FORTH-ICS/TR-382 July 2006

Design and Implementation of a Coherent Memory Sub-System for Shared Memory Multiprocessors

Evaggelos Vlachos

Abstract

Recent technology advances in integrated electronics offer the ability to add more and more transistors into modern chips. Chip Multiprocessors (CMPs) are architectures that feature multiple processing cores on a single chip. They result in higher processing power, easier design scalability, and greater performance/power ratio. CMPs appear to be one of the dominating architectural approaches for the years to come in the area of high performance architectures.

The purpose of this work is to design and implement a shared memory multi-core system that matches the needs of future CMPs. Specifically, an FPGA-based prototype has been implemented, which constitutes a two-node processing system. The design takes advantage of the two PowerPC cores that are embedded in the FPGA fabric. We have implemented external coherent caches equipped with a MESI protocol, and a bus-based coherent memory interconnect to connect the two processors. Shared memory resides in external DDR memory accessible through the interconnect and the DDR controller.

We find that the area overhead of our coherent memory system is 33.4% of a medium-size FPGA. We evaluate the performance of the system by using both simulations and custom software benchmarks running on the two processors. Our simulations show that the system implemented is more efficient than systems based exclusively by Xilinx soft-cores that offer the same type of memory coherence. Our custom benchmarks simulate basic operations found commonly in parallel programs. Our results show that our design scales well with respect to a single processor, for the Merge-Sort algorithm and the Producer-Consumer benchmark that don"t require a great amount of synchronization traffic. The speedup measured ranges between 1.89 to 1.92 and 1.89 to 3.45, respectively. On the other hand, the Shared-Counter benchmark slows down by 3 to 10 times due to excessive synchronization traffic.

Design and Implementation of a Coherent Memory Sub-System for Shared Memory Multiprocessors

Evaggelos Vlachos

Computer Architecture & VLSI Systems (CARV) Laboratory
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Science and Technology Park of Crete
P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece
Tel.: +30-81-391660 Fax: +30-81-391661
email: vlachos@ics.forth.gr

Technical Report FORTH-ICS/TR-382 – July 2006

Copyright 2006 by FORTH

Work performed as a M.Sc. Thesis at the Department of Computer Science, University of Crete, under the supervision of Prof. Manolis Katevenis

Abstract

Recent technology advances in integrated electronics offer the ability to add more and more transistors into modern chips. Chip Multiprocessors (CMPs) are architectures that feature multiple processing cores on a single chip. They result in higher processing power, easier design scalability, and greater performance/power ratio. CMPs appear to be one of the dominating architectural approaches for the years to come in the area of high performance architectures.

The purpose of this work is to design and implement a shared memory multi-core system that matches the needs of future CMPs. Specifically, an FPGA-based prototype has been implemented, which constitutes a two-node processing system. The design takes advantage of the two PowerPC cores that are embedded in the FPGA fabric. We have implemented external coherent caches equipped with a MESI protocol, and a bus-based coherent memory interconnect to connect the two processors. Shared memory resides in external DDR memory accessible through the interconnect and the DDR controller.

We find that the area overhead of our coherent memory system is 33.4% of a medium-size FPGA. We evaluate the performance of the system by using both simulations and custom software benchmarks running on the two processors. Our simulations show that the system implemented is more efficient than systems based exclusively by Xilinx soft-cores that offer the same type of memory coherence. Our custom benchmarks simulate

basic operations found commonly in parallel programs. Our results show that our design scales well with respect to a single processor, for the Merge-Sort algorithm and the Producer-Consumer benchmark that don”t require a great amount of synchronization traffic. The speedup measured ranges between 1.89 to 1.92 and 1.89 to 3.45, respectively. On the other hand, the Shared-Counter benchmark slows down by 3 to 10 times due to excessive synchronization traffic.

Acknowledgments

This work was carried out with the financial and technical support from FORTH - ICS and in the framework of the European FP6-IST program through the SIVSS (STREP 002075), UNIsIX (MC EXT 509595), SARC(FET 027648) projects, and the HiPEAC Network of Excellence (NoE 004408).

First of all I would like to thank my advisor Prof. Manolis Katevenis as also Prof. Angelos Bilas and Prof. Dionysios Pnevmatikatos, for their guidance and their support throughout this work. Their constructive remarks and the time they devoted to me constitute a significant amount of help.

Furthermore, I would like to thank all my fellow students and/or co-workers at FORTH for their help and their support in all the good and bad times. Working in the same environment with you my friends has been a pleasant and an honor.
(Dr. Manolis Marazakis, Vasilis Papaefstathiou, George Kalokairinos, Mixalis Ligerakis, Stamatis Kavadias, Mixalis Papamicheal, George Mihelogiannakis, Aggelos Ioannou, Nikos Andrikos, George Passas, Kostas Kapelonis, Evriklis Kounalakis, George Panagiotakis, Dimitris Antoniadis, Manolis Athanatos and many other....)

Finally, I would like to thank my family (Kostas, Marditsa, Apostolis and Anda) for their love and support they have offered all these years. They have sacrificed everything in order to help me reach my goals. Without their help I would certainly have not made it to here.

To my family

Contents

1 Introduction **1**

1.1 Background . . . 2

1.1.1 Shared Memory Multirocessors . . . 2

1.1.2 Chip Multiprocessors . . . 8

1.2 Releated Work . . . 9

2 Design and Implementation **10**

2.1 General Description . . . 10

2.2 PowerPC 405 . . . 12

2.3 PLB Bus Interconnect . . . 13

2.4 PLB BRAM Controller, BRAM blocks and PLB2IPIF Controller . . . 14

2.5 PLB2Cache module . . . 15

2.5.1 PLB2Cache Module Architecture . . . 16

2.6 Coherent Cache . . . 17

2.6.1 Cache Characteristics . . . 18

2.6.2 Tags and Data Memory Organization . . . 19

2.6.3 Coherency Protocol . . . 19

2.6.4 Part A: Cache's Processor side . . . 21

2.6.5 Part B: Cache's Bus side . . . 25

2.7 Coherent Memory Interconnect . . . 28

2.8 DDR Multiplexer . . . 31

2.9 DDR Controller . . . 32

3 Evaluation and Verification **33**

3.1 Hardware Resources . . . 33

3.1.1 Target FPGA . . . 33

3.1.2 Hardware Resources . . . 33

3.1.3 Timing Conciderations . . . 35

3.2 Performance Evaluation . . . 36

3.2.1 Comparison with other Coherent Shared Memory Organizations . . . 41

3.3 Correctness verification . . . 43

3.3.1 Software primitives . . . 43

3.3.2 Shared Memory Programs . . . 44

4 Conclusion and Future Work **53**

References **54**

A DSPLB & PLB2Cache module **57**
A.1 DSPLB Behavior 57
A.2 DSPLB Signal Summary 57
A.3 PLB2Cache module FSMs 61
A.4 Returning Data 62

B Details about the Coherent Cache **64**
B.1 Part A FSMs 64
B.1.1 FSM_CPU_ACCESS 64
B.1.2 WB_FSM 65
B.2 Completing Requests and Cache Status 67
B.3 Part B FSMs 68
B.3.1 BUS_FSM 68
B.3.2 FSM_REQ_IN 70
B.4 Communication with the dependency check module 71

C Coherent Bus Module FSMs **72**
C.1 FSM_Arb 72
C.2 FSM_BUS_WB 74

D Detailed Reports **75**
D.1 Waveforms for Halted IPIF Burst Accesses 75
D.2 Detailed Timing Reports for Critical Paths 76

List of Figures

1.1 The Cache Coherence Problem 3
1.2 State diagram for the MSI write-invalidate cache coherece protocol 6
1.3 State diagram for the write-update cache coherece protocol 7

2.1 Architecture of the System 11
2.2 PowerPC 405 Organization 13
2.3 3-cycle PLB Arbitration 14
2.4 Connection between the PowerPC, the PLB2Cache module, and the Bus 15
2.5 PLB2Cache Architecture 16
2.6 Coherent Cache General Description 18
2.7 Coherency Protocol State Diagram 20
2.8 Part A block diagram 22
2.9 Two First Cycles of a Processor Access 23
2.10 Cache2Bus Block Diagram 26
2.11 Bus2Cache Block Diagram 27
2.12 Coherent Memory Interconnect Block Diagram 29
2.13 IPIF Accesses 31

3.1 Floorplanned view of the system 34
3.2 Write hit and Read hit Scenarios 37
3.3 Invalidate Scenario 38
3.4 BusRd Remote Cache hit Scenario 39
3.5 BusRd Remote miss. Fetching cache block from external memory. Block eviction at the same time 40
3.6 Non-Cacheable Write & Read Accesses 41
3.7 Locking Algorithms 44
3.8 Competing for Access to Shared Memory 47
3.9 Producer - Consumer Program 49

A.1 Data-Side PLB Interface Block Symbol 59
A.2 PLB Accesses 60
A.3 PLB2Cache module FSMs 61
A.4 DSPLB Three Consecutive Word Reads 63

B.1 FSM_CPU_ACCESS State Diagram 65
B.2 WB_FSM State Diagram 66

B.3 BUS_FSM State Diagram 69
B.4 FSM_REQ_IN State Diagram 70

C.1 FSM_Arb State Diagram 73

D.1 IPIF Burst Accesses Halted 75

List of Tables

2.1 Generation of Coherent Requests 20
2.2 Combinations of Accesses 24

3.1 Virtex II Pro Resource Summary 33
3.2 Hardware Utilization 35
3.3 Delay of Critical Paths 36
3.4 Access Latency (measured in PLB cycles - 100MHz clock freq.) 42
3.5 Additional penalties imposed to Shared Memory Requests (measured in PLB cycles - 100MHz clock freq.) 42
3.6 System Parameters 45
3.7 Duration of "Shared-Counter" program for different arcitectures in processor cycles (clk. freq. 100 MHz) 45
3.8 Duration of "Producer-Consumer" in processor cycles (clk. freq. 100 MHz) . . . 50
3.9 Duration of "Merge-Sort" program in processor cycles (clk. freq. 100 MHz) . . . 52

A.1 DSPLB PLB Interface Signal Summary 58

D.1 Time Consumption at the Read Hit Path 76
D.2 Time Consumption at the end of the Read Miss Path 77
D.3 Time Consumption when requesting the Bus 77

Chapter 1

Introduction

Recent technology advances in integrated electronics offer to computer engineers the ability of adding more and more transistors into modern chips. The logic being added is becoming more and more complex, describing and implementing more powerful designs. This effect has resulted in the occurrence of Chip Multiprocessors (CMPs). This approach suggests that high-performance processor architectures should move towards designs that feature multiple processing cores on a single chip. These designs have the potential to provide higher peak throughput, easier design scalability, and greater performance/power ration than nowadays uniprocessor ones. This trend appears to be one of the dominating architectural approaches for the years to come in the area of high performance architecture. Specifically, there are already some multi-core architectures on the market [1, 2, 3] that dispose a small number of processing cores. In the near future CMPs are expected to have a larger number of processors, since global wire delays will limit the area of the chip that is useful for a single conventional processing core. It is following that this area will be dedicated to deploy additional cores [4, 5, 6].

Having so many units on a single chip, it certainly alters the architectural decisions that have been considered safe until now. Processing power no longer constitutes the bottleneck of these designs. The vast amount of transistors available on chip has transferred the bottleneck to the need of making all the processing cores cooperate efficiently. Thus, one of the most important characteristics, on which a great amount of consideration will be focused, will deal with communication issues between the multiple processors, either on-chip or even off-chip. The on-chip communication is usually carried out by on-chip interconnection networks that connect the on-chip processing elements. The off-chip communication is usually the responsibility of a Network Interface, which make feasible the communication with other multinode systems. However, efficiency in communication has a close relation with the proximity of computation to communication. This tight coupling between computation and communication is usually expressed as customizing the features of Network Interfaces in order to meet particular application domain demands. This, however, influence the design of the interconnect network. Thus, an integrated design of the network interface and the interconnection network will provide the desired features to the whole system.

An approach that offers tight coupling between communication and computation suggests the use of Coherent Network Interfaces (CNI) [7, 8]. According to this approach, the multiple nodes (CPUs) use a coherency protocol to share the available memory. The Network Interface is connected to the memory bus and uses the underlying coherency protocol to transfer data to

the memory and/or the cache hierarchies. In this way, low-latency communication is provided, as opposed to the currently long-latency coupling through the I/O bus.

The purpose of this study is to design and implement a system that comes closer to the future architecture that described above. Specifically, an FPGA-based prototype has been implemented, which constitutes a two-node processing system. A Xilinx Virtex-II Pro FPGA has been used to host the whole system. The design takes advantage of the two PowerPC cores that are embedded in the FPGA fabric [9, 10, 11]. External coherent caches and a coherent memory interconnect have been implemented to connect properly the two processors. Shared memory resides in external DDR memory accessible through the interconnect and the Xilinx DDR controller. Each processor is also connected to a PLB bus in order to have access to instructions and private data. Finally, the coherent memory interconnect has been designed to accept also a third participant, which can be a coherent network interface. However, this entity remains a future objective.

The rest of the thesis is organized as follows: Background information and related work are discussed in the rest of this chapter. In Chapter 2 the design and the implementation of the whole system is presented. Experimental results and comments from the evaluation of the system are shown in Chapter 3. Finally, the conclusion of this study and future work directives can be found in Chapter 4.

1.1 Background

This section of the chapter presents some background information about Shared Memory Multiprocessors and Chip Multiprocessors. Specifically, the Cache Coherence property will be described in detail, and some Cache Coherence protocols will be presented. Finally, the CMP architectural organization will be presented and analyzed.

1.1.1 Shared Memory Multirocessors

The Cache Coherence Problem

Figure 1.1 depicts the problem that arises when multiple processors have access to a shared region of memory. Processors P1 and P2 are connected through an interconnection network to the main memory, while both of them have one private write-back cache. In this example P1 and P2 share variable Y, which both use it in the normal flow of their parallel program. At some point in time processor P1 reads (arc no. 1) variable Y from main memory and places a copy of Y into its cache. Let's say that at that time Y has the value of '10'. Then, processor P2 reads (arc no. 2) variable Y from main memory, placing a copy of Y into its cache, too. The value of Y is still '10' and at that point both processors have an up-to-date copy of Y. The problem arises when processor P1 attempts to modify (arc no. 3) the value of Y. The action of writing to Y the value of '5' updates only the local copy that P1 retains in its cache. However, the copy of Y that P2 has remains intact maintaining an older value of Y. When processor P2 attempts to read Y (arc no. 4), it reads a stale value of that variable. At this point processor P2 has an out-of-date view of the specific memory location, resulting to wrong execution of the parallel program. This is usually referred to as the cache coherence problem.

A naive solution to this problem would be to prevent caching of shared memory by the processors. However, this would have a tremendous negative impact on the performance of the parallel program. Furthermore, in shared memory multiprocessor architectures, reading from and writing to shared memory regions by different processors is expected to happen frequently.

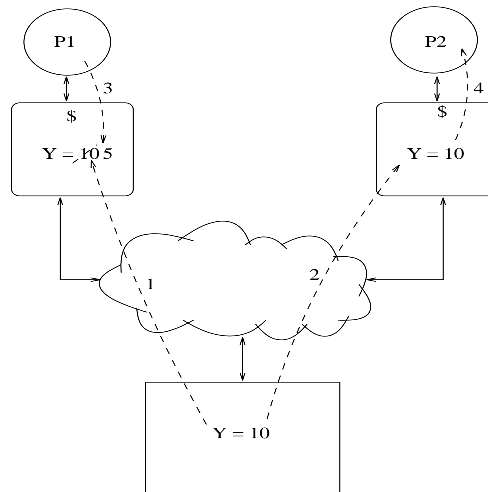


Figure 1.1: The Cache Coherence Problem

This frequent event is used by processes of a parallel program to communicate with each other. This concludes to the fact that addressing shared memory in these architectures must be addressed in different way than convenient uniprocessors do. The answer to this problem is given by hardware techniques that provide coherency among shared data.

Cache Coherence

In the previous example the problem appeared when the last read operation issued by processor P2 didn't return the up-to-date value of variable Y. This happened because the last write access to this variable was made by processor P1, and P2 was never informed about that. This problem is attributed to the memory interconnect, part of which are also the processors' caches. In [12] a strict definition of cache coherence is given and is apposed below for completeness. According to it,

> *A memory system is coherent if:*
>
> 1. *A read by a processor P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always return the value written by P.*
> 2. *A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated and no other writes to X occur between the two accesses.*
> 3. *Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.*

The first property indicates that operations issued by any processor occur in the order which they are issued to the memory system by that processor. That means that the memory system

doesn't change the relevant ordering between memory operations from the same processor. The order, which is preserved in this way, is the same order that the memory operations appear in the program, since the processor does not issue memory operations in a way that will violate the program semantics. The second property indicates that the value returned by each read operation is the value written by the last write operation to that location. If that couldn't hold then the whole system wouldn't be able to become in any way coherent. The violation of this rule was presented in the above example. Finally, the third property indicates the serialization of memory operations from all the processors. Every memory operation accesses a physical location at main memory. Since the physical memory module can serve one request at a time, it would impose a serial order on all the read and write operations from all the processors to any location.

In order to enforce coherency among multiple processors, hardware protocols embedded in the cache hierarchy of each processor manage all the generated accesses [13, 14]. They use the coherent memory interconnect to exchange messages with the other caches to impose synchronization among them. The objectives that they serve are: upon a write request the local cache must ensure that it holds the only copy of the data accessed. Also, all the other caches of the system must be informed about this action and must either invalidate or update their specific copy of the data, if they have one. Upon a read request the local cache must notify all the other caches of the system that a new copy of the data accessed has been generated. In this way a cache that had until now a unique copy is informed that this situation has changed.

Hardware Schemes for Enforcing Coherence

An important property of coherency protocols is the way they track the state of each data block accessed by a processor. There are two dominant approaches that have been proposed (found in [15]). The first one, named *Directory based*, suggests that the state of every block is kept in a single location. This location is responsible for administering the specific block of memory, by making it available to other processors in a manner that preserves data coherency. The second approach, named *Snooping based* suggests that no centralized state is kept. On the contrary, every cache that maintains a copy of a block also maintains a copy of the sharing status of the same block. Every time a memory block is requested all the caches must search themselves to see if they hold a copy of this block. If a cache does have a copy, then it follows the steps imposed by the coherency protocol.

Shared memory systems that follow the first approach separates the whole memory to n parts, where n the number of available processors. Each part is assigned to one processor, which is called the host-processor and is responsible to administer it properly. The host-processor maintains the status of every block memory assigned to it, and also a list of all the other processors that have access to memory blocks, for every memory block to its jurisdiction. This hardware structure is called *directory* and is maintained within the cache hierarchy of each processor. Every time a processor wants to access a block that doesn't resides in its local cache, it sends a request to the *directory* of the owner processor. Depending on the type of the action requested, the *directory* is responsible to notify all the other caches that already hold a copy, and also provide the requested data, if needed. The interconnect connecting the processors of the systems is not required to have any specific properties, and affects only the form of the messages exchanged. The *directory based* schemes have appeared later in the bibliography and as has been proved they have a very good performance. The key characteristic of this architecture is that it

scales well when the number of processors available in a system increases. However, its major drawback is its complexity. Specifically, each directory structures constitutes a hot-spot of the system, since they receive requests by all the processors, including the local one, that require access to the range of memory assigned to it. Due to its complexity the implementation of such a scheme was rejected, and thus there will be no longer reference to it in the rest of the thesis.

On the other hand, shared memory systems that follow the second approach no parts of memory are assigned to any processor. At any given time, a processor that requests to access a memory block, which doesn't resides in its local cache, it sends a message to all the other caches. All the caches *snoop* the traffic on the interconnection network to identify a new message. If no cache has a copy of the requested block then the block is loaded from main memory. If, however, one or more caches maintain a valid copy, one of them sends the requested block back to the cache that requested it. Messages are used not only to facilitate data transferring. Every message is assigned a type, which has a specific meaning for the coherency protocol. Based on this type caches that receive such messages are becoming familiar about the intention of the requesting processor. Having this knowledge they are able to follow the steps imposed by the coherency protocol. This kind of coherency protocols adds an additional requirement from the interconnection network, which constitutes the basic property of the protocol. This requirement refer to the ability that must be offered to any cache to broadcast messages and also to *snoop* the bus activity. Otherwise, it is impossible for the distributed protocol to synchronize the processors' requests.

