocol with increasing cache size. It has been proven that the write once protocol generates more bus traffic than the Berkeley Ownership protocol because of these extra writes [Katz1985].

In addition, for a block that is continually read and updated, the write once protocol requires that the block be read into the cache, modified locally in the cache, and then reported to main memory. This routine requires two bus actions: reading from memory and then writing back to it. The Berkeley protocol, on the other hand, requires that the block be acquired with a Read-For-Ownership bus operation. The cache block is then modified in the cache. If it is not accessed before it is evicted from the cache, the number of bus cycles required is the same as that for the write once protocol: one for the Read-For-Ownership operation and one for the Write-Without-Invalidation operation. However, it is likely that the block will be requested again, so from a single cache viewpoint, updating the cache block only requires the one bus action that reads it for initial ownership. Thus, in general, the Berkeley protocol transfers blocks directly between the
caches, whereas the write once protocol transfers blocks from the original cache, through to main memory, and then to the other caches that request it, thus invoking the entire latency of the memory system [Katz1985].

8.11 Firefly Protocol

The Firefly cache coherence protocol, proposed by Thacker et al [Thacker1988], is implemented in the Firefly Multiprocessor Workstation, developed at the Digital Equipment Corporation Systems Research Center, and is very similar to the protocol used with the Xerox Dragon multiprocessor network. The Firefly protocol allows multiple caches to contain a writeable cache block simultaneously, with no pre-arrangement required for a processor to write to a shared location.

The key idea in the Firefly protocol is to have the cache detect when another cache shares a particular memory location. With this concept, a write-back strategy is used for non-shared memory locations so that reads and writes of non-shared locations go directly to the cache, requiring no bus traffic and increasing the performance of the processor. Writes to main memory are then only needed when the block is evicted from the cache. For shared memory locations, a read of the cache block is serviced by the cache, but a write enforces a write through strategy, so that all other caches that contain the cache block are updated, as well as main memory.

To implement this protocol, two status bits are associated with each cache block: one to tell if the block is clean or dirty, and one to tell if the block is shared or exclusively owned. The two bits entail four states, which are shown in Figure 14, along with the transitions among the states.
**Figure 14. Firefly Coherence Protocol [Thacker1988]**

### 8.12 Advantages and Disadvantages of the Firefly Protocol

The Firefly protocol is advantageous over the other protocols presented above since the write through strategy is only used when it is logically necessary in order to support sharing. When a location ceases to be shared, only one extra write through operation needs to be done; by the last cache that contains the location. This suggests that the Firefly protocol has substantial bus traffic savings over the other protocols.
On the other side of the coin, the write through strategy will continue to be enforced so long as the cache block is shared, even though only one processor may actually be reading or writing to it. This actually increases the bus traffic, and proves this protocol to be futile in future snoopy cache coherence designs.
9 Future Work

Although extensive work and major goals have been accomplished in this thesis, there is still room for improvement. In addition, some additional tests can be performed using the VHDL implementations created in this thesis. In the following sections, some future plans are discussed, as well as different experiments that could be performed using the VHDL code that already exists.

9.1 Implementing Multiprocessor Cache Coherence Solutions in VHDL

The main purpose of this thesis was to implement different uniprocessor cache architectures in VHDL so that these implementations could be instantiated into a multiprocessor implementing one of the snoopy cache coherence protocols explained starting on page 100. As mentioned in the abstract, without a thorough understanding of the uniprocessor cache performance, one would not know how to begin to implement shared-memory multiprocessors. For this reason, more emphasis was placed on the correct implementation of uniprocessors than on multiprocessors. However, the VHDL implementation of a multiprocessor must be discussed.

To create a shared-memory multiprocessor in VHDL, one needs to create an entity called SMM, for example. In describing the architecture of SMM, there needs to be one component instantiation for each processor and local cache pair of the multiprocessor. For example, suppose one wants to create a shared-memory multiprocessor using four processors, each one containing a write-through, fully-associative cache using the LRU replacement algorithm. First, one would load the component WRITE_THROUGH_FULL_LRU_NO_INVALID into the architecture by using the component function of VHDL. Next, we would need to create four instances of this component in the behavioral description of SMM. To do this, we need to create devices CACHE_1, CACHE_2, CACHE_3, and CACHE_4. Each of these devices would stand for a different instance of the WRITE_THROUGH_FULL_LRU_NO_INVALID component. Within the declaration of the four caches, we need to create a port mapping that maps the signals used within the four caches to those used within the uniprocessor implementation of the write-through, fully-associative cache using the LRU replacement algorithm, as defined in the WRITE_THROUGH_FULL_LRU_NO_INVALID.vhd file. These signal names should be mnemonic but different among the four caches so that the designer can differentiate among the signals from the four caches.