Snooping Based Coherency Protocols

There are two types of snooping based coherency protocols that have been proposed until now. Each one of them has been presented in many versions; however the basic steps remain identical. This separation of protocols is based on the action taken by the protocol in order to preserve coherency. Coherency is usually threatened by write actions, as was also presented in Figure 1.1. There are two ways to maintain the coherence requirement. The first one is to ensure that a processor has exclusive access to a data item before it writes that item. Protocols that follow this approach are usually called *write invalidate* protocol because of the action taken. The second way is to update all the other caches that hold a copy of the item addressed, with the new value that is about to be written. This type of protocol is called *write update* protocol. These two types of coherency protocols are described below.

Write Invalidate Protocol A basic invalidate protocol is the three-state MSI write-back invalidation protocol. The protocol uses three states to encode the state of a cache block that resides in a processor's cache. These three states are the *Invalid*, the *Shared* and the *Modified*. The *Invalid* state corresponds to the absence of the requested block from the cache. The *Shared* state means the block is present in an unmodified state in the cache, the main memory is up-to-date, and zero or more copies of the block can be found in other caches. Finally, the *Modified* state means that only this cache has a valid copy of the block, and the copy in main memory is stall. Figure 1.2 depicts the state diagram of the protocol. Actions inducing transitions between states are shown next to the arcs. Processors issue two types of accesses, reads (*PrRd*) and writes (*PrWr*). Next to these accesses the corresponding bus messages, which will be generated upon a cache miss, are shown. A *BusRd* message is generated when a *PrRd* misses in the cache. The cache sends a *BusRd* message to request a copy of the specific block that doesn't intent to modify. A memory

system participant, either some other cache or the main memory, will reply. A *BusRdX* message is generated when the processor wants to write a cache block that is either not present in the cache or is in the cache but not in the *Modified* state. The message is sent to all the other caches, which invalidate their copies, if they have one. A cache or the main memory supplies an exclusive copy of the block. The write action completes when the copy arrives in the cache.

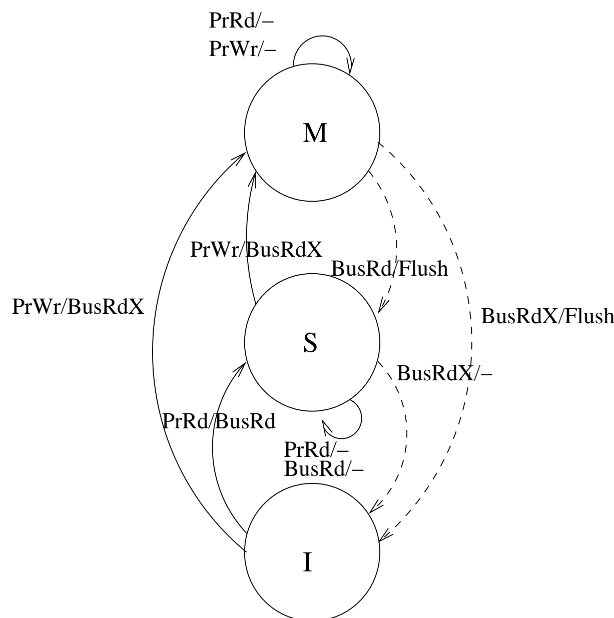


Figure 1.2: State diagram for the MSI write-invalidate cache coherece protocol

Observing the state of blocks from the processor's side, a *PrRd* always causes the requested block to transit to *Shared* state (transitions I to S and S to S). Furthermore, a *PrWr* always causes the requested block to transit to *Modified* state (transitions I to M , S to M, and M to M). Observing the state of the block from the bus's side then the following transitions can take place. A *BusRdX* message always causes the requested block to transit to *Invalid* state (transitions M to I and S to I). If the block is initially found in the *Modified* state, then the cache may have to reply with the cache block (flush action). Furthermore, a *BusRd* message always causes the requested block to transit to the *Shared* state (transitions M to S and S to S). Again, if the block was initially found in the *Modified* state, the cache may have to reply with the cache block (flush action).

Write Update Protocol A basic update protocol is the three-state MSmSc write-back update protocol. The protocol uses three states to encode the state of a cache block that resides in a processor's cache. These three states are the *SharedClean*, the *SharedModified* and the *Modified*. The *SharedClean* state means the block is present in an unmodified state in the cache, the main memory may or may not be up-to-date, and zero or more copies of the block can be found in other caches. The *SharedModified* state means that one or more caches have a copy of this block, main memory isn't up-to-date, and it's this cache responsibility to update the main memory. Only one cache can be in *SharedModified* state for a specific block at each time. Finally, the *Modified* state means that only this cache has a valid copy of the block, and the copy in main

memory is stall. Figure 1.3 depicts the state diagram of the protocol. Actions inducing transitions between states are shown next to the arcs. Processors issue two types of accesses, reads (*PrRd* or *PrRdMiss*) and writes (*PrWr* or *PrWr*).Next to these accesses the corresponding bus messages, which will be generated, are shown. A *BusRd* message is generated when a *PrRd* misses in the cache. The cache sends a *BusRd* message to request a copy of the specific block that doesn't intent to modify. A memory system participant, either some other cache or the main memory, will reply. A *BusUpdate* message is generated when the processor writes to a cache block. The bytes written by the processor are broadcasted to all the other processors so that they can update their copies, if they have one.

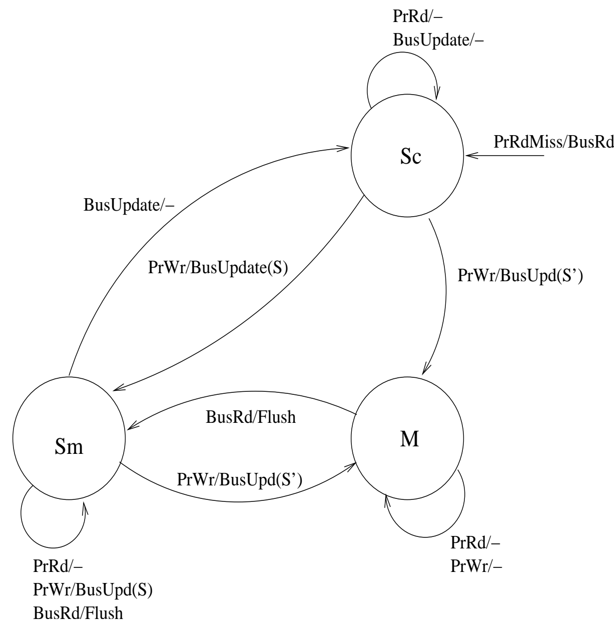


Figure 1.3: State diagram for the write-update cache coherece protocol

Observing the state of blocks from the processor's side, a *PrRd* always causes the requested block to enters the cache in the *SharedClean* state, or maintain itself in this state. On the other hand, a *PrWr* message can cause the block to transit either to *SharedModified* or to *Modified*. The final transition depends on the state of the block in the other caches. If the block is shared then the Sc to Sm transition takes place, otherwise the block transits to *Modified* state. If the block is initially absent from the cache a *PrWrMiss* is translated to a *PrRdMiss* followed by a *PrWr* scenario. Observing the state of the block from the bus's side then the following actions are taken. A *BusRd* message causes a modified block to enter one of the *Sc* or *Sm* states. It depends on the initial state of the block which one will be the resulting state. If the block was found in the *M* state then it transits to *Sm* state. Otherwise, if the block was found in either two states then it remains in that states. Whatever the transition might be the cache may be forced to transmit a copy of the block. On the other hand, a *BusUpdate* message can find a block only in the *Sc* or *Sm* state. In this case the block transits to *Sc* state, updating its part that is modified. Both of these two types of coherency protocols that resented above have been used the past years. Patterns of memory accesses may be presented that are served more efficiently by an invalidate or an update protocol. However, throughout the years the scientific community has shown a preference for

invalidate protocols over the update ones. The main reason for this comes from the fact that the update protocols generate great amounts of traffic on the memory interconnect. For this study, an invalidate coherency protocol was chosen to be implemented. The protocol, which is presented in the next chapter, constitutes an extension of MSI protocol presented above.

1.1.2 Chip Multiprocessors

While CMOS manufacturing technology continues to improve, reducing the size of single gates, physical limits of semiconductor-based microelectronics become a major design concern. Some effects of these physical limitations can cause significant heat dissipation and data synchronization problems. The demand for more complex and capable microprocessors causes CPU designers to utilize various methods of increasing performance. Some Instruction Level Parallelism (ILP) methods like superscalar pipelining are suitable for many applications, but are inefficient for others that tend to contain difficult-to-predict code. Many applications are better suited to Thread Level Parallelism (TLP) methods, which suggest the parallel execution of multiple threads in one or more processors. Multiple independent CPUs is one common method used to increase a system's overall TLP. A combination of increased available space due to refined manufacturing processes and the demand for increased TLP led to the logical creation of Chip MultiProcessors (CMPs). CMP organizations combine two or more independent processors, often called cores, into a single integrated circuit. The cores are usually connected together using a Network-on-Chip (NoC) type interconnection network.

In general, the existence of multiple processors on a single chip provides excessive computational power. This can be translated by parallel applications as opportunity to exhibit thread-level parallelism (TLP) to a higher extend. Furthermore, communication between different CPUs is carried out faster as opposed to parallel computers. In a CMP system the end points of communication are found in the same chip. The messages exchanged between them don”t experience the latency of traveling off-chip. As far as the hardware perspective is concerned, a CMP design disposes greater performance/power ratio than monolithic designs. An n-node CMP system consumes less power than n single processors. This comes from the fact that a CMP system has fewer pins, which also means that less power is consumed to drive signals external to the chip. Furthermore, the smaller silicon process geometry allows the cores to operate at lower voltages; while a part of significant size of the circuitry (part of the cache hierarchy) is usually shared among the processors. Finally, CMP designs are based on duplicates of the same core, which eases design scalability, and produces a product with lower risk of design error.

The major disadvantage of the CMP designs is the great power dissipation that results in increasing chip temperatures. A CMP chip may consume less power than n equivalent uniprocessor chips, however the total amount of power consumption remains prohibitive. Not all the cores are able to function at the same time for a long period of time, due to the fact that the circuitry will melt down. A solution to this problem suggests to dynamically switching on and off some of the available cores, in order to lower the consumption level. From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores.

1.2 Releated Work

Many projects have undertaken the task of designing and implementing multiprocessor designs, in order to evaluate new architectures and ideas. Some of them [16, 17] come from the early 90s, when the *proof by construction* approach was very popular. In recent years, however, high financial cost and human effort of fabricating chips in modern process technologies have made building more daunting, and as a result fewer projects [18, 19, 20] have endeavored to build a full implementation.

The J-Machine [16] and the MIT Alewife multiprocessor [17] come from the parallel computers family, the predecessor architecture of the CMP organization. They are both organized as multi-node systems, using a mesh network to connect all the components. The J-Machine uses messages to communicate data between the processors, while the Alewife multiprocessor follows the shared memory approach, using directory based coherence. However, there is also a hardware-software co-operation to support some sort of messaging among the processors.

The architecture of the Hydra Chip Multiprocessor [18] comes closer to the one studied here. The Hydra chip features four MIPS-based processors and their primary caches on a single chip together with a shared secondary cache. Each processor incorporates separate instruction and data caches, connected on a bus-based interconnect. An invalidation snooping coherency protocol is used to maintain coherency among shared data. To simplify parallel programming, the Hydra CMP supports thread-level speculation and memory renaming.

The Piranha architecture [19] follows the same approach with the Hydra CMP, however there are a number of differences. The Piranha has eight cores, each one of them having private first level cache. The second level cache, which does not maintain inclusion, is shared among processors, while the cache controllers use a more complex protocol to maintain coherence. A high-speed switch is used to connect the on-chip cores, instead of the bus that Hydra uses. Finally, the Piranha architecture is designed to provide scalability past a single chip by integrating the required on-chip functionality to support glueless multiprocessing.

The Raw CMP [20] is the first chip to organize its processors in a mesh. Specifically, it comprises of sixteen *tiles*; each one incorporating a compute processor, routers, network wires, and instruction and data memories. Raw distinguishes itself from others by being a modeless architecture and supporting all forms of parallelism, including ILP, DLP, TLP and streams. It uses messages to communicate data among the processors, while the on-chip interconnects belong to the class of scalar operand networks.

Chapter 2

Design and Implementation

In this chapter a detailed description about the design and the implementation of the whole system is given. Steps followed and choices made, concerning the different kinds of alternative solutions that could be followed, are presented and discussed.

2.1 General Description

The architectural organization of the system can be seen in Figure 2.1. The whole system is organized based on the two IBM-PowerPC processors [9]. Each one of them is connected to a private bus, the architecture of which follows the "Processor Local Bus" specification given by the IBM [21]. The connection is established through the two PLB-master interfaces, Instruction-Side PLB (ISPLB) and Data-Side PLB (DSPLB). ISPLB is responsible for fetching instructions into the PowerPC's Instruction Cache Unit (ICU) and the DSPLB for fetching the required data into the Data Cache Unit (DCU). Physical memory is provided to the processors by internal BRAM blocks and external DDR memory. Instructions and private data can be stored in either the external or internal memory, while shared data can be stored only in the external DDR memory. BRAM blocks are directly connected to the corresponding PLB bus using an appropriate PLB-slave controller. The external DDR memory can be addressed in two different ways, regarding the kind of memory being accessed. If a processor accesses shared memory the only way to reach it is through the coherent memory system, otherwise private data and instructions located in the external memory are being accessed through the PLB2IPIF bridge, bypassing the coherent memory hierarchy. Requests for the external DDR memory are being multiplexed before reaching the DDR controller. DDR_MUX is responsible for this operation, selecting and forwarding one request at a time to the DDR controller.

PowerPC's accesses, upon their initiation, are being separated to shared and private. The disjunction is performed based on the address provided by the processor. This operation is performed by the PLB2Cache module, which filters memory accesses considered to be shared and redirecting them towards the coherent memory system. In order this to become feasible the PLB2Cache module is inserted between the DCU pins of the PowerPC and the PLB bus.

The next step on the coherent path are the coherent caches. Both of them are equipped with a

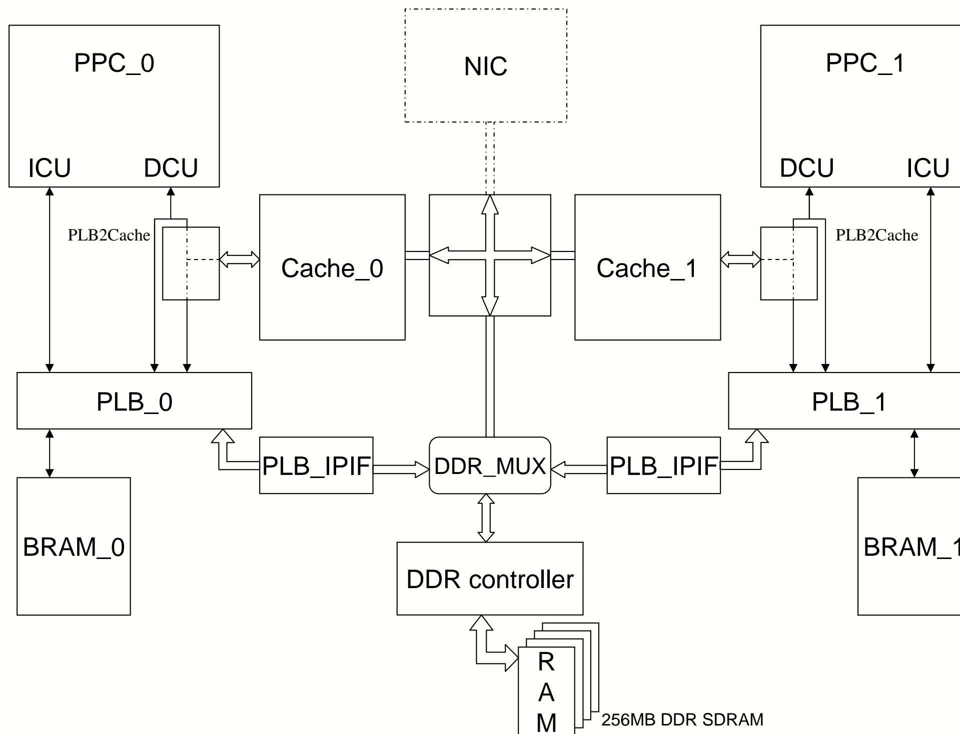


Figure 2.1: Architecture of the System

MESI-like cache coherent protocol in order to enforce cache coherency. The size of each cache is fully parameterized in order to try different organizations, while their associativity and the size of the cache lines are fixed and configured to 2-way and 8 words per cache line, respectively (as the internal caches of the PowerPC are). Regarding to the write policy, both caches handle "dirty" blocks following the write-back approach; every cache block that has been modified is written back to main memory when evicted by the cache and not when it is modified. Finally, evictions are handled in an LRU fashion.

When an access is forwarded to the cache there are two possibilities, either it can be handled by the cache or the request should be forwarded to the next level of the memory hierarchy. There are three reasons why an access cannot be handled by the cache. The first one is that the address requested lies in a non-cacheable shared address space and thus the request should be forwarded to the DDR controller. The second reason is that the requested block is not present in the cache and thus it should be requested by the next level of the memory hierarchy, and finally, the third reason is that the specific cache may have the requested data, but it doesn't hold the appropriate privileges to access them. In any of the above cases the coherency protocol must decide the set of actions required to satisfy the processor"s request while preserving coherency. Simultaneously with processor's requests, the coherent cache is designed to receive requests issued by the adjacent processor, too. The coherency protocol is again responsible to detect any conflict that will result in loss of coherence and enforce a sequence of actions that will preserve

coherency. Local and remote accesses to the cache can be handled in parallel, provided that there is no conflict, since tag information is stored in BRAM memory, which disposes two write ports and two read ports.

Finally, the two coherent caches and the DDR controller are connected to a memory interconnect, which is used to transfer requests and shared data to and from the participants. The interconnect has been designed to have the properties of a bus. Every memory access that doesn't hit in the local cache is being broadcasted. The remote cache snoops the bus activity in order to respond to requests issued by the local cache. If the requested data are found in the remote cache, which is called a remote-hit, they are sent back to the local cache. Otherwise, a negative respond is issued and the bus then forwards the request to the DDR controller. When the data become available are eventually sent to the local cache. During this whole procedure, the bus has been acquired by one cache only and no other cache is able to send another request. This means that the bus doesn't support interleaved accesses and it starts to serve the next request only after having finished serving the previous one.

The rest of the chapter presents all the entities involved one by one. The behavior and the details of all the functional blocks designed and implemented are presented. Furthermore, basic blocks, which are part of the Xilinx EDK library, are also presented and their behavior is analyzed.

2.2 PowerPC 405

The PowerPC 405 is a 32-bit implementation of the PowerPC embedded-environment architecture that is derived from the PowerPC architecture [9, 10, 11]. Xilinx Virtex-II Pro FPGA family is equipped with two embedded PowerPC 405 processors, as hard blocks within the circuitry of the FPGA.

Figure 2.2 shows the internal structure and organization of the PowerPC 405. The central-processing unit (CPU) implements a 5-stage instruction pipeline consisting of fetch, decode, execute, write-back, and load write-back stages. The fetch and decode logic sends a steady flow of instructions to the execute unit, which are executed in-order.

Memory accesses initiated by the pipeline pass through the Memory Management Unit (MMU) before they reach the caches. Two modes of address translation, real and virtual, are supported. When operating in real mode the addresses generated by the program (logical address space) running on the processor is used directly by the hardware to access the data. In virtual mode, there is an intermediate step, where the logical address is translated (mapped) to a physical address, according to the translation found in the Translation Look-aside Buffer (TLB) table.

An instruction-cache unit and a data-cache unit are found next to the MMU. Each cache unit contains a 16 KB, 2-way set-associative cache array, plus control logic for managing cache accesses. The caches contain copies of the most frequently used instructions and data and can typically be accessed much faster than system memory. Each cache line stores 32 bytes of continuous and aligned memory, a tag used to identify the line within the set, a dirty bit that indicates whether the cacheline has been modified since the time that was loaded into the cache and an LRU bit, which specifies if the specific line was (or not) the one least-recently used in the set. This information is used by the cache-line replacement algorithm when it is necessary

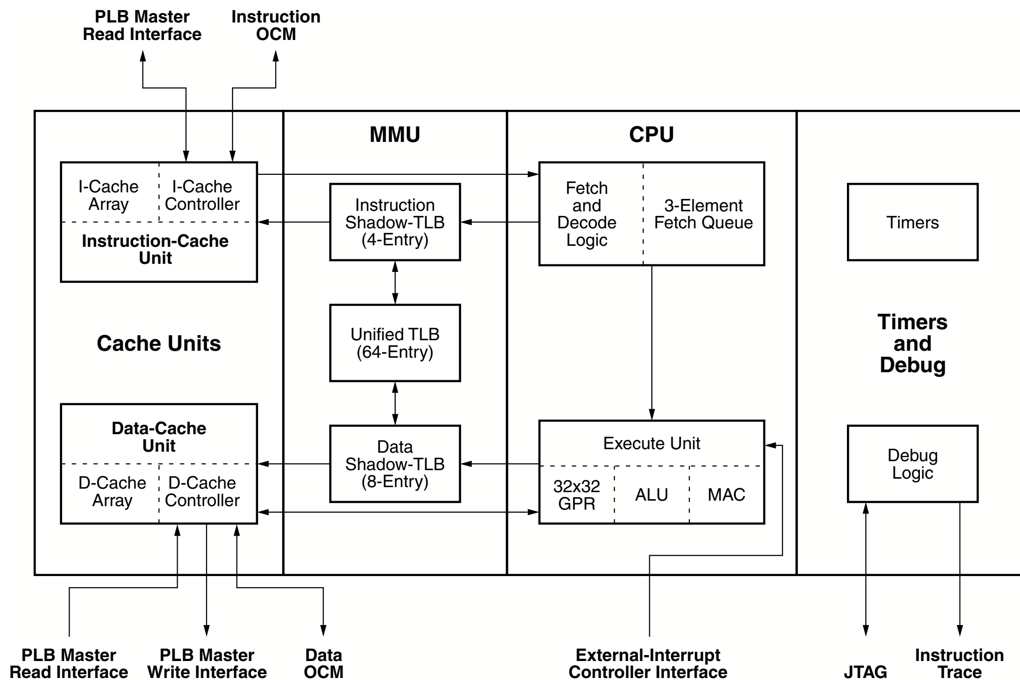


Figure 2.2: PowerPC 405 Organization

to evict a cache line in order to fetch a new one. In that case, the least recently used cacheline is evicted.

2.3 PLB Bus Interconnect

The Xilinx PLB [21] is an IBM CoreConnect compliant interconnect which allows multiple masters and multiple slaves to be connected to the bus. It consists of a central bus arbiter, the necessary bus control and gating logic, and all the necessary bus OR/MUX structures. The entire Xilinx PLB bus structure is provided as soft-core and allows for direct connection for up to 16 masters and 16 slaves. It supports both 64-bit and 32-bit peripherals to be connected at the same time. Each access requires 3-cycles for bus arbitration before the corresponding peripheral becomes aware of the request. At the end of the third cycle, the request can be safely latched by the peripheral. Figure 2.3 depicts a 3-cycle arbitration scenario.

Except for the multiple masters, the PLB implementation also supports pipelining of the requests, permitting the existence of more than one pending requests. This capability maximizes PLB-transfer throughput by reducing dead cycles between multiple requests. Finally, in order to differentiate the importance of each of the multiple masters, four levels of dynamic master request priority are available.

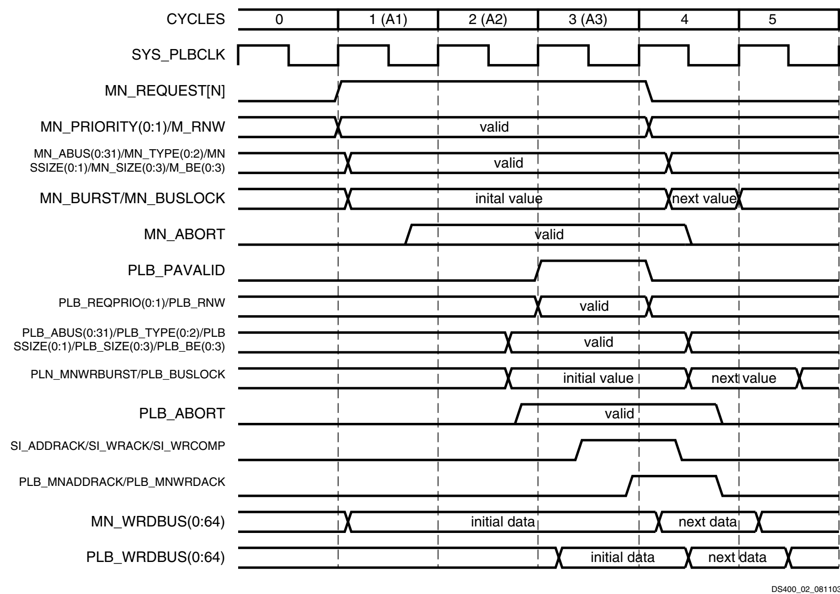


Figure 2.3: 3-cycle PLB Arbitration

2.4 PLB BRAM Controller, BRAM blocks and PLB2IPIF Controller

The PLB BRAM controller provides the opportunity of connecting some of the available BRAM memory to the PLB bus, and thus provides to the processors an easy way to access the embedded memory. It is attached to the PLB bus as a slave peripheral and translates the PLB protocol to a simpler form, which is recognizable by the BRAM block module.

The BRAM block is a reconfigurable memory module that provides the abstraction of the distributed memory available in the FPGA. Each block of memory has a size of 18432 bits, and can be available in different organizations, e.g. 512 lines x 36 bits per line. The Virtex-II Pro FPGA family has the physical block memory placed in columns equally distributed in the area of the FPGA. The total amount of BRAM memory available in an FPGA is relevant to the size of the FPGA. The bigger the FPGA the more available block ram embedded.

The PLB IPIF controller is a Xilinx soft-core [22], which targets to make the connection between User IPs and the PLB bus easier. It provides a bi-directional interface between a User IP core and the PLB 64-bit bus standard. For the purpose of this study two PLB2IPIF controllers were used, each one attached to a PLB bus. Both of them are connected as slave peripherals to the PLB. They were used as an additional way for the processor to access the external DRAM memory, through the DDR MUX block. The memory accessed this way is supposed to be private for each processor. However, for debugging purposes only, the PLB2IPIF controllers can be programmed to have a full view of the DRAM memory.

2.5 PLB2Cache module

PLB2Cache module is one of the two modules that connect the PowerPC with the rest of the units available on the system. Figure 2.4 presents the connection between the PowerPC, the PLB bus, and the PLB2Cache module. As it can be seen, the PLB2Cache module stands between the processor and the PLB bus. Its purpose is to examine all the requests generated by the DCU and filter out those that access shared memory. Filtered requests are forwarded towards the coherent memory system, while requests that access private memory are served by the PLB bus. Filtering of the accesses is based on the same signals that are used to connect the DCU to the PLB.

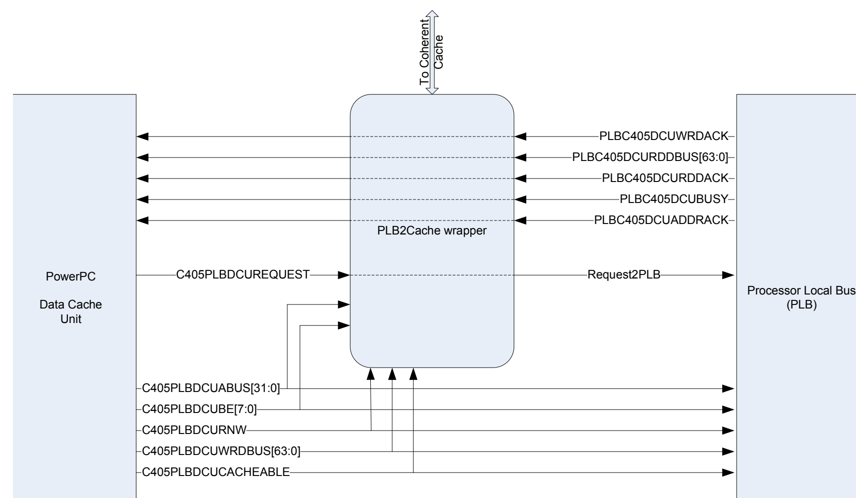


Figure 2.4: Connection between the PowerPC, the PLB2Cache module, and the Bus

As it is shown in the figure the signals leaving the DCU C405PLBDCUABUS, C405PLBDCUBE, C405PLBDCURNW, C405PLBDCUABORT, C405PLBDCUWRDBUS, and C405PLBDCUCACHEABLE are connected to both the PLB and the PLB2Cache module. Signal C405PLBDCUREQUEST, which was until now connected directly to the PLB bus, it passes through the PLB2Cache module. Every time the DCU generates a new PLB request, the PLB2Cache module examines the address. If the address accessed lies in private memory the request propagates through the PLB2Cache module reaching the PLB. Otherwise, the propagation of the request is stopped in the PLB2Cache module, the PLB never becomes aware of the request, which is forwarded and served by the coherent memory system. In the other way around signals PLBC405DCUADDRACK, PLBC405DCUBUSY, PLBC405DCURDDACK, PLBC405DCURDDBUS, PLBC405DCUWRDACK, and PLBC405DCUSBUSYS that were connecting the PLB to the DCU are now routed through the PLB2Cache module. They are driven by the PLB when a read PLB access returns data or by the PLB2Cache module when a shared memory read returns data.

Placing the PLB2Cache module between the processor and the bus offers many advantages. Accesses to shared memory are not served by the PLB bus. This means that they don't sustain the timing costs of the arbitration and the sharing of the bus with the ICU, which is also connected

to the PLB. For these accesses, the DCU transfers data at its fastest speed, since the custom design (PLB2Cache, Cache) is aware of the potentials of the DCU and tries to take advantage of them every time an external cache hit occurs. Furthermore, when the DCU operates in maximum speed, many accesses are acknowledged as soon as possible, which favours pipelined requests to take place. Finally, filtering out some memory accesses favours the rest of the instructions and private data requests to be served faster, since less traffic passes through the PLB bus.

2.5.1 PLB2Cache Module Architecture

Figure 2.5 depicts the internal organization of the PLB2Cache module. The organization of the whole module is based on the two FSMs that co-operate in order to make the DCU communicate with the coherent cache. The left one, FSM_PLB, is responsible for listening and filtering requests whose addresses lie in the shared memory region. It simulates the behaviour of a PLB slave peripheral in order to give the illusion to the DCU that is still directly connected to the PLB bus. The logic generated for FSM_PLB also contains the signals from PLB to DCU. When a request is served by the PLB, these signals are forwarded to the DCU. When a request is served by the coherent cache, equivalent signals driven by custom logic are forwarded to the DCU. The right one FSM, FSM_ACCESS, is responsible for translating the PLB protocol to a simpler format. This format is used from the PLB2Cache module in order to communicate with the coherent cache. The need of having two FSMs in parallel comes from the intention to operate the DCU in its maximum speed for the shared memory accesses. In order this to become feasible, both read and write accesses must complete (with respect to the DCU) as soon as possible. Read

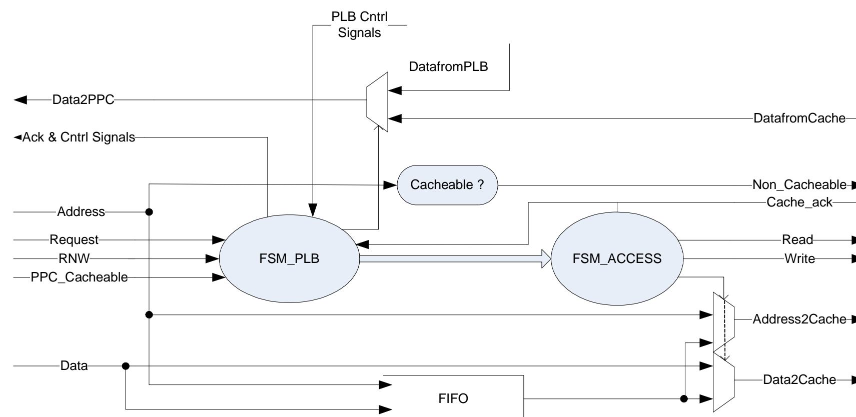


Figure 2.5: PLB2Cache Architecture

requests cannot complete before the requested data become available, however, write requests can complete in two cycles time. Additionally, in any kind of request (read or write) the first phase, which has to do with the acknowledgement of the address, can also complete very fast, specifically in the first cycle of the request. Following this tactic the DCU will operate in its maximum speed and the number of overlapping data accesses will increase, since requests are acknowledged immediately.

No matter how fast requests are acknowledged, they cannot be considered completed until the action is considered to have finished for the rest of the coherent memory system, too. Ac-

knowledging requests that fast requires storing them in a queue in order to be served in the future. This is accomplished by the use of a FIFO, in which pending requests are stored. The FIFO has a depth of 16 words. Every time the FSM_PLB identifies a new access to shared memory and FSM_ACCESS is busy with a previous request, the new request is pushed into the FIFO. If it is a read request it will occupy only one word, otherwise a write request occupies two. During the cycle that the address of the request is acknowledged, it is also pushed into the FIFO. In the following cycle the data is also written, in case of a write access. When the FSM_ACCESS becomes available it will serve the first request in the queue. Since write requests are able to complete (with respect to the DCU) in two cycles time, the FIFO will usually contain some write requests that will have been gathered and will be waiting to be served. At some point in time the program will eventually generate a read request, which will be pushed at the end of the queue. The execution of the program will be blocked waiting for the read to complete. As it can be seen, acknowledging and queueing requests improve the performance in a short-term period. However, in a longer-term period a read access will face the cumulative delay of all the previous accesses, balancing the progress of the execution of the program.

Finally, within the range of shared addresses there is a sub-range, which corresponds to non-cacheable shared data. The addition of this feature was considered to be crucial, since many processors use non-cacheable accesses to read or write device registers or other special I/O components. The size of that space is parameterized and must be defined before the implementation of the system. The PLB2Cache module is designed to check some most significant bits of the address provided by the processor in order to identify a non-cacheable access to shared data. This happens in parallel with the examination of the address that recognizes a shared memory access.

2.6 Coherent Cache

The next step, after the PLB2Cache module, in the coherent memory system is the coherent cache. As explained in the previous chapter, the role of a coherent cache is twofold. It acts as a normal cache, providing access to commonly used data, but it's also responsible for maintaining the shared data coherent with respect to other processors as well. For this study, a bus-based coherence scheme was chosen to be implemented. Figure 2.6 depicts a coarse description of the coherent cache architecture. As it can be seen, the whole module is organized around the BRAMs that store data and tag information, and is divided in two parts. Part A is responsible for serving the processor's requests that come from the PLB2Cache module. Part B is responsible for sending requests generated by part A to the bus, and serving requests that are received from the bus. Both parts are responsible for maintaining coherency among data, thus, both of them must have access to tag information and run the coherency protocol.

Modern microprocessors achieve this by making a duplicate of the tags memory. In this study, the two parts take advantage of the BRAM architecture. BRAMs can be configured to have two two-port interfaces, without paying any area or hardware cost. Each interface provides read and write ports, and thus simultaneous access to data. However, care must be taken when both interfaces access the same address. The architectural definition of the BRAMs doesn't provide any guaranty of what can happen in this case, except for the fact that the hardware will not be damaged. Solution to this problem was given by clocking the two interfaces with clocks

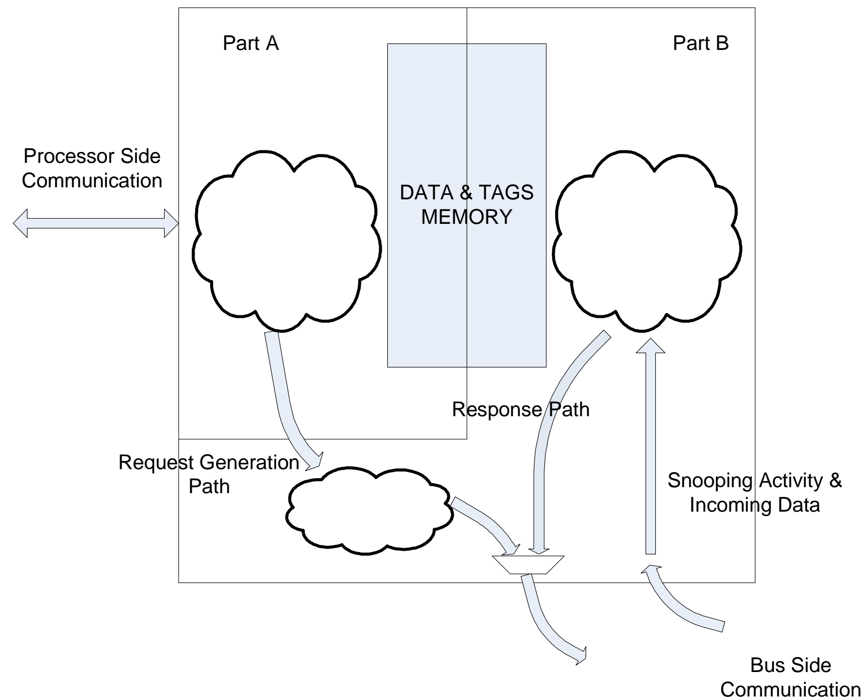


Figure 2.6: Coherent Cache General Description

of the same frequency but different phase. More details about that will be given below.

2.6.1 Cache Characteristics

The coherent cache was designed to have similar characteristics with the internal caches of the PowerPC, without losing the ability of making slight transformations in order to simulate different operational parameters. The size of the coherent cache is fully parameterized, while its associativity is programmed to be 2-way. The size of the cache line is fixed and set to 8 words per cache line. As far as the write policy is concerned, the cache follows the write-back scheme. In order to resolve block conflicts, the cache is equipped with an LRU algorithm. An LRU-bit corresponds to each set of cache lines. Every time an access occurs, this bit stores the information of which associate of the set served the corresponding access. Specifically, the bit is cleared if the access is served by the associate '0', or set to '1' if the access is served by the associate '1'. In this way the least recently used line within the set is unmarked. When a block conflict occurs, it is resolved by evicting the least recently used cache line of the set. Finally, in order to decrease cache miss penalty, the cache is organized to return the requested word within the line as soon as the word becomes available. The cache requests the missing cache line providing the address of the word that caused the miss. If the entity that responds to the cache sends the requested cache line transmitting the "critical" word first, then the cache follows the critical-word-first scheme. Otherwise, if the cache line is sent to the cache in a lowest-to-highest address order then the cache follows the early-restart scheme.

2.6.2 Tags and Data Memory Organization

The internal distributed BRAM blocks are used to store tag and data information. The amount of BRAM memory required is not fixed, since the size of the cache is parameterized. However, the organization of the BRAMs is predefined. A set of BRAM blocks is assigned to each associate in order to store data and tags information. The BRAM blocks that form the data part are organized in order to handle 32-bit wide words, while words that lie in the same cache line are stored in subsequent addresses. On the other hand, the tag-word has not a fixed size. Its size depends on the whole size of the cache, since the information stored contains a part of the address provided by the processor. Addressing of the data and tag parts is similar. The tag memory is addressed using the 'index' part of the initial address, while the data part uses the 'index' along with the 'block offset'. The index bits are calculated by the formula:

$$2^{index} = \frac{Cache\ size}{block\ size \cdot set\ associativity} \tag{2.1}$$

The block size and the associativity are constant, 32 bytes and 2-way, respectively. The size of the address given by the processor is 32-bit wide. The two least significant bits are not used by the cache. Bits 2 to 4 form the block offset, which is the address of each word in the cache line. Bits 5 to 5 + index - 1 are the index bits, and the rest bits of the address are the tag bits.

A tag-word includes the tag part of the address that corresponds to the specific cache line, 2 bits for the coherency protocol and one 'dirty' bit. Tag information for both the associates is being kept in different BRAM blocks. This is justified by the fact that two tag-words don't fit in 36 bits, which is the actual width of a BRAM block. If tag information for both the associates was supposed to be kept in a single word, then each tag-word should be at most 18-bits wide. This corresponds to 15 bits of tag size, which is equivalent to cache size of at least 256KBytes.

2.6.3 Coherency Protocol

The coherency protocol used in this study is a typical four-state MESI protocol. The protocol uses 4 states to encode the status of a cache line. These are: Invalid, Shared, Exclusive and Modified. Cache lines that don't store any data are marked as invalid. Shared means that the block is present in an unmodified state in this cache, the main memory is up-to-date, and zero or more other caches may also have an up-to-date copy. The exclusive state has the same meaning with the Shared, however no other cache has a copy of the block. Finally, Modified means that only this cache has a valid copy of the block and main memory is out-of-date. Figure 2.7 depicts a state diagram of the protocol.