In addition to the four caches, a memory unit would also have to be included in the SMM entity declaration. To do this, the designer would need to include the MEMORY component into the architecture
of SMM by using the component function. This MEMORY component must be instantiated later, as SHARED_MEM, for example, and signals must be created and mapped to each signal defined in the MEMORY.vhd file.

After all of the components have been instantiated, and the signals have been named and mapped, the snoop process must be written. This process would implement one of the snoopy cache coherence protocols discussed on page 100, or any other shared-memory cache coherence solution created by the designer.

Now that the snoopy mechanism has been established, the designer must write the actual programs that would be executed on the processors. This program could be a multitasked program divided up among the four processors, or each processor could run its own program, with all processors using data within its own local cache and accessing the same shared main memory module on a cache miss.

The final step must be to create the interface between the caches and the shared main memory unit. This would consist of signal assignments that would assign outputs from the caches to inputs of main memory, and vice-versa. These signal assignments would also include those required for the snoop process to correctly monitor the bus activity.

The VHDL implementation of the above component declaration and instantiation is outlined below.

```vhdl
entity SMM is
  port(
    ... define and declare input and output signals);
end entity;
architecture BEHAVE of SMM is
  component WRITE_THROUGH_FULL_LRU_NO_INVALID
    port(
      ... signals for WRITE_THROUGH_FULL_LRU_NO_INVALID);
  end component;
  component MEMORY
    port(
      ... signals for MEMORY);
  end component;
begin
  CACHE_1: WRITE_THROUGH_FULL_LRU_NO_INVALID
    port map(
      ... map signals used in CACHE_1 to those in
      WRITE_THROUGH_FULL_LRU_NO_INVALID);
  CACHE_2: WRITE_THROUGH_FULL_LRU_NO_INVALID
    port map(
      ... map signals used in CACHE_2 to those in
      WRITE_THROUGH_FULL_LRU_NO_INVALID);
  CACHE_3: WRITE_THROUGH_FULL_LRU_NO_INVALID
```

117
9.2 Using the Existing VHDL Code as a Teaching Aide

Another use of the existing VHDL code is to perform various simulations to obtain memory and cache access patterns of different programs. Since all of the main parameters such as the cache size, memory size, block size, and degree of associativity are all constants, their values need only be modified in the WRITE_BACK_CACHE_TYPE.vhd, WRITE_ONCE_CACHE_TYPE.vhd, WRITE_THROUGH_CACHE_TYPE.vhd, and/or the MEMORY_TYPE.vhd files. Then, all of the files in the thesis need to be re-compiled in order for the changes to take affect. Experimenting with the size and degree of associativity, as performed earlier in this thesis, would enable one to better understand the performance of uniprocessor caches, and the effect of different parameter changes on the performance of the cache.

9.3 Using the VHDL Code to Create Prototypes of New Cache Designs

One feature of software modeling is the ability to test new features and to see their impact on the design without spending exorbitant amounts of money on hardware prototypes. As mentioned earlier, the parameters of the cache size and block size can be changed in order to see the effects they have on the cache performance. These changes could also be performed in order to evaluate new cache designs. With the rising increase of the size of memory modules in today’s market, we predict that it will not be long before larger cache sizes are built. Using the software model created in this thesis, one can experiment with
larger cache sizes, and try to avoid the effects of larger cache sizes on the performance of the cache, as mentioned on page 75.

Once a satisfactory design has been achieved, the VHDL code could be run through a synthesizer such as Synopsys in order to obtain a gate-level schematic of the design. This design could then be sent to a fabricator who would then fabricate the chip for hardware implementation. This process was attempted early in the thesis, however, the behavioral synthesizer of Synopsys required significant amounts of VHDL code changes in order to accept our design. Since the focus of the thesis was on the modeling of cache architectures that already existed, there was no point in modifying the code in order to achieve a gate-level schematic of a piece of hardware that already exists.
REFERENCES