Actions that cause each transition to occur can be seen next to the vertices. Table 2.1 shows the bus messages that will be generated for each one possible situation. PrRd stands for processor's read actions and PrWr for write. BusRd, Invalidate and BusRdX are the messages that are generated in order to maintain coherency. BusRd is generated every time a processor's read misses in the cache. Invalidate is generated when a processor's write doesn't have the privileges to complete, which means that the block is found in shared state, and BusRdX when a processor's write misses in the cache.

The letter 'S', in Figure 2.7, next to the actions denotes the status of the requested block in

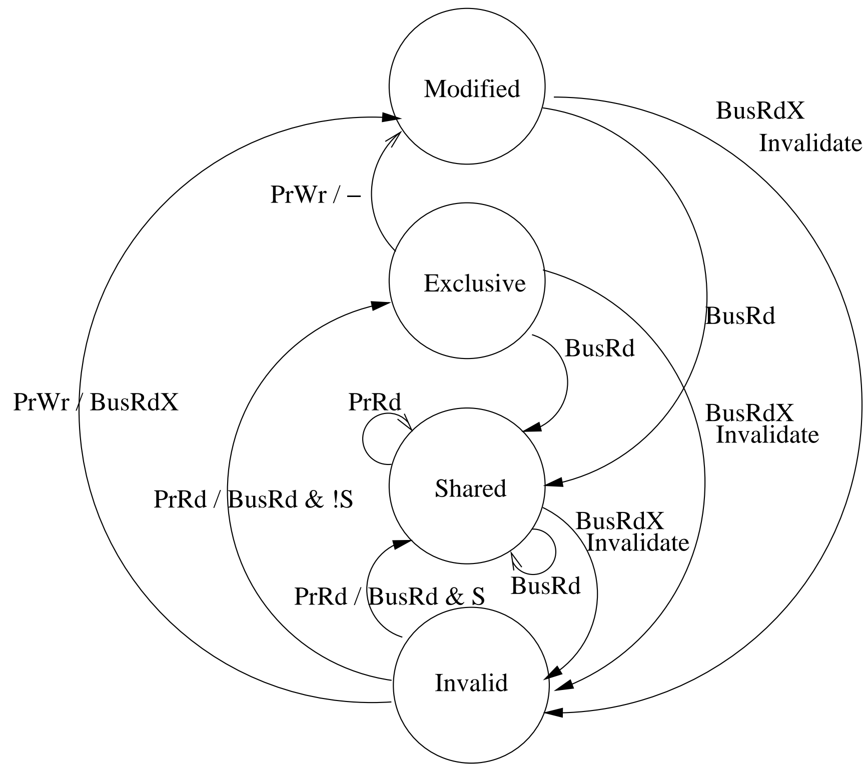


Figure 2.7: Coherency Protocol State Diagram

Block's State	Processor's Request	
	PrRd	PrWr
I	BusRd	BusRdX
S	-	Invalidate
E	-	-
M	-	-

Table 2.1: Generation of Coherent Requests

all the other caches. 'S' means that the block is found shared in at least one cache, while 'S' that the block is not shared at any cache. Transitions between states are caused by processor or bus accesses. When the block is first read by a processor a BusRd message is broadcasted. If a valid copy exists in another cache, then it enters the cache in the Shared state. However, if no other cache has a copy at that time, the block enters in the Exclusive state. When the block is written by the same processor, it can directly transition from the Exclusive state to Modified state, without generating any bus transaction. If another cache has obtained a copy in the meantime, the state of the block would have been denoted from Exclusive to Shared. In that case an Invalidate message

is broadcasted to notify every other cache to invalidate the copy they hold. When a processor tries to write a cache block but the block is absent from the cache a BusRDX message is broadcasted. The caches that hold a copy of the requested block will invalidate it. One of these caches will transmit the cache block back to the cache that requested the write privileges. The block enters the cache in Exclusive state and then immediately transits to Modified (with the completion of the write access). From the side of the bus, a BusRd message always results in a demotion of the block's state to Shared, if the block is found in the cache. A BusRdX message also demotes the block's state, if the block is found in the cache, but the resulting state is the Invalid state. Same things hold for the Invalidate bus message. BusRd and BusRdX messages always cause the transmission of a cache block. On the other hand, an Invalidate message doesn't create any other traffic, except for the message itself. That's the difference between the BusRdX and the Invalidate message. They both demotes the privileges of the cache block, when the block is found in remote caches, but only a BusRdX message requires a respond to be generated.

An extra property has been given to the cache, which has to do with the capability of receiving update messages. Any MESI-like protocol is based on invalidating copies of the same block when exclusive access is required. Thus, the kind of update messages supported in this study doesn't modify the state of the block updated. When an update message is issued (by a coherent network interface maybe, but not a cache) it is broadcasted to all the participants, including the main memory. The state of the block, wherever it resides, doesn't change. However, the data do change, resulting to everyone having an up-to-date version of the block.

2.6.4 Part A: Cache's Processor side

Figure 2.8 shows a block diagram for the processor's side part of the coherent cache. Part A operates with the same clock that the processor uses to generate requests. The same clock is used by the PLB2Cache module and the PLB bus. It receives the following signals from the PLB2Cache module: Read_Cmd, Write_Cmd, Address, Data_In and Non_cacheable_access. It returns an acknowledge signal, Cache_Ack, and the Data_Out signal. During a cache access all input signals are stable and valid, and remain so until the cache asserts the Cache_Ack signal. This signal remains high for only one positive edge in order to mark the completion of the access. If the access that finishes is a read access (cacheable or non-cacheable), signal Data_Out carries the requested data.

There are two FSMs that handle this part, FSM_CPU_ACCESS and WB_FSM. FSM_CPU_ACCESS manages every single request made by the processor and is also responsible for maintaining coherence. It is one of the basic blocks of logic that run the coherency protocol to decide the next actions that should be followed. Every time a cache block doesn"t have the appropriate sharing status for the processor"s access to complete FSM_CPU_ACCESS generates a bus message. The message is forwarded to part B in order to be broadcasted on the interconnection network. WB_FSM is responsible for handling write back activity. Write back activity corresponds to transferring modified blocks back to main memory. This transfer takes place when an access that misses in the cache requires a new cache block to be loaded in. The new block that comes in conflicts with the two blocks that are already present in the cache. These two blocks occupy the whole specific set, in which the new block is mapped. The replacement algorithm is called to resolve this conflict by choosing a block to be evicted from the cache. The block that is least recently used is chosen to leave the cache. If that block is 'clean' then the

incoming block just over-writes it. If, however, the chosen block has been modified, it must be written back to main memory.

A write-back transfer is not considered to be a separate bus transaction. It is hidden behind the block transfer that generated the eviction. That is feasible because the implementation of the bus offers different sub-buses for transferring data from and to the cache. Immediately after the bus request has been transferred, the write-back action is initiated. The address of the evicted block is first transferred and then the data in a critical-word-first fashion.

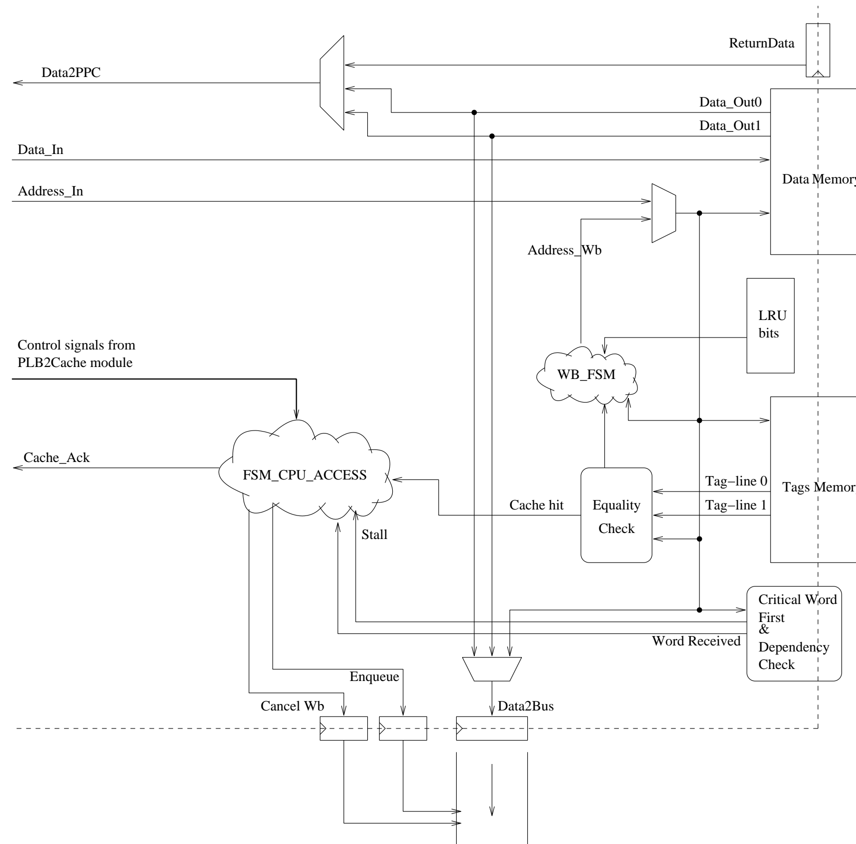


Figure 2.8: Part A block diagram

Apart from the two FSMs, other important entities in the figure are the equality check modules and the dependency check logic. The purpose of the equality check module is to compare (for equality) the tag part of the incoming address with the tag stored in each of the two associates. The use of the '==' Verilog operator is interpreted by the Xilinx flow as the instantiation of a complete comparator, from which only the equality operation is used. This adds extra area and timing cost. The solution to this was the implementation of a simpler module, which is checking bit by bit the tag parts of the addresses. Finally, the purpose of the dependency check module is to compare the addresses that are accessed by the processor and the bus. If both of them want to use the same cache block for conflicting purposes (read and write, write and write) an order must be enforced and one of them to wait. In this case the processor side is chosen to wait. The dependency check module blocks the processor's access until part B finishes its operation on the data. This module also implements the "critical-word-first" feature of the cache.

Dependency Check module

As mentioned above, concurrent accesses by the processor and the bus must be checked in order to verify that they touch different cache blocks. If not, ordering is imposed to them by making processor's access to wait for the completion of the bus access. Not all combinations of accesses require ordering even if they access the same data.

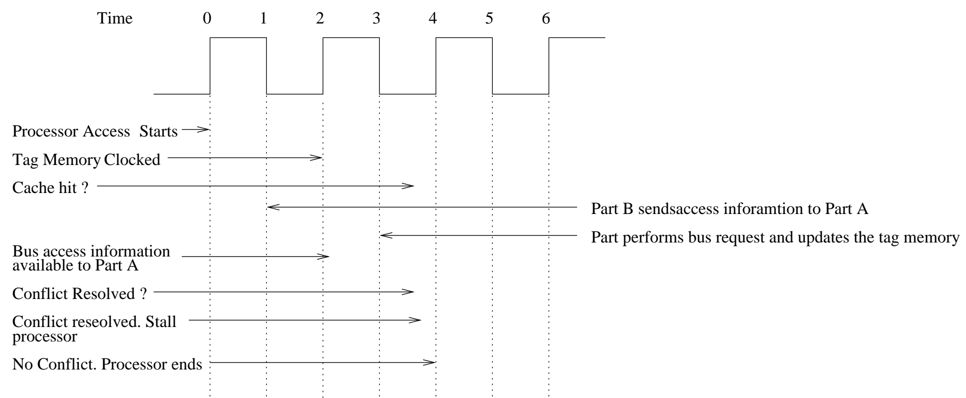


Figure 2.9: Two First Cycles of a Processor Access

Figure 2.9 presents the first two cycles of a processor's access and the two crucial periods, the periods that part B modifies the tag memory, of a bus access. The processor's access is placed starting at time 0 and finishing at time 4. Such an access is a cache hit. There would be no meaning to talk for cache misses, since they never conflict with bus accesses. During the first cycle, from time 0 to time 2, address becomes stable on the address pins of the tag memories. The memories are clocked at time 2 by the positive edge of the clock and return data somewhere between time 2 and time 3 (actually, 1.5 ns after time 2). Then the tags are compared with the address accessed to check for cache hit. Cache hit is resolved and at time 4 the access completes by either returning data to the processor, in case of a read, or data written in cache, in case of a write. Part B doesn't operate with the same clock. It uses a clock of the same frequency but different phase. Actually, it uses the negation of part A's clock. Thus, events as memory accesses in part B may take place in time 1 or time 3 etc. All of the bus requests shown in Table 2.2 may modify the tag information of the cache block they access. BusRd, BusRdX and Invalidate messages modify the tag and read the data, while Update messages modify both the tag and the data. Thus, processor accesses compete with BusRd, BusRdX and Invalidate accesses for accessing the tag memories, when both sides request the same cache block. On the other hand processor accesses must wait the completion of an Update message if they access the block that is being updated.

As far as the first scenario is concerned (BusRD, BusRDX and Invalidate messages), writing to tag memory may occur at any of the odd times. If it occurs at time 1 then the accesses are not considered to be concurrent. When part A accesses the tag memory in time 2 it reads the updated tags. No conflict occurs. The same holds when part B updates the tag memory at time 5. By then the processor's access has finished. The conflict arises if part B wishes to update tag memory at time 3. There are two possibilities in this case. If the processor performs a read request then it can be safely assumed that the request completes at time 2 when data and tags are read. Since

	BusRd	BusRdX	Invalidate	Update
Processor Read	No	No	No	Yes
Processor Write	Yes	Yes	Yes	Yes

Table 2.2: Combinations of Accesses

nothing is going to change for part A until time 4, when the request will officially be completed, then it can be said that there is no conflict between the processor and the bus access. On the other hand, if the processor attempts to write to the conflicting cache block then it is not safely to assume that the access completes at time 2. The data memory is updated at time 4. By then, however, part B will have already invalidate or downgrade the privileges on the specific block. In this case the processor write access is blocked, and a proper bus message is generated for the cache to request again access to the specific cache block.

As far as the second scenario is concerned (Update message), the update sequence starts writing in parallel the tags and the first word of the cache block. It continues writing the rest of the words in the next cycles. Again, if this sequence starts in time 5, then the processor's access and the update message cannot be considered to be concurrent conflicting events. The write action performed on the tag information of the block changes only the dirty bit by clearing it. A request that accesses a block that is being updated always hit, assuming the cache has the proper privileges for write accesses. The request, however, is blocked because the address accessed by the processor may not be available yet.

Blocking Mechanism and Critical-Word-First

The mechanism used to block processor accesses is rather simple. Every time part B receives a new bus message and is about to perform the requested operation it passes to part A the following information:

- the address requested (Addr_Refilled)
- the associate of the block to be modified (if this information is available, associate2Refill)
- a start/stop control signal (Refilling)
- a control signal to notify an incoming block (update or refill message, ReqStall)
- eight valid bits corresponding to the eight words of the cache block that is being modified (ValidBits)

This information is passed one cycle before the tag or data memory gets updated. For example, if a bus request is going to write to tag memory at time 3 (Figure 2.9) then all the above information will be written to part B's synchronization registers at time 1. It crosses domains at time 2, and becomes available to the processor side. Within this period, from time 2 to time 4, part A may use this information to resolve a conflict and stalls. For each one of the incoming messages the following actions are taken:

- Invalidate message: In case of an Invalidate message part B puts the address to signal Addr_Refilled, sets signal Refilling high to notify a new incoming message, sets signal ReqStall low and puts a predefined number (8'b11110000) to signal ValidBits. Part A compares the address given by the processor with the signal Addr_Refilled. If they match and the processor attempts to write this address then the processor access is considered to have missed. The coherency protocol generates a BusRdX request and forwards it to part B.

- BusRd and BusRdX messages: In case of a BusRd or BusRdX message, part B puts the address to signal Addr_Refilled, sets signal Refilling high to notify a new incoming message and sets signal ReqStall low. Also, it puts a predefined number (8'b00001111 for BusRdX or 8'b0 for BusRd) to signal ValidBits, and finally, identifies the associate to which the block is stored, by driving properly the signal associate2Refill. Part A compares the index part of the address given by the processor with the Addr_Refilled signal. At the same time comparison of the tag part of the address with the outputs of tag memories take place. In order to conclude that the bus and the processor wish to access the same address these two comparisons must come true. Specifically, the tag-equality between the associate defined by the associate2Refill signal and the address given by the processor must come true in order to be sure that all of the comparisons are related with the same block. However, the processor access will considered to be a conflicting one if it is a write access, as shown in Table 2.2. The coherency protocol will then initiate the corresponding coherency request. This will be an Invalidate message if the cache had received a BusRd request or a BusRsX message if the cache had received a BusRdX request.

- Update message and Refill Block: In case of an update message or a refilling block part B puts the address to signal Addr_Refilled, and sets signal Refilling high to notify the incoming of a new cache block. It sets signal ReqStall high and sets the bits of signal ValidBits high one by one, with regard the availability of each word. Part A understands that the processor's access conflicts with the bus access, but no coherency message is initiated. Part A waits for the specific requested word to become available. When that happens the processor's access completes normally.

2.6.5 Part B: Cache's Bus side

Part B is responsible for the communication of the cache with the bus. There are two sub-parts in it. The one handles the outgoing communication and it is also the least complex of the two. The second handles the incoming communication and the snooping traffic. It is also responsible of running a part of the coherency protocol by responding to bus requests. Both of these parts create and exchange messages with the adjacent cache or the DDR controller. The messages in this system are all designed to follow a typical format. The first word of the message corresponds to the address on which the operation will apply, while the op-code of the message is sent in parallel with the address from different control lines. If data accompany the request, as in the case of an Update message, then they are transmitted back to back with the address. Usually, the address of such a message corresponds to the address of the first datum of the sequence. It isn't necessary that a block should be transmitted in a zero-offset-word first. The transmission

may start at any offset and wrap-around to the start of the block. Cache block refills and block evictions are also organized to be bus messages. However, they are not broadcasted. A cache refill is a message that carries the requested cache block, which may reside either in another cache or in the main memory. The cache maintaining a copy of that block, or the DDR controller, sends such a message directly to the cache that requested the block. Furthermore, a block eviction is sent by the cache to the DDR controller. Finally, non-cacheable messages, which are also not broadcasted, and the data returned to a cache from a non-cacheable read share the same format. The uniformity given to the bus messages ease the design of the FSMs, which are responsible for the bus traffic.

Outgoing Communication

Figure 2.10 depicts a block diagram of this part. As it can be seen the whole logic is organized around the BUS_FSM. Requests generated by part A and block evictions all pass through a synchronous FIFO. Control signals and data from part A are synchronized as they cross clock domains. However, in some cases where half of a period is enough for particular actions, signals from part A are directly used.

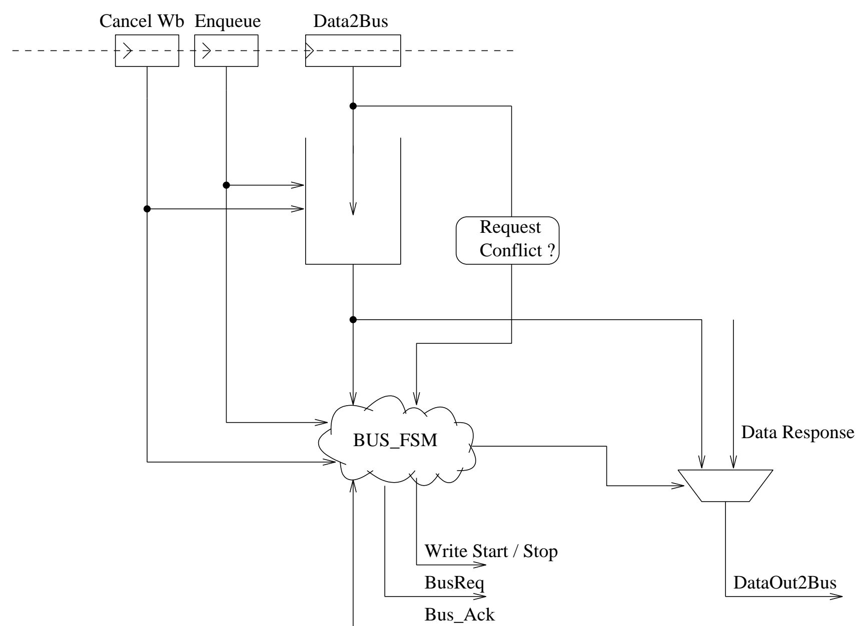


Figure 2.10: Cache2Bus Block Diagram

The synchronous FIFO used doesn't hold the properties of a Xilinx FIFO. Specifically, the first write in an empty Xilinx FIFO doesn't propagate to the exit. The module requires an extra cycle, during which the read enable signal is asserted, in order to output data. If this scenario was adopted a dummy cycle would have been added to the latency imposed on any request moving towards the bus. On the contrary a custom implementation of a synchronous FIFO was selected, which permits the first written datum to flow through.

BUS_FSM performs the required handshaking with the coherent bus. Two signals are used for this purpose; *BusReq* and *Bus_Ack*. The first one is driven by the BUS_FSM and indicates the

intention of this part to send a new message. When the bus arbitrator decides to serve the specific cache it raises the *Bus_Ack* signal to notify the availability of the bus. Bus messages and data are then dequeued from the head of the FIFO and sent to the bus.

BUS_FSM has also the responsibility to check any conflict between the incoming and the outgoing requests. The reason to do so is that there may be the need to change the type of the outgoing request. This need comes from the fact that an Invalidate request doesn't cause the transfer of a block. The scenario that threatens the coherency of the data is the case where the local processor sends an Invalidate message for a certain block, which is shared and wishes to write to it. At the same time a message broadcasted on the bus invalidates (Invalidate or BusRdX) the specific block. The message from the remote cache has come before the local message, and thus the remote processor will use the up-to-date copy of the block. If the local Invalidate message is not changed to BusRdX message then the local processor will operate upon the old copy of the block.

Incoming communication

Figure 2.11 depicts a block diagram of the part of the cache that handles the incoming communication and the snooping activity. The logic of FSM_REQ_IN is responsible for: accepting messages from the bus, delivering incoming data to the data cache, and also responding to requests by transmitting the requested cache lines. Furthermore, it passes data and control information to part A through the dependency check module, as described above.

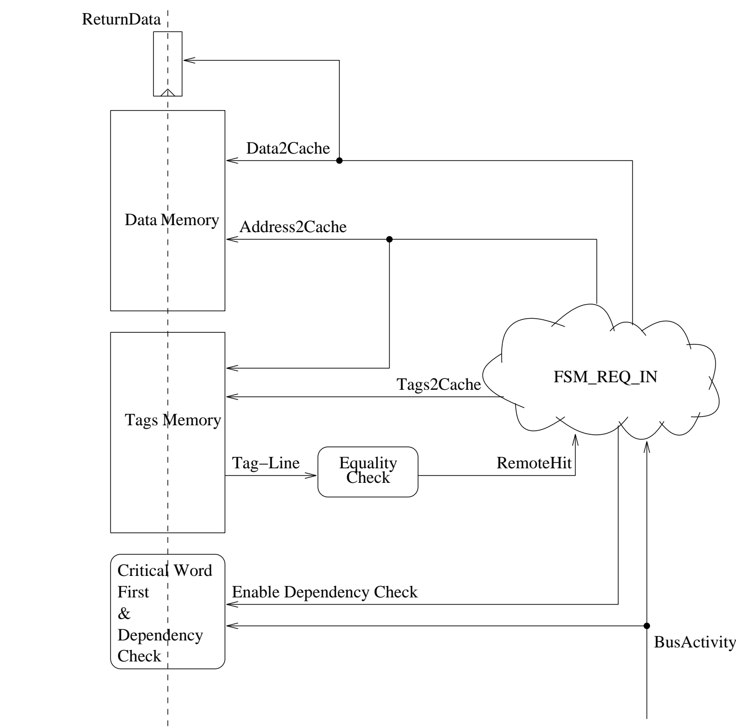


Figure 2.11: Bus2Cache Block Diagram

Upon the arrival of a new request FSM_REQ_IN tries to resolve if the requested block resides in the local cache. It does so by comparing the incoming address with the tags stored in

the tag-memory. If a miss is resolved no actions are taken concerning the sharing status of any cache block stored locally, and the request is ignored. In case of BusRd or BusRdX message FSM_REQ_IN also raises a signal to notify that the requested has missed in this cache. If a hit is resolved the next actions to take are decided by the coherency protocol. The sharing status of the requested block may be denoted or even invalidated, while the data may be updated (Update message). In case of a BusRd or BusRdX message FSM_REQ_IN undertakes also the responsibility to reply to the incoming request with the requested block.

FSM_REQ_IN also receives incoming data that have been requested by the local processor. This may corresponds to a cache block coming in, if the address accessed resides in cacheable shared memory, or to a single word, if the address accessed resides in non-cacheable shared memory. In the first case the block is written in the data memory of the cache; one word at a time. The requested word is also written in the *ReturnData* register in order to be sent to part A. In the second case, the incoming data must not be written in the data memory, as this will violate their non-cacheable property. The path followed involves the *ReturnData* register. The data are written only there in order to be sent to part A.

2.7 Coherent Memory Interconnect

The next level of the coherent memory system is the memory interconnect. Figure 2.12 depicts the architecture of the interconnect, which looks more like a switch than a bus. However, the logic implementing the control units of the module makes sure to provide all the advantageous properties of a bus. High performance systems have rejected the option of using a bus to connect multiple functional units. The reason for doing so is that bus architectures don't scale for many participants, as it has been proven by different studies. In this study, however, the number of the participants is fixed and it is predictable not to overcome the number three, the two processors and a coherent network interface (in the future). For so few participants, no scalability issues exist and the bus can manage to have a good performance. Furthermore, a bus is usually easier to implement and occupies less area.

The bus is designed to accept connections from up to three active participants plus a connection to an IPIF DDR controller. A participant is characterized as active when it has the ability to generate and respond to requests. The DDR controller cannot be characterized active, since it only accepts and responds to requests. The active participants are assumed to be entities that support snooping bus-based coherency, as it was described earlier, such as coherent caches. The number of the caches connected to the bus is parameterized and defined in implementation time.

Simultaneous requests from different interfaces are served in a round-robin fashion. Priority is given to the one considered to be the next in the round-robin order, or the closest one to it, if the next interface doesn't request the bus. When a request is chosen to be served it reserves the bus until its completion, which means that interleaved accesses are not supported. The FIFO's shown in Figure 2.12 stand between the bus participants and the DDR controller. Their purpose is to convert the 32-bit wide datapath of the coherent memory system to 64-bit in order to match the width of the DDR memory. Additionally, they are used to decouple the two points of scheduling. The first point is among the bus participants and the second is among the coherent memory interconnect and the two PLB_IPIF modules (in the DDR_MUX module that follows). These two points need to be decoupled in order to decrease the time required for some requests to

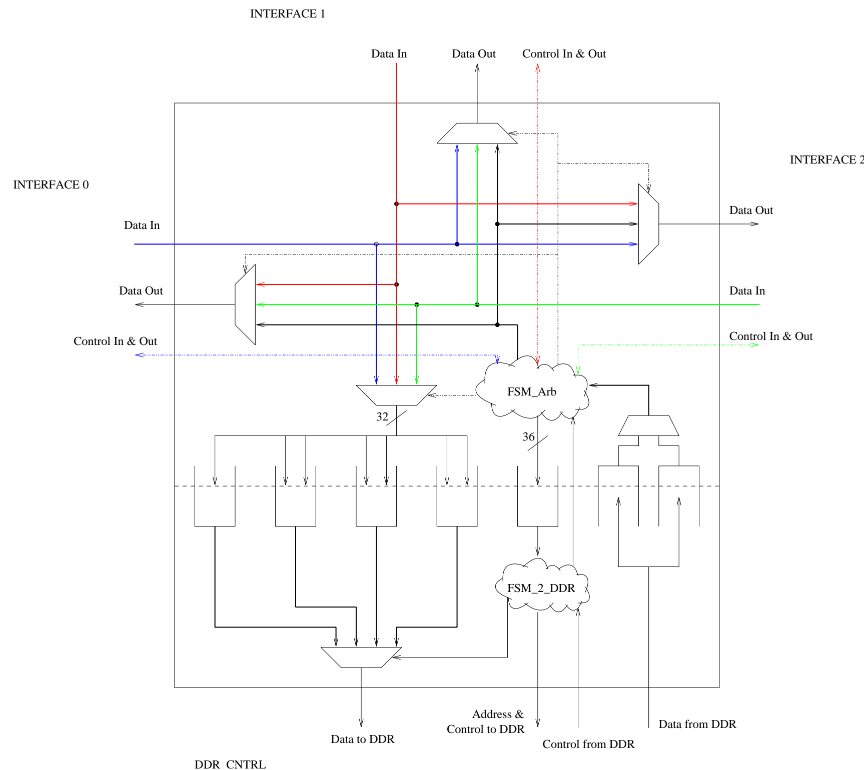


Figure 2.12: Coherent Memory Interconnect Block Diagram

complete. Otherwise, a non-cacheable write, for example, would block access to the bus until the word was eventually written to the external memory. The same hold for Update messages and also for requests that cause blocks to be evicted. The FIFOs provide a temporary place for storing data towards the external memory, giving the impression to the bus' participants that the requests have been completed. All the cases mentioned above had in common the fact that there were store requests. For all of these cases temporary storing decrease the required time or equivalently provides some sort of interleaving-ness. A subsequent bus request that can be served by a remote cache is executed in parallel with a part of the execution of the previous request. However, a load request that must be served by the external memory will experience the accumulated delay since pending requests residing in the FIFOs are served in order. The FIFO (Commands_fifo) shown in the figure that receives data from the FSM_Arb logic is used to store and place in order requests towards the DDR controller. All the other FIFOs are used to store data of different kind of accesses, such as block evictions (WB_0_x and WB_1_x), non-cacheable write (NC_fifo) and update messages. Finally, there is also a combination of FIFOs that deliver data to the coherent bus module. The FSM_2_DDR logic uses them to enqueue data read from the external memory. The FSM_Arb receives these data and forwards them to the coherent cache that requested them.

FSM_Arb and FSM_BUS_WB (not shown in the figure) are the two FSMs that handle the traffic between the participants of the bus. FSM_Arb decides the next request to serve and sends an acknowledge signal to the corresponding cache. Depending on the type of the request it either broadcasts it to the adjacent caches (BusRd, BusRdX, Invalidate) or forwards it to the DDR controller enqueueing it to the corresponding FIFO (Non-cacheable read and write accesses), or

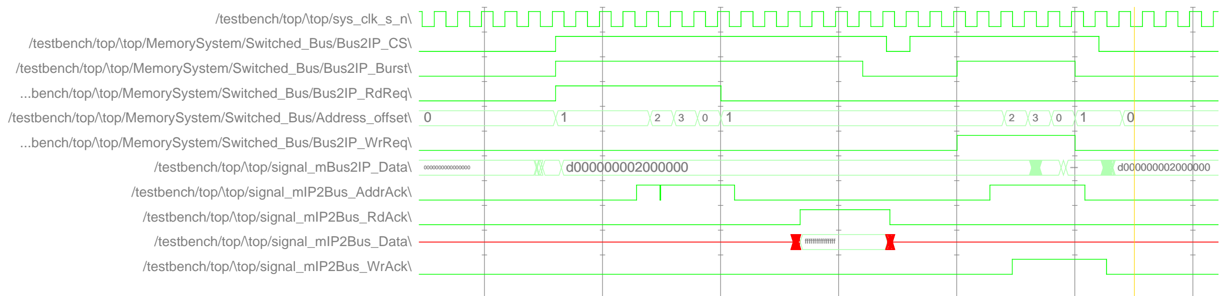
both (Update). BusRd and BusRdX request that hit in the remote cache result in a block being transferred as response through the bus. On the other hand, if these requests miss in all the remote caches the bus is notified and forwards the request to the DDR controller enqueueing it in the corresponding FIFO. Data returning to the . FSM_BUS_WB is much simpler and it is only responsible for the blocks that are evicted from any cache. It enqueues the evicted block and the write-back request in the corresponding FIFOs.

In order to decrease the loss of performance from the lack of interleaving-ness the messages on the bus are programmed to last as less as possible. However, not all requests have a guaranteed time of completion. Non-cacheable write requests and Invalidate requests always occupy two bus cycles before completion. Update requests take up 9 bus cycles, while BusRd and BusRdX requests that hit in the remote cache 11 cycles. On the other hand, the time required for serving non-cacheable read, BusRd and BusRdX requests varies, and depends on two factors. The first one has to do with the availability of the DDR controller. The DDR multiplexer may be busy serving one or more private memory requests by the time the coherent memory interconnect requests access to the external memory. The total amount of time spent depends on the number and the type of the private memory requests. The second factor that affects the duration of the above accesses is the occupancy of the FIFO on the DDR path. Some preceding shared memory requests may have already been queued towards the external memory but not sent to it. The new incoming request is blocked behind them, and is imposed the accumulated delay from all requests that lie in front of it. A small analysis for these two cases and the amount of the time required is made in the next chapter.

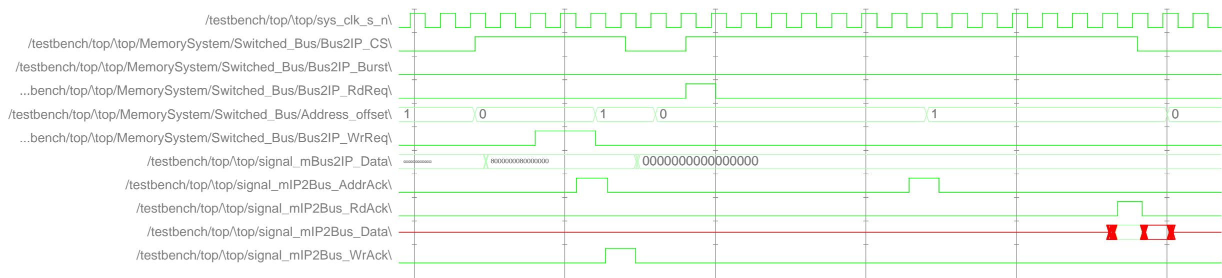
The FIFOs constitutes the point in the data flow where a crossing back to the clock domain clocked by the PLB clock is necessary. The type of the FIFO is similar to that used in the cache from part B to receive requests generated by part A. It has the ability to let the first datum written flow through but also synchronizes the two clock domains. The two PLB_IPIF modules operate using the PLB clock, while the bus and the rest of the coherent memory system use the negation of it. The DDR controller however can use only one clock. Thus, it is necessary all the three participants to operate with the same clock.

The third and final FSM in the coherent memory interconnect module is the one responsible for communicating with the DDR_MUX and the DDR controller. The DDR_MUX module doesn't change in any way the communication protocol, even though it stands in between the two entities. The DDR_MUX could also be omitted if there was no need of instructions and private data to reside in the external memory. In that case none part of the logic should change, except for some output signals that should be registered for 'place & route' purposes. The communication protocol used by the DDR controller is the IPIF. Xilinx provide some documentation about the details of the IPIF protocol and timing diagrams that should be followed. Unfortunately, the available documentation doesn't cover all the cases. In fact it doesn't cover some basic cases, which appear in this study. Thus, the precise timing diagrams were pulled out of simulations. The behavior of the IPIF interface was monitored by performing different kind of accesses. The timing diagrams acquired can be seen in Figure 2.13. These timing diagrams are also followed by the FSM_2_DDR which is responsible for this. There are also two figures in the appendix that show the behavior of the IPIF signals when a burst access is halted by a refresh action of the DDR memory. This important detail is also missing from Xilinx's documentation.

There is no need to describe in detail the FSM_2_DDR as its behavior is shown by the figures above. However, some things must be mentioned about the functionality of the logic as a whole.



(a) IPIF Burst Accesses (1 cache-block)



(b) IPIF Single Word Accesses

Figure 2.13: IPIF Accesses

The FSM waits for a new request to be available by reading constantly the empty signal of the Commands_fifo. As soon as the FIFO becomes non-empty the FSM starts functioning in order to generate the proper signaling. In case of write accesses, it gets the data to be written from the FIFOs. Single words for non-cacheable writes are read each time from the head of the NC_fifo. Written back cache blocks heading towards the main memory are read by the two pairs of WB_0_x and WB_1_x FIFOs. WB_0_x and WB_1_x are organized in pairs of sub-FIFOs in order to transform the 32-bit data path of the coherent memory system to 64-bit data path in order to match the width of the DDR controller. Each pair has the capacity to store a single cache line. The same holds for the FIFOs that store the blocks to be updated before they reach the main memory. Update and write back commands generate burst write accesses of 8 words, while non-cacheable write commands single word write accesses. On the other hand, non-cacheable read accesses generate single word read accesses to the DDR controller, while BusRd and BusRdX messages burst read accesses of 8 words. Data read from main memory are pushed into the pair of DataIn_fifo_x. Upon arrival of the first word the other part of the bus is notified to start the creation of the message that will be sent as response to the cache.

2.8 DDR Multiplexer

A step before the DDR controller is the DDR multiplexer module. The module of the DDR controller is supposed to be connected to only one entity. It is not designed to accept requests by multiple entities. The purpose of the DDR multiplexer module is to provide connectivity to multiple entities, and specifically to the coherent interconnect and the two processors. It receives

the IPIF signals from the coherent bus and the two PLB_IPIF modules and outputs a set of IPIF signals towards the DDR controller. At each point in time the DDR multiplexer selects one of the three input interfaces that requests access to the external memory and connects it to the output interface. If any other interface requires access to the DDR memory at the same time, it waits until all the previous requests have completed and also the controller to become available. In this way, to each one of the input interfaces is given the illusion that it is connected point-to-point with the DDR controller. Additionally, the controller has the impression that receives requests from only one source, the one that is connected to.

The DDR multiplexer module is in general a simple round-robin selection of the input requests. The logic of the module identifies the interfaces that require access to the controller by the status of the Bus2IP_CS signal. The Bus2IP_CS signal rises high to notify the start of a new IPIF request and remains high until the end of the request. The only complexity in this module comes from the single-word IPIF accesses. Observing again Figure 2.13, someone may notice that in a single-word read signal Bus2IP_RdReq stays high for one cycle only, and then returns to '0' without the initiator of the request to receive any kind of acknowledgement. If such a request occurs when the DDR controller is busy serving a previous request from a different interface, then the type of the request, which is the rise of the Bus2IP_RdReq signal, will be lost. The scheduler will eventually choose to serve the single-word access since the Bus2IP_CS remains high until completion. The controller, however, having lost the type of the request will remain inactive not knowing what kind of access to perform. In this way the system is driven to deadlock. In order to resolve this problem, each input IPIF interface is monitored by an FSM, named FSM_DDR_ACCESS, similar to FSM_2_DDR. Each transition of the signals is identified making the corresponding FSM to move forward to some state. If the signals transitioning are part of the input IPIF interface that has been selected to be served then no problem exists. Otherwise, when the corresponding IPIF interface is given access to the DDR controller, the corresponding FSM_DDR_ACCESS will be responsible to generate the proper IPIF waveforms towards to the controller but also backwards to the FSM that will be waiting for the proper acknowledgement.

2.9 DDR Controller

The DDR controller used in this study is provided by Xilinx and is part of the EDK library. It is fully parameterized in order to be able to communicate with different types of DDR DIMM modules. The required parameters that must be declared in the instantiation of the module can be found in the ".mhs" file of the EDK project. The data path of the controller is 64-bit wide and up to 32-bit wide addresses is supported. Cache blocks are transmitted in 4-word burst accesses. Requesting a part of the memory in critical-word-first fashion is also supported. A disadvantage of the controller is that it doesn't do any kind of specialized scheduling, to take advantage of the whole available DDR bandwidth. The controller just receives the next request and serves it. Finally, it was observed during the implementation of the system that the specific DDR controller doesn't work properly when operating in clock frequencies less than 100 MHz. This bug was also observed in [23].

Chapter 3

Evaluation and Verification

3.1 Hardware Resources

3.1.1 Target FPGA

The whole system has been designed for and implemented on a Virtex-II Pro FPGA, embedded in a Xilinx University Program board. The size of the FPGA is 30K and the speedgrade is -7C. The FPGA is equipped with two embedded PowerPC processors, as mentioned earlier, which are implemented as hard blocks within the circuitry. The resources available in each FPGA of the specific family can be seen in Table 3.1. As it can be observed, the specific FPGA is rather a medium one. However, the attractive element of all the system was the low price of the XUP board, which enables a multimode system, out of many XUP boards, to be built.

3.1.2 Hardware Resources

Figure 3.1 depicts a floorplanned view of the whole design. Except for the modules presented in the previous chapter, there is also an I/O device that has been added to it. Specifically, an RS232 module is used to print data through the serial port of a host PC to a hyper-terminal-like program.

Device	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells	CLB (= 4 slices)		18 X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/O Pads
				Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VPX20	8	1	22,032	9,792	306	88	88	1,584	8	552
XC2VP30	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0, 8, or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0 or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VPX70	20	2	74,448	33,088	1,034	308	308	5,544	8	992
XC2VP100	0 or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164

Table 3.1: Virtex II Pro Resource Summary

The module is connected to an On-board Peripheral Bus (OPB). The OPB bus is connected to one of the two PLB busses using a PLB2OPB bridge.

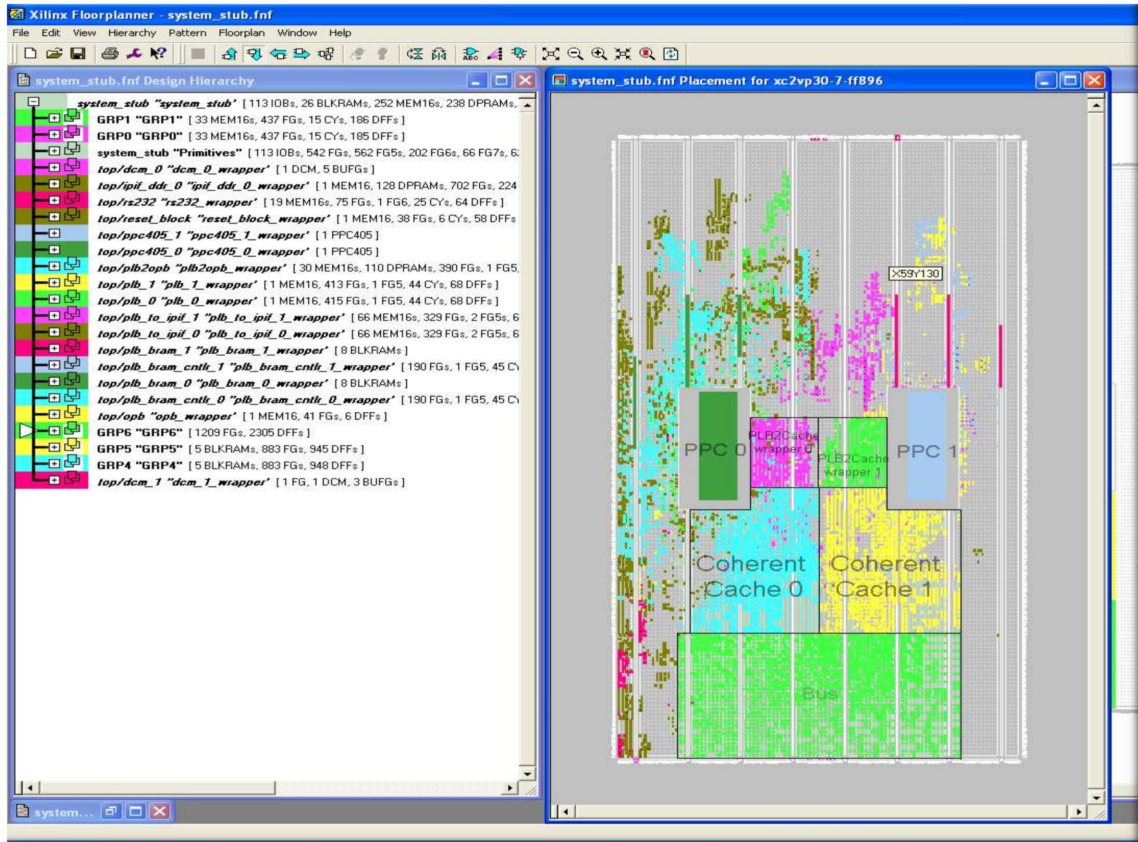


Figure 3.1: Floorplanned view of the system

Table 3.2(a) presents the utilization of resources required for the system described above. The numbers presented corresponds to the whole experimental system, including parts, such as the I/O path, that are not relevant to the coherent system. The resources occupied by the coherent part of the system are shown in Table 3.2(b). The numbers presented there are those reported after the "map" procedure of the design.

As it can be seen, the whole system occupies half of the logic-related resources available in the chip. Specifically, 51% of the available slices host logic of the system. The 33.4% is occupied by the implemented coherent system, while the rest 17.6% by soft-cores provided by Xilinx. As far as the memory resources of the system are concerned, only 26 out of the 136 available BRAM blocks are used. 10 BRAM blocks are dedicated to the coherent system for the implementation of the 2 4Kbytes coherent caches. The rest of the 16 blocks form the private memory available to the processors, 16Kbytes to each one of them. Finally, only 20% of the available IOBs are used, mainly for communicating with the external DDR memory. The rest of them can be easily used by a network module, which will provide connectivity with the rest of the world.

(a) Utilization summary for the whole system

Resources	Occupied	Available	%
FFs	6,943	27,392	25
LUTs	9,876	27,392	36
Slices	7,051	13,696	51
BRAM	26	136	19
IOBs	113	556	20
PPC	2	2	100
GCLKs	7	16	43
DCMs	2	8	25
Equivalent Gate Count	1,910,012		

(b) Utilization Summary for coherent system only

Block Type	FFs	LUTs	Slices	BRAM
2 x PLB2Cache	390	965	492	0
2 x CCache	1,959	2,533	1,830	10
Bus	2,247	2,914	2,048	0
DDR_MUX	52	349	218	0
Total	4,648	6,761	4,588	10

Table 3.2: Hardware Utilization

3.1.3 Timing Conciderations

The clock frequency of the system is constrained by two factors that constitute the upper and the lower ceiling. The first factor comes from the inability of the Xilinx DDR controller to operate in clock frequencies lower than 100 MHz. This behavior is a documented bug and has also been reported in [23]. Even the most recent version of the controller, which comes along with the latest version of EDK 8.2 software, has this disadvantage. Other possible frequencies that are not turned down by this factor are these above 100MHz, which are also supported by the external DIMM module. On the other hand, the complexity of the system implemented restricts the use of high frequencies. As it will also be shown below, the system is not able to operate above 100 or 105 MHz. Thus, the cut of these two sets of possible frequencies (100MHz), which meets all the criteria, is chosen to be the frequency of the whole system.

Critical Paths

Some of the critical paths of the system are presented below in order to identify which functionality requires the biggest part of a clock period. Table 3.3 presents these paths and their required time. The first conclusion that it can be drawn is about the cost of the routing delay. In any of the cases presented, routing delay corresponds at least to the two thirds of the total delay, apart from the first case where it possesses the 64%.

The specific case corresponds to the read cache hit scenario. The tag memory has just been clocked and output the corresponding tag lines. Address and tags are checked for equality. Cache hit is resolved and data are returned to the processor. A detailed report of how the delay is split

Action	Combinational Delay	Routing Delay	Total	Available
Read hit	3.526ns (35.9%)	6.294ns (64.1%)	9.820ns	10ns
Read miss ending	1.459ns (29.4%)	3.511ns (70.6%)	4.970ns	5ns
Bus request	0.541ns (24.3%)	1.689ns (75.7%)	2.230ns	5ns
Bus transaction	2.956ns (29.9%)	6.915ns (70.1%)	9.871ns	10ns

Table 3.3: Delay of Critical Paths

for this and the rest of the cases can be found in . As it can be seen, almost three ns are paid in order to move from hard blocks to and from the FPGA fabric. This time corresponds to tag memory's clock-to-output delay plus the time required to drive PowerPC's pins. The second case corresponds to the read miss scenario, when data are returned to the processor over the ReturnData register. Data cross between domains without being synchronized. Half of the period is given for this function. Here again, a great amount of time is required for the PowerPC's pins to be driven. The third case corresponds to the bus request action. In this case, too, data cross domains without applying any synchronization technique on them. Half period is devoted for the part B to receive a new message generated by part A and to raise the request signal towards the bus. Finally, the last case corresponds to a word moving from one cache to another. The bus is consulted its scheduling algorithm to choose the next participant that will be granted the bus. A word coming from the sender flows through the bus fabric and arrives in the remote cache.

3.2 Performance Evaluation

The next step of the evaluation procedure inspects the performance exploited by the system. As mentioned earlier, requests generated by a processor are separated to two groups. The first group includes requests that access private memory, and the second group requests that access shared memory. The former type of requests is not imposed to any type of additional delay. They spend the same amount of time as if there was no coherent memory system present. On the other hand, requests that access shared memory differ. The hardware that serves them is designed to take advantage of the full potentials the DCU is equipped with. As described in the previous chapter, too, accesses to shared memory, which hit in the coherent cache, make the DCU to operate in its maximum bandwidth. A write request to shared memory that hits in the coherent cache occupies the DCU logic for two cycles, before the processor become able to issue another request. A read request of the same type can occupy the DCU logic for five cycles, as Figure A.4 suggests. However, this number of cycles can be reduced to three, if the processor operates three times faster that the logic attached to the DCU pins. Specifically, the DCU is designed to wait for the current single-word read request to be satisfied before making a subsequent request. This requirement results in the delay of the three cycles between requests, as shown in that figure. During these three cycles the newly received datum is forwarded to the pipeline of the processor, in order the execution of the program to continue. The hardware that is responsible for this functionality is hidden within the processor's block. This means that it can also operate in the maximum frequency that the whole processor can. Thus, clocking the core of the processor three times faster (e.g. processor 300MHz - memory system 100MHz) than the rest of the outer

system, it results in fitting that three processor's cycles in a single PLB cycle. As a result of that, the duration of a read access can be reduced to three PLB cycles. Both read and write requests that hit in the coherent cache complete within the first two PLB cycles. This means that the coherent memory system doesn't add any excessive delay to these accesses. Figure 3.2 depicts a write-hit and a read-hit scenario. Given that the processor operates in the frequency of 300MHz and the rest of the system in the frequency of 100MHz, then the write hit costs 20ns and the read hit 30ns.

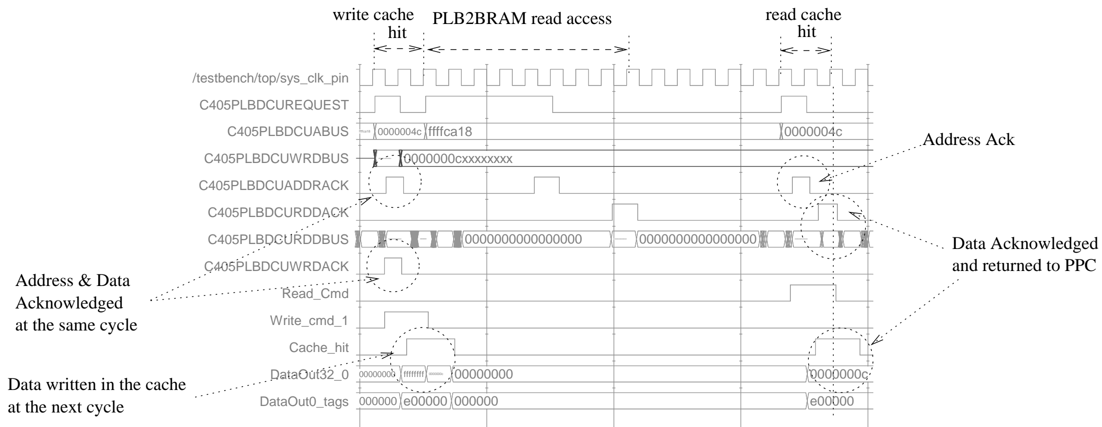


Figure 3.2: Write hit and Read hit Scenarios

Apart from the hit scenario examined above there is also the possibility of the request to miss in the cache. A miss can occur either if the requested data are absent, or the specific cache doesn't hold the proper privileges to apply the request. In the latter case, the cache issues an Invalidate request, which is broadcasted to all the bus participants. In this way the cache requires to be granted exclusive access to a block that wants to modify it. Figure 3.3 depicts such a scenario. It takes three additional cycles for the system to resolve this kind of miss, provided that the bus is not busy when the cache tries to send the Invalidate request. After the two initial cycles, the FSM of the cache recognizes that the write request doesn't have the privileges to complete. The Invalidate message is generated and crosses clock domains in the next negative edge of the processor clock, 2.5 cycles away from its initiation time. At this point the arbitration and the transfer of the message start. The bus is acquired by the cache, and one period later, in the next negative edge of the clock the message arrives at the remote caches. In parallel with this, part B of the local cache updates the tag-line of the block. The tag memory is written in the negative edge of the processor clock, 3.5 cycles away from the initiation of the request. Part A reads the tag memory, starting at the next positive edge. During the fifth period the cache hit is resolved and the data are written at the end of the cycle. Finally, there is also the possibility of spending one more cycle before sending the request, in order to check for possible address conflict between the outgoing message and an incoming one, which has just finished. In that case the total of five cycles is increased to six.

In the former case, where the block is absent a BusRd or BusRdX message is generated, depending on the kind of access the processor has initiated. A read request causes the generation of a BusRd message and a write request of a BusRdX message. Both of these messages induce a cache block to be transmitted over the bus towards the local cache. This block transfer may

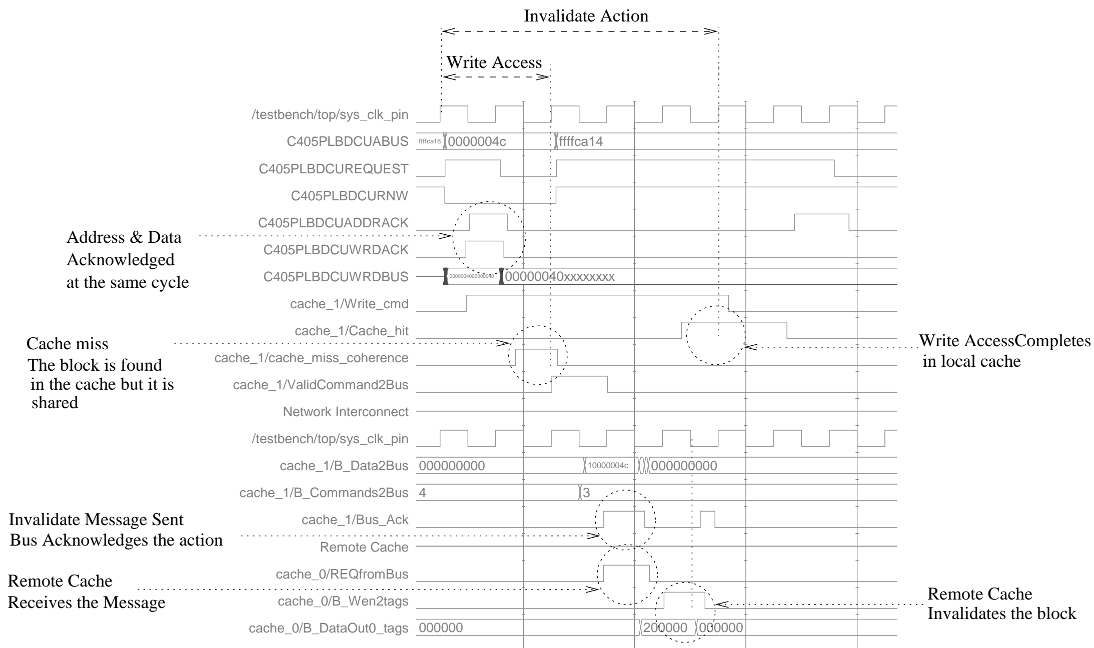


Figure 3.3: Invalidate Scenario

originate either from a remote cache or the external DDR memory. Depending on the entity of origination the status on which the block will be loaded into the cache changes.

The remote cache hit scenario, as shown in Figure 3.4, will be examined first. As also happens with the Invalidate message, the cache requests to be granted the bus 2.5 cycles away from the initiation of the processor's request. During this cycle the message is being sent, which results in driving the address pins of the remote tag memory. The memory is clocked in the next negative edge of the clock (negative edge 3). It outputs the indexed tag-lines and the check equality process starts. By the end of this period (negative edge 4) the remote cache hit is resolved. The response message is generated and its transmission starts in the next cycle (period 4.5 - 5.5). The cache always responses to a BusRd (or BusRdX) request in a critical-word-first fashion. The first word transmitted is the address of the message, while the second one is the word requested by the processor. This word arrives at the local cache at the next negative edge of the clock, 6.5 cycles away from the initiation of the processor's request, and it is being written in the ReturnData register. During the half period remaining until the next positive edge of the clock part A identifies the arrival of the requested word. In the meantime the tag-line of the block has been updated, giving to the processors the required privileges to perform the access. At that positive clock edge, 7 clock cycles after the initiation of the processor's request, the cache acknowledges its completion. The transmission of the whole response message completes 13.5 cycles after the initiation of the processor's request.

When the requested block doesn't reside in the remote cache then a copy of it is retrieved by the external memory. Figure 3.5 depicts a BusRd request that misses in the remote cache. The initial steps taken are similar to those of a remote hit request. The flow of actions changes when the remote cache resolves the miss, during the period of negative edges 3.5 and 4.5. At the end of this period the remote cache rises B_Request_miss signal, notifying the bus logic to

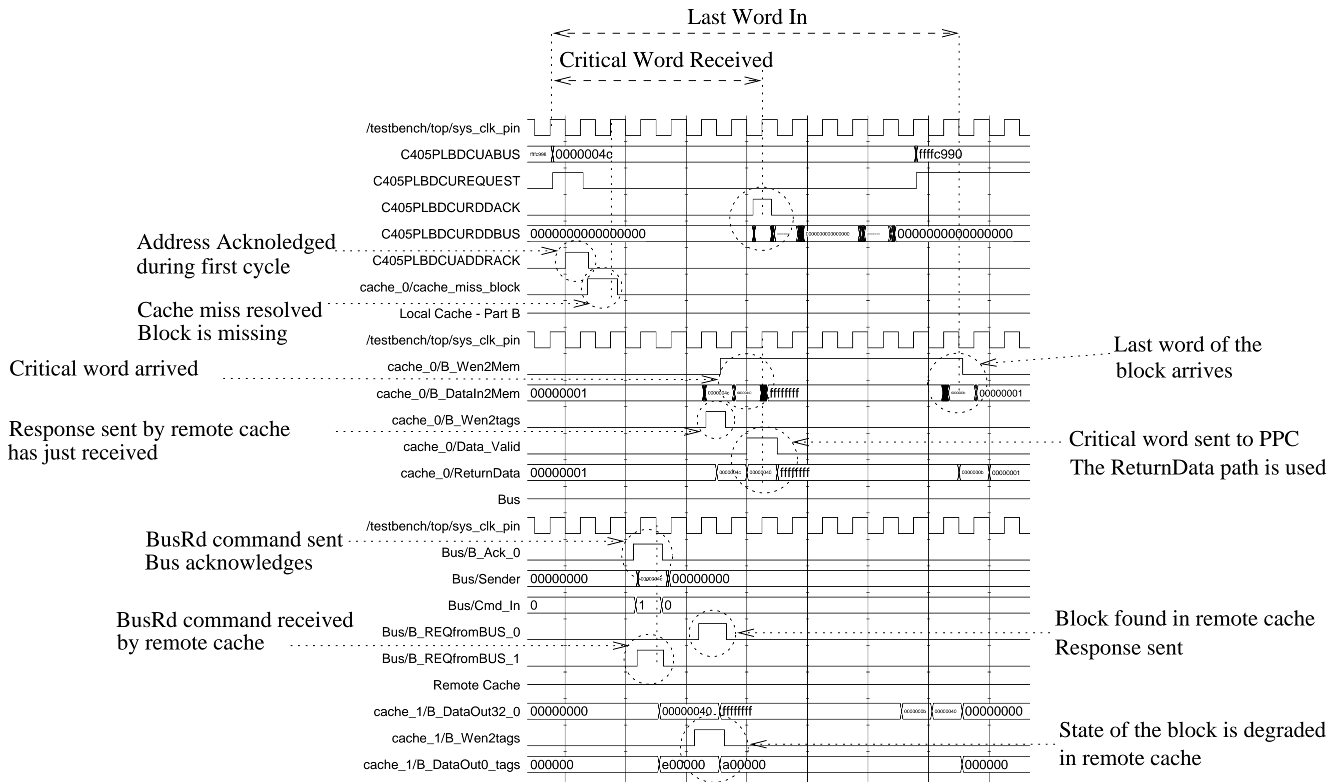


Figure 3.4: BusRd Remote Cache hit Scenario

forward the request towards the DDR controller. The request is written in the Commands_fifo at the negative edge. It becomes available for processing by the FSM_2_DDR at the start of the fifth cycle, provided that there were no other requests in the FIFO. The request is initiated towards the DDR controller and the data become available at the end of the 20th cycle. Specifically, the first double word from the read burst is written in the DataIn_fifo at the 18th positive edge. It becomes available to the bus logic starting the next negative edge. The requested word, which is the half part of the double word that has just crossed back to the bus clock domain, is immediately delivered to the cache. At the 20th negative edge it is written at the ReturnData register and at the next positive edge the processor's request completes.

In the same figure a write back action takes place. The write back message is sent back-to-back with the BusRd message. Its address is temporarily stored in the bus logic, while the first words of the block are written in the proper write-back FIFO (negative edges 6 - 14). When half of the block is stored, which means that the write back is sure to complete, the address of the block is written in the Commands_fifo as a request for a write burst. The burst is initiated at the start of the 22nd cycle and ends at cycle 30.

The two final accesses to evaluate are the non-cacheable ones. Figure 3.6 shows one non-cacheable write and one non-cacheable read. Both of them require one cycle less to reach the bus logic. Specifically, the cache is granted the bus and sends the non-cacheable access between the two negative edges 1 and 2. The corresponding command is written in the Commands_fifo. In case of a non-cacheable write the data that is sent back-to-back with the address are written in the NC_fifo at the next clock cycle. At this point a non-cacheable write can be considered completed

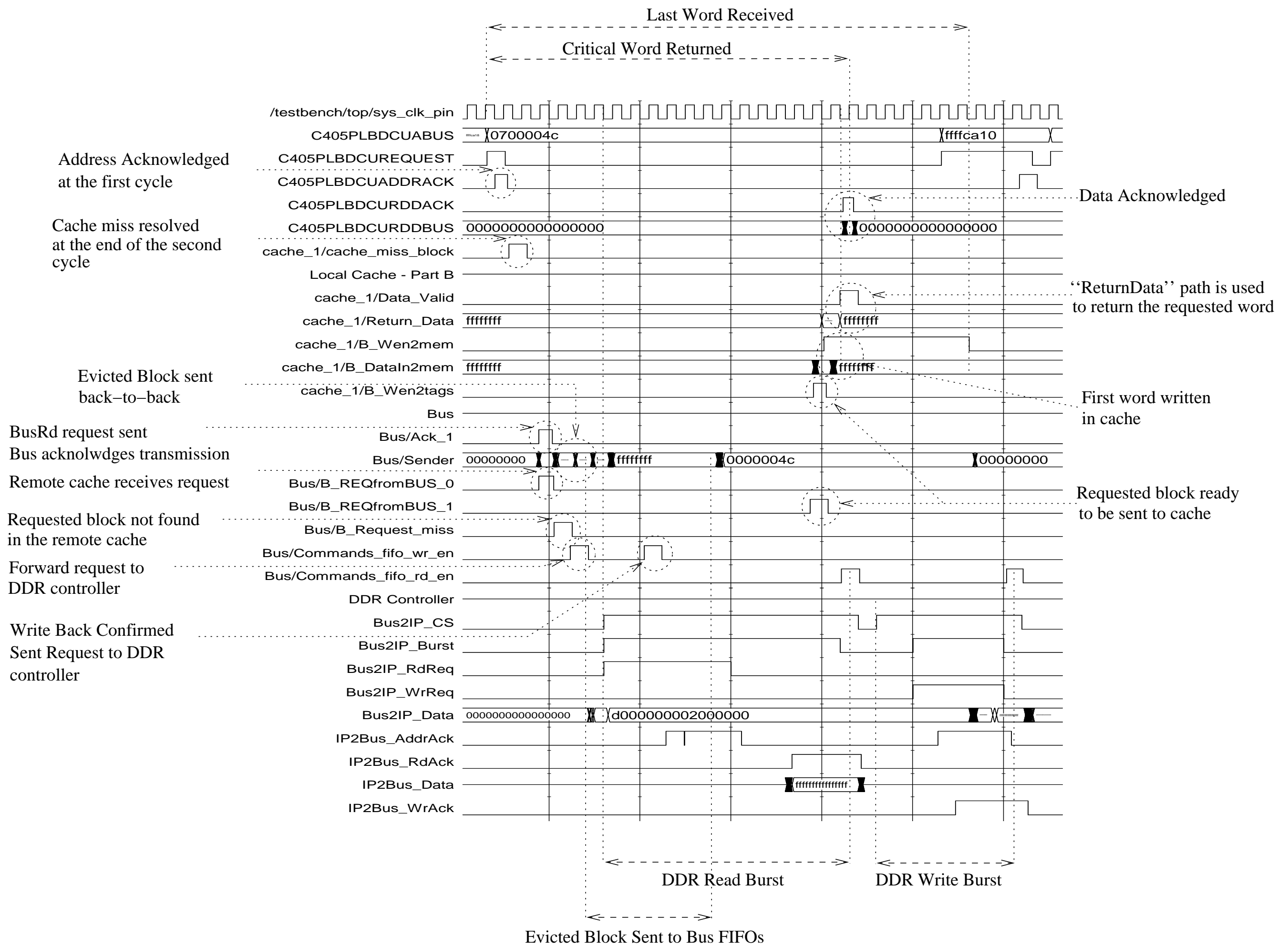


Figure 3.5: BusRd Remote miss. Fetching cache block from external memory. Block eviction at the same time

and the processor to continue the execution of the program. The address of the request becomes available to the FSM_2_DDR logic after the 3rd positive edge. A single-word access is initiated. The non-cacheable write completes at the 8th cycle. On the other hand, the non-cacheable read returns data at the 14th cycle. The word is written at the DataIn_fifo during this cycle. It then becomes available at the next negative edge and is delivered to the cache immediately. The access completes at the 17th positive edge when the data are sent back to the processor.

In all the above examples the request examined wasn't blocked at any point in the system. Possible blocking points are the FIFOs from the bus to the DDR controller and the sharing of the DDR controller. An access that finds previous requests to wait in the FIFOs is delayed for additional time. As it is observed by the figures above, this time equals to 6 cycles for each non-cacheable write request and 9 cycles for each burst-write request. These delays correspond to the duration of each type of access at the DDR controller increased by one cycle. The increment is required by the IPIF interface, which requests that the Bus2IP_CS signal to be driven low for at least one cycle between any subsequent accesses. No other types of actions (word read - burst

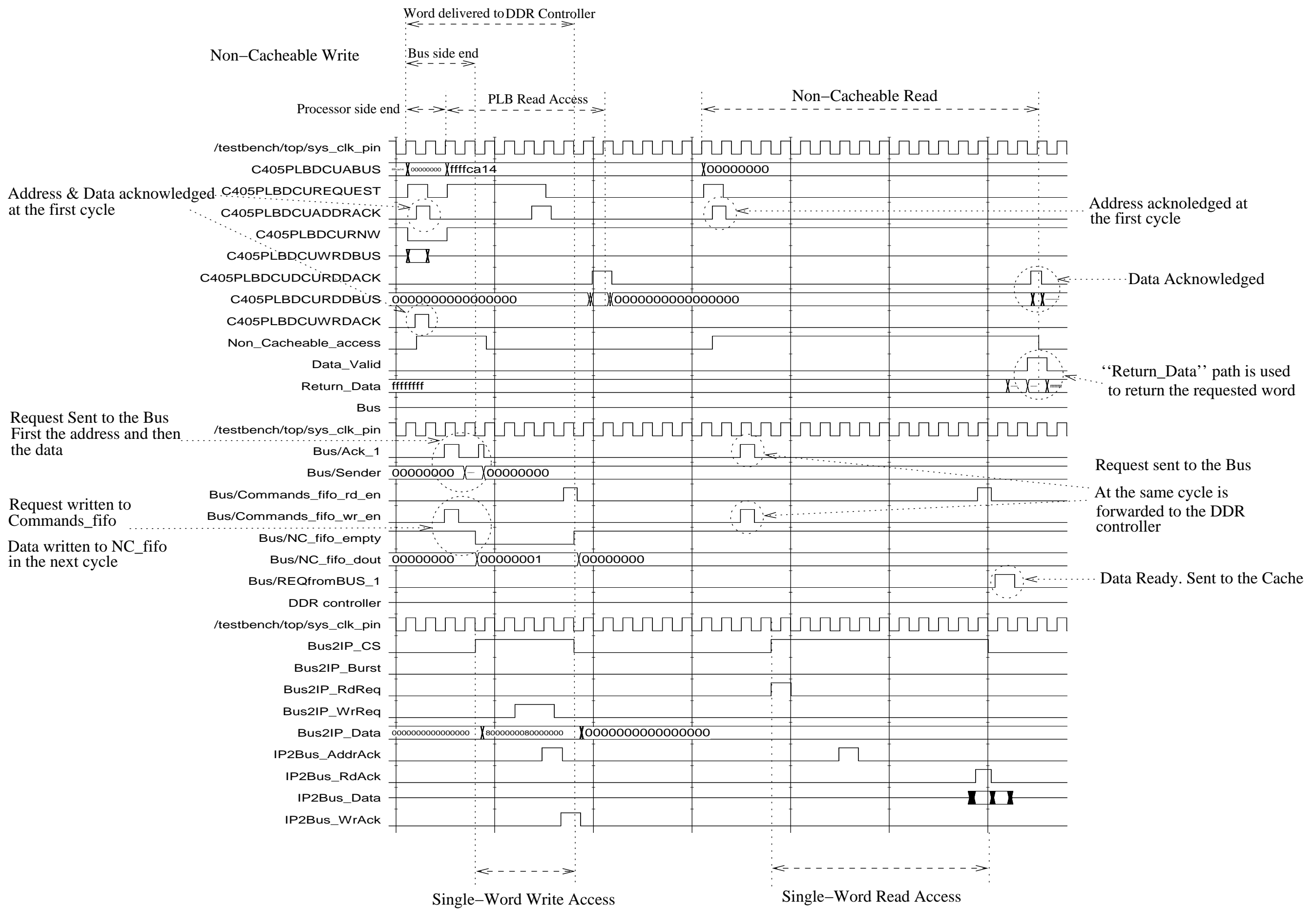


Figure 3.6: Non-Cacheable Write & Read Accesses

read) can be found to wait in the FIFOs when a new request is added. This comes from the fact that read accesses block the coherent memory system until data reception. Additionally, between accesses to shared memory, the DDR_MUX module may decide to serve any pending private memory requests. This will further increase the delay by 6 cycles for non-cacheable writes, 12 cycles for non-cacheable reads, 15 cycles for burst reads and 9 cycles for burst writes. Table 3.4 and Table 3.5 summarize the above results.

3.2.1 Comparison with other Coherent Shared Memory Organizations

As it has already been mentioned, PowerPC doesn't provide any hardware mechanism to maintain coherency between shared data. However, it provides the opportunity to manage coherency issues by using proper software. As it is proposed in PowerPC's manual [reference] only one processor is able to access shared memory at any given time. The processors agree on who will be the next one to acquire access to shared data. After having performed the required operations the specific processor flushes the internal data cache. It is obvious that while this approach may maintain data coherency, it isn't performance aware. Not only does it take many cycles to flush the cache, but also this action degrades the performance of the running process. It is clear that

Type of Access	Access Latency
Read Hit	3
Write Hit	2
Invalidation	5
Remote Hit	7 (critical word) / 13.5 (block transfer)
Block fetch from DDR	21 (critical word) / 27.5 (block transfer)
Block sent to DDR	delay hidden by the block fetch action
Non-cacheable write	2 (processor side) / 4 (cache side) / 8 (word delivered to DDR ctrl)
Non-cacheable read	17

Table 3.4: Access Latency (measured in PLB cycles - 100MHz clock freq.)

Type of Delay	Maximum penalty
Preceding Single-Word Write	6
Preceding Burst Write	9
Single-Word Write to Privatre Memory	6
Burst Write to Private Memory	9
Single-Word Read from Private Memory	12
Burst Read from Private Memory	15

Table 3.5: Additional penalties imposed to Shared Memory Requests (measured in PLB cycles - 100MHz clock freq.)

the approach presented in this study is more efficient than the one just mentioned.

There is also an alternative hardware solution to provide coherent shared memory to the two embedded processors. The DCU of both processors can be connected to the same PLB bus and use single-word requests to access any available memory connected to that shared PLB. If the shared memory lies in BRAM blocks then accessing it over the PLB costs 7 PLB cycles for single-word write requests and 8 for single-word read requests. However, BRAM resources are limited and it is more likely for shared memory to be held in external DDR memory. In that case, a single-word read takes 23 PLB cycles to return data to the processor. On the other hand, a single-word write occupies the processor for 10 PLB cycles before it is acknowledged by the PLB. After 10 cycles the processor is free to continue the execution of the rest of the program. However, the ongoing write access requires 6 mores cycles to be handled by the DDR control, for a total of 16 PLB cycles.

The conclusion that can be reached from this comparison is that the implemented system studied here is certainly faster than the above when external DDR memory is used. On the other hand, when BRAM is used to host shared memory then the comparison of these two systems must take into consideration the behavior of the program executed. The percentage of accesses that are served locally or remotely (but not from the main memory) in the implemented system is the most crucial factor that specify the outcome.

3.3 Correctness verification

A large part of the verification procedure was carried out in the implementation phase of the system. Each part was intensively simulated to check as many cases as possible. Furthermore, parts were put together and simulated in order to check their in-between communication. However, deeper investigation of the system is required in order to verify the correctness of the system. Larger programs were written to produce large amounts of coherent traffic. In this way the system's functionality was stretched out to cover all the various cases that it is supposed to support.

3.3.1 Software primitives

The implementation of any "useful" shared memory program requires the existence of the proper software primitives. Such primitives offer the ability to the programmer to manage the shared memory, to instantiate multiple execution threads and to protect shared data by locking mechanisms. Some of these primitives were implemented for the purposes of this study, while others were described indirectly. As far as the existence of multiple threads is concerned, EDK software is supplied with a library that provides the thread abstraction. The library provides memory management and locking mechanisms between the threads. Unfortunately, this library assumes that the created threads are able to run only on the *one* of the two PowerPC. Parts of the library are able to run simultaneously on the two processors, provided that each part doesn't interfere with the job carried out by the other. This results in two uniprocessor systems that are independent to each other, rather than a single multiprocessor one. Furthermore, a shared memory program is usually parameterized according to the number of processors available in the system. In this particular study, the number of processor is constant. Having multiple threads running on both processors doesn't provide the illusion of having multiple processors, since only one thread will be able to run on each processor at any specific moment. Thus, the approach of providing the ability to run multiple threads on both processors was rejected, without any loss of generality. Each test program was written for the specific number of processors, dividing the amount of work carried out in two parts.

On the other hand, the locking mechanism was considered to be crucial for the evaluation of the system, since most kinds of shared memory programs require some kind of synchronization between the processors. Two versions of spin-locks were developed. The first one is a simple implementation of the Peterson algorithm, which imposes strict alternation between the processors that request the lock. In order to provide a little more randomization to the behavior of the program, the second version of locks doesn't follow the strict alternation pattern. In that case, when a processor is trying to acquire an already granted lock it backs off for some time and tries again some time later. The implementation of these two releases can be seen in Figure 3.7. Finally, no barriers primitives were implemented. Any time one of the processors has to wait for the other to complete, it is constantly reading a predefined address to take a specific value. Using this handshake the semantics of a barrier are described indirectly. Finally, the shared memory of the system is managed "hardwired" by each application. This means that the needs of the program are well known a priori, when the program is being written. The programmer distributes the memory according to these needs. Again, the correctness and the performance of the program are not harmed, since the resulting behavior is equivalent to the case where software manages

the memory.

```
struct lock_st{ int ppc_0; int ppc_1; int turn; }

void mutex_lock(struct lock_st *lock, int processor){

    if(processor == 0){
        lock->ppc_0 = 1;
        lock->turn = 1;

        while((lock->ppc_1 == 1) && (lock->turn != 0));
    }
    else {
        lock->ppc_1 = 1;
        lock->turn = 0;

        while((lock->ppc_0 == 1) && (lock->turn != 1));
    }


void mutex_unlock(struct lock_st *lock, int processor){

    if(processor == 0)
        lock->ppc_0 = 0;
    else
        lock->ppc_1 = 0;
}
```

```
void mutex_lock(struct lock_st *lock, int processor){

  int delay;

  if(processor == 0){
     while(1){
         delay = 3;
         lock->ppc_0 = 1;
         if(lock->ppc_1){
             lock->ppc_0 = 0;
             while(delay--);
         }
         else break;
     }
  }
  else {
     while(1){
         delay = 1;
         lock->ppc_1 = 1;
         if(lock->ppc_0){
             lock->ppc_1 = 0;
             while(delay--);
         }
         else break;
     }
  }
}
```

Figure 3.7: Locking Algorithms

3.3.2 Shared Memory Programs

Testing Environment

The parameters of the systems (uniprocessor - multiprocessor) are shown in Table 3.6. The processors and all the remaining parts of the system operate in the same frequency (100 MHz). The internal instruction cache of the PowerPC is enabled in all the experiments, while the internal data cache is used only once for a specific organization running the “shared counter” program. The processors are operating standalone; no operating system is present. As a result these is also no memory translation. The addresses generated by the programs are physical. Finally, the size of each coherent cache is 4 KBytes.

Shared Counter

The first program written for the system is the increment of a shared counter. The two processors using a lock structure try to gain access to the shared variable that corresponds to the counter. Once the lock has been acquired by a processor the variable is increased by one and then the lock is released. Once the lock is released the other processor can acquire it to perform the same action. Using the first type of locks presented above, the system behavior results to an alternation of the two processors on having access to the shared variable.

The traffic generated by the two processors when running this program corresponds to the transfer of the lock structure and the shared variable from one coherent cache to another. The first action taken by the processor (processor A) to get the lock is a write access to its variable in the lock-structure. This corresponds to a BusRdX message to be broadcasted and the corresponding cache block to enter the cache. The same processor reads constantly the variable within the lock

Processors	Clock freq. = 100 MHz
	Internal Instruction Cache enabled (16KB)
	Internal Data Cache disabled
	No O/S. Processors operate standalone
	No Memory Translation - Physical addressing used
Coherent System - PLB - Peripherals	Clock freq. = 100 MHz
	Coherent Data Cache 4KBytes

Table 3.6: System Parameters

that corresponds to the other processor (processor B), in order to be notified for the its release. No traffic generated, since all the read accesses hit in the cache. Processor B eventually releases the lock by clearing its variable. A BusRdX message is generated and the lock travels back to processor B's cache. The next time processor A reads the lock a BusRd message is generated and the block returns to cache A. At that time processor A has access to its critical section. Its actions are to read the shared counter (generation of a BusRd message) and to write to it the new value (generation of an Invalidation message). Finally, it releases the lock by clearing its value in the lock structure. This set of actions is continuously repeated until the end of the program.

In order to measure the performance of the program, its length of execution is measured in processor cycles. The result is also compared against to the same time required by a uniprocessor system to perform the same kind of counter manipulation. Such a program is rather meaningless as far as its functionality is concerned, however, it provides a good insight of the cost of the shared memory synchronization. Table 3.7 reports the time taken for four different architectures to execute that program.

	200.000 iterations		2.000.000 iterations	
Architecture	int i;	register int i;	int i;	register int i;
Uniprocessor - Internal Cache Enabled	3,600,028	2,600,030	36,000,028	26,000,030
Uniprocessor - External Cache Used	9,400,044	3,400,023	94,000,044	34,000,023
Uniprocessor - Access to BRAM	11,400,041	4,600,023	114,000,041	46,000,023
Multiprocessor - Coherent Cache Used	33,200,308	29,500,275	332,000,308	295,000,276
Multiprocessor - Non-Cacheable Memory	33,449,129	30,516,658	334,489,656	305,149,990

Table 3.7: Duration of "Shared-Counter" program for different arcitectures in processor cycles (clk. freq. 100 MHz)

Specifically, there are three different uniprocessor architectures studied and one multiprocessor. The uniprocessor architectures are differentiated by the place where the variables of the program are stored. The first one uses the internal data cache of the PowerPC to enable quick access to frequently used data, such as the counter variable and the iteration variable *i*. In the second one, the internal data cache is disabled. The counter variable of the program is found in

shared memory cached by the external cache, while the stack resides in PLB BRAM memory. Finally, in the last uniprocessor organization data are not cached anywhere and they are loaded from and stored directly to private memory located in the PLB BRAM block. The multiprocessor organization used is the one described in the previous chapter. Two cases have been tested. In the first one the lock variables and the shared counter variable reside in cacheable shared memory, while in the second the shared counter variable has been moved to non-cacheable shared memory. For all the above organizations the instruction cache of the PowerPC was enebled.

The numbers shown above corresponds to two different executions of the program. The left one refers to execution length of 200.000 counter iterations, while the right one to 2.000.000 iterations. For each execution and for each organization, two lengths are reported in the two sub-columns. Each one of them corresponds to the declaration of the iteration variable *i* used for the *for-loop*. In the left column it has been declared as a simple variable, while in the right one the compiler was advised to maintain the iteration variable to a register throughout the execution of the program. This is the only hint given to the compiler, and the rest of the program is compiled with no further optimizations (for all the architectures to ensure fairness). Eliminating optimizations from the compiling process is required for the proper built of the multiprocessor program. In the start of that program there are several shared memory accesses that are used for the proper initialization of the system and the initial synchronization between the processors. These accesses seem meaningless to the compiler, which expects as input a uniprocessor program, and eliminates these instructions. As it is expected to be, the organization using the internal cache is the fastest one, for both types of programs. Following it, the organization using the external cache is the second fastest. As it has been mentioned before, the external cache takes advantage of the full bandwidth of the DCU interface, when accesses hit in it. The specific program is supposed to experience high hit ratio since the few words are always found in the cache. On the other hand, an access to the PLB BRAM private memory takes more cycles to complete, resulting to lower performance.

Moving to the multiprocessors organizations the number of cycles required for the execution of the program is multiplied by a factor of 3.6 to 10.0. The main reason for this has to do with the synchronization between the two processors. It can be estimated that for a bad scenario where both processors try to acquire the lock at the same time, only the messaging may cost up to 11.000.000 cycles. In a scenario like this, every access to shared memory initiated by the local processor, competes with the corresponding one initiated by the remote processor. Let assume, for example, the set of actions shown in Figure 3.8 in the sequence they appear. The total cost required for exchanging the corresponding messages generated by the processors come up to 110 cycles, while the whole execution increases the shared counter twice only. Actions 1, 7 and 11 cost 13.5 cycles since they cause a cache block to be sent from the one cache to the other, and also the start of the message is placed in sequence with the previous action. On the other hand, actions 2, 3, 4, 5 and 10 cost 11 cycles. They cause a cache block to be transmitted; however the start of the message comes in parallel with the previous action. This means that the message is ready to be sent but the bus cannot be granted, yet. The cycles of initiation of this message overlap with a request from the remote processor, and thus don't count in the total cost. Finally, actions 8 and 12 cost 5 cycles, while actions 9 and 13 2 cycles. If this set of actions was supposed to be repeated 100.000 times then 11.000.000 cycles would be spent, which corresponds to at least the 1/3 of the total execution time. However, this scenario is not likely to appear so many times in this specific program. Actually, it is expected, due to locking activity, that the two processors to

balance program execution in a state where at each point in time one of them has access to the shared counter, while the other one waits the lock to be released.

```
            Processor 0                        Processor 1

1)       lock->ppc_0 = 1;(BusRdX)
                                          2)     lock->ppc_1 = 1;(BusRdX)
3)       lock->turn = 1;(BusRdX)
                                          4)     lock->turn = 0;(BusRdX)
5)     while((lock->ppc_1 == 1)(BusRd)
         && (lock->turn != 0));           6) while((lock->ppc_0 == 1) (cache hit)
                                                 && (lock->turn != 1));

                                          7+8)   shared_counter += 1;(BusRd
                                                                & Invalidate)

                                          9)     lock->ppc_1 = 0;(Invalidate)
10)    while((lock->ppc_1 == 1)(BusRd)
         && (lock->turn != 0));

11+12) shared_counter +=1;(BusRd & Invalidate)

13)    lock->ppc_0 = 0;(Invalidate)
```

Figure 3.8: Competing for Access to Shared Memory

The last entry of the table corresponds to the placement of the shared counter in non-cacheable shared memory. The synchronization lock structure still resides in cacheable memory. The only things that changes in this case are accesses of the type of 7 and 8. The non-cacheable read request takes the place of the BusRd message and the invalidation message gives its place to a non-cacheable write request. The former exchange adds 4 cycles to the total amount of cycles, while the second 0 cycles. The additional cost when repeated 200,000 times corresponds to additional 800,000 cycles, which estimates the difference between the two last entries of Table 3.7.

Finally, it is easy to observe the relationship of the size of the workload with the execution length. Using a workload ten times larger results in multiplying the execution length by a factor of ten. This means that the behavior of the system is “locked” to a specific set of actions when this program is executed. The results measured are representative to that behavior, and as it should be expected they change linearly as the size of the workload changes.

Producer - Consumer

The second program implemented simulates a producer-consumer relationship between the two processors. Processor A is responsible for generating new data and placing them in the shared buffer. Processor B consumes these data by reading them from the buffer. The buffer lies in shared address space and is organized as a FIFO. It is equipped with a head and a tail pointer, which point at the start and the end of the queue, respectively. The producer processor appends new data at the end of the queue by updating the tail pointer properly. The consumer processor

retrieves new data from the head of the queue. Access to head and tail pointer is not protected by a shared lock, since these two words lie in subsequent cache blocks. Each time the producer processor wants to add a new word to the queue, it first writes the data to the memory location pointed by the tail pointer and then increments the tail pointer. On the other side, the consumer, which constantly reads the tail pointer, identifies the availability of shared data, by comparing head and tail pointer. In order to provide the illusion of processing the data, the consumer doesn't immediately consume the available data, and also it isn't necessary to consume them all at once. It decides, by calling the *rand()* function, how many words will be consumed. If this amount of data is present in the shared buffer, then it will also be dequeued. If not, the current iteration will end, and the consumer will start over. On every de-queue of a single word, the head pointer is updated. The first assumption made is that the FIFO has infinite space. The producer is *not* required to check if there is available space in the FIFO. It just appends new data. This property is translated to less traffic in the shared medium and also less competition for shared variables, and thus must be taken into account when studying the results. Finally, shared data, either for the uniprocessor or the multiprocessor program, are stored in the external DDR memory, and thus they are cached by the coherent cache. Local data, such as the stack, are stored in PLB BRAM. The programs of the producer and consumer is shown in Figure 3.9.

The performance of the system is compared against an equivalent uni-processor one. In such a system two threads are simultaneously executed on a single processor. One thread produces data and the other consumes. The scheduling policy between two threads is preemptive. Each thread is given a quantum of time to use the processor before the scheduler switches execution to the other thread. The same conventions as before are followed in this case, too. The memory of the shared buffer is infinite and the consumer program has the same behavior.

Table 3.8 shows the length of the execution of the two programs. Three reports are given for this program, each one referring to a different type of optimization applied during compile time. Table 3.8(a) reports the length of the execution of the program for one and two processors, when no optimizations are made in the compilation phase. The only hints given to the compiler are the declaration of some crucial variables as registers. The first conclusion that can be drawn is that the length of the execution of the program for each organization of the system is proportional to the size of the workload. Increment of the workload by a factor of x entails the increment of the execution time by the same factor. As it can be seen the multiprocessor system is always faster than the uniprocessor one. There are several reasons why this happens. First of all, the uniprocessor system has to pay the penalty of using threads in a single processor. Every time the producer thread runs, useful work is carried out. However, the quantum given to the consumer thread is not always used at its entirety. Every time the consumer empties the FIFO it then spins constantly reading the tail pointer, waiting for new data. Unfortunately, the producer is not able to enqueue new data since it isn't running. The rest of the quantum is wasted doing useless work. Furthermore, the actual switching between threads, and the interrupt handler running at the end of every quantum, have also an impact on the execution length of the program, and also constitute penalties not paid by the multiprocessor program. Finally, one thing that favors the multiprocessor organization is the existence of actual parallelism that can be derived by the two processors. Both of them can carry out useful work, since the producer thread produces data unstoppable and the consumer is able to withdraw them from the FIFO.

In order to reveal the penalty imposed to the uniprocessor system by the switching procedure the program is compiled again with optimization flags set on. Table 3.8(b) shows the performance

```
Consumer

for(i=0; i< ITEMS;){
  if(*head_pointer == *tail_pointer){
    items = (rand() % 32) +1;
    j = 0;

    while((j<items) &&
       (*head_pointer!= *tail_pointer)){
            k = *(*head_pointer);
            *head_pointer += 1;
            ++j;
    }
    i += j
  }
}
end = 1;
```

```
Producer

end = 0;

XTime_GetTime(&cycles_before);

for(i=0;i<ITEMS;++i){
  *(*tail_pointer) = i;
  *tail_pointer += 1;
}

while(end == 0);

XTime_GetTime(&cycles_after);
```

Figure 3.9: Producer - Consumer Program

of both systems for the same (optimized) program. The performance of the uniprocessor system has improved, since the compiler has managed to evict useless functionality (or has described it more efficiently). The same holds for the performance of the multiprocessor system. However, the relevant speedup has increased. The reason why this happens has to do with the amount of help each program receives by the compiler's optimizations. The uniprocessor version of the program takes some advantage of the optimizations, and this result to the lower execution time as regards with the previous version. However, a large amount of quantum of time will be lost again. Producer and consumer now work in a higher rate, but both work in the same rate. Wasting time by the consumer is again inevitable, since at some points in time the FIFO will become empty again. On the other hand, the multiprocessor version of the program takes full advantage of the compiler's optimizations. Thanks to them, the whole work is described with fewer steps. The system is able to take these steps without interruption, and finish earlier. This opposition is reflected to the relevant speedup of the system, which increases.

In order to understand the different kinds of costs in the uniprocessor program, the *rand()* function call has been removed from the code of the consumer. Now, the iterations of the consumer are much simpler. It first compares the head and the tail pointer. If it resolves availability

(a) No Compiler Optimizations

4-byte Words	Uniprocessor	Multiprocessor	Speedup
1,000	249,366	110,899	2.24
2,000	493,536	225,411	2.18
10,000	2,436,055	1,132,676	2.15
100,000	24,492,985	11,255,403	2.17
200,000	48,985,374	22,488,258	2.17
1,000,000	225,962,156	112,553,299	2.00
10,000,000	2,135,603,374	1,125,367,379	1.89

(b) Compiler Optimization Level -O2 (*rand()* used)

4-byte Words	Uniprocessor	Multiprocessor	Speedup
1,000	183,353	54,118	3.38
2,000	361,659	110,284	3.28
5,000	890,717	276,798	3.22
10,000	1,776,651	560,760	3.17
100,000	17,731,506	5,609,254	3.16
200,000	35,432,192	11,204,784	3.16
1,000,000	177,212,319	56,103,330	3.15
5,000,000	885,975,684	280,489,649	3.15
10,000,000	1,771,887,494	560,924,941	3.15

(c) Compiler Optimization Level -O2 (*rand()* not used)

4-byte Words	Uniprocessor	Multiprocessor	Speedup
1,000	152,236	44,084	3.45
2,000	297,417	88,173	3.37
5,000	733,501	221,458	3.31
10,000	1,460,060	443,593	3.29
100,000	14,567,150	4,442,037	3.27
200,000	29,127,419	8,884,761	3.27
1,000,000	145,619,237	44,426,176	3.27
5,000,000	728,070,541	222,133,438	3.27
10,000,000	1,456,136,058	444,267,532	3.27

Table 3.8: Duration of "Producer-Consumer" in processor cycles (clk. freq. 100 MHz)

of new data it dequeues the word found at the head of the queue. It updates the head pointer and then tries to execute this loop once more. The new performance of the uniprocessor system is presented in Table 3.8(c). As it can be seen, the execution time of the program has been further decreased, as result to executing less code during each iteration loop.The same holds for the execution time of the multiprocessor program. It also decreases, however the multiprocessor system seems to be favored again. Removing the *rand()* function call from the consumer program results eventually in "optimizing" in a way the whole application. As before, optimizing the program results in growing the performance gap between the two architectures. The multiprocessor or-

ganization is able to take full advantage of the optimizations and decrease the execution time at the lowest possible level. On the other hand, the uniprocessor architecture will be affected by this change; however, the execution time will decrease only by a fraction, due to wasted time when consumer finds the FIFO empty and also when switching between threads. This effect is reflected on the relative speedup between the two architectures, which increases with regard the results of Table 3.8(b).

A final comment that should be made here concerns the effect of doubling the available cache of the system when using the two processors. Due to the nature of the program, none processor can take advantage of the increased cache capacity when both processors operate. The consumer program uses only a single cache block from its available cache, either this is the one and only cache of the system or is the private coherent cache of the processor. This cache block keeps the next words to be consumed by the application. If the system disposes only one processor then this block will already be in the cache, because the producer program will have already touched it. On the other hand, if the system disposes two caches, the processor running the consumer program may have to pay the penalty of fetching it from the remote cache, where will certainly be. After the block has been consumed then it has no more meaning of existence, since none will request it. Furthermore, the performance of these two system is identical also when the producer program has gone ahead in execution and touches addresses “one cache size away”. In this case, both organizations will have to visit the external DDR in order to retrieve the next block to be consumed.

Sorting Algorithm

The third and final program implemented uses the two processors to sort an array of integers. Two versions of the program have been written. The first one sorts the given array in a manner that results to excessive computation to be carried out, and excessive communication to be generated. The purpose for doing so is to push the system to its limits, having to perform many tasks. The sort program was chosen to be implemented in these two forms, since it’s the most complex of all the three used. As far as its behavior is concerned, it follows the divide-&-conquer approach and the merge-sort algorithm to achieve its purpose. Both programs follow the three same steps. Each half of the table with the numbers to be sorted is assigned to a processor. At the first step of the execution each processor works on every cache block in separate. This corresponds to the base of the recursion in the merge-sort algorithm. The sorting of this basic element is done by a bubble-sort algorithm. The cost of this is considered to be a constant number in the total cost of the whole procedure. The two processors do this for all the cache blocks, and the first step ends with having table_size / cache_block_size different sorted subtables. The second step constitutes the merge process. Each processor takes 2, 4, 8 ... sequential cache blocks to form two already sorted subtables and use them to merge them in a bigger sorted one subtable. In order to do so it uses a temporal buffer, which then copies it back to the initial one. The second step ends with having the whole table of numbers split into two sorted subtables. In the third and last step, only one of the two processors works. With the same process it sorts the two remaining subtables, merging them in one, which brings the desirable result. The complexity of this algorithm is O(nlogn).

The two versions of the program differentiate in the way the initial separation of the table is made. The first version, which costs a lot, suggests that each processor to take on the odd

subtables and the other processor the even. At any point in time each processor has cache blocks required by the other processor. This results to excessive communication and lower performance. The second version, which is the one usually followed around the world, suggests each processor to undertake an independent part of the table to sort. In this case, where two processors are available, each processor is assigned one half of the table, either the first or the second. The two processors follow the steps described above to end up with the whole table separated into two sorted subtables. Finally, one processor undertakes the third step to produce the final form of the initial table. The performance of these two programs was compared against the equivalent uniprocessor merge-sort program.

4-byte words	uniprocessor	multiprocessor 1	Speedup 1	multiprocessor 2	Speedup 2
8,192	4,685,431	4,132,038	1.13	2,473,063	1.89
16,384	10,158,520	9,070,199	1.11	5,345,342	1.90
32,768	21,867,901	19,785,706	1.10	11,526,511	1.89
65,536	46,968,166	42,768,827	1.10	24,577,065	1.91
131,072	100,256,519	91,915,826	1.09	52,318,443	1.91
262,144	213,236,539	195,714,795	1.08	110,977,051	1.92
524,288	451,039,058	416,539,343	1.08	234,495,627	1.92
1,048,576	952,872,819	884,714,939	1.07	494,437,564	1.92

Table 3.9: Duration of “Merge-Sort” program in processor cycles (clk. freq. 100 MHz)

As it was expected the second multiprocessor algorithm has the best performance. The first multiprocessor algorithm has a very small speedup, due to the extra cost of the excessive communication. It is almost as fast as the uniprocessor algorithm. On the other hand, the second multiprocessor algorithm presents a very good and stable speedup throughout the experiments. Concerning this algorithm, the only part that interprocessor communication arises is at the last step of the algorithm, where the final result must be composed out of the two subtables. This corresponds to the minor penalty paid by the system.

Chapter 4

Conclusion and Future Work

Due to technology advances future high performance computer architectures will employ multiple processing cores on a single chip. This alters the whole setting of the uniprocessor system that stands true until now, and takes up the biggest part of the processors market. In order to acquire a greater understanding of future architectures an attempt to come closer to a real multiprocessor system is being done in this study. A multinode FPGA-based prototype has been designed and implemented. The system is equipped with two PowerPC cores, which are embedded in the Xilinx Virtex-II Pro FPGA. External caches equipped with a MESI cache coherence protocol are implemented. The two caches are connected by a custom interconnect, which has been given the properties of a bus, to the DDR controller. Through this path the two processors are able to access shared coherent memory. The opportunity of addressing private memory has also be given to the processors by sending requests over the PLB bus. Simulations carried out has shown that the system implemented is more efficient than any other composed exclusively by Xilinx soft-cores and offers the same type of hardware coherency. Additionally, the system offers lower latency accesses to data stored outside of the boundaries of the processors, when these accesses hit in the coherent caches. Furthermore, custom benchmarks have been written, simulating basic program behaviors found in parallel programs. Specifically, the first program describes the increment of a shared counter between the two processors. The second program implements a producer-consumer relationship, while the third uses the processors to sort an array of random integers. Software primitives, such as locks, have been implemented in order to achieve processor synchronization. The performance of the system measured by these programs was compared against the performance of an equivalent uniprocessor system. The comparison from the first program has revealed the negative impact of synchronizing the processors before accessing shared data. The shared program was found to be 3 to 10 times slower than the best uniprocessor one. The main reason for that constitutes the additional instructions that the shared program has to execute and the software primitives used for synchronization. Furthermore, the nature of the program creates a vast amount of bus traffic in order to synchronize the accesses. As far as the second program is concerned, the multiprocessor system is 2 to 3.5 times faster than the equivalent uniprocessor one. The main reason for that comes from the fact that the uniprocessor one has to pay the penalty of using threads on a single processor. Switching between threads, and wasted quantum of time due to lack of data to be consumed cause this extraordinary

speedup. Finally, the sorting program that implements the *merge-sort* algorithm, is run from 1.8 to 1.95 times faster in the multiprocessor system than in the uniprocessor. The speedup measured is considered to be the prospective one, since the specific algorithm can be well divided in two parts.

One of the future objectives of this work is to design and implement a coherent network interface that will be attached in the coherent network interconnect. By doing so, we will be able to build systems with more than two processors and also measure the effectiveness of attaching the network interface to the memory bus. Furthermore, a more efficient way of implementing locks must be designed. Software locks have turned out to be expensive. The PowerPC doesn"t dispose atomic instructions, however, atomic instructions can be generated by the coherent cache if they can be properly identified by the rest of the coherent memory system.

Bibliography

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal for Research and Development*, 49(4/5):589–604, 2005.

[2] IBM. Power4:http://www.research.ibm.com/power4.

[3] IBM. Power5:presentation at microprocessor forum. 2003.

[4] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the Design Space of Future CMPs. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, Washington, DC, USA, 2005. IEEE Computer Society.

[7] S. Mukherjee, B. Falsafi, M. Hill, and D. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of 23rd ACM Int. Symposium on Computer Architecture (ISCA 1996)*, pages 247–258, Philadelphia, PA USA, May 1996.

[8] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. *SIGARCH Comput. Archit. News*, 33(2):50–59, 2005.

[9] Xilinx Inc. *PowerPC 405 Processor Block Reference Guide*, August 2004.

[10] Xilinx Inc. *PowerPC Processor Reference Guide*, September 2003.

[11] Xilinx Inc. *Processor IP Reference Guide*, December 2004.

[12] J. L. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[13] C. Scheurich and M. Dubois. Correct Memory Operation of Cached-Based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA 1987)*, pages 234–243, June 1987.

[14] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993.

[15] David Culler, J.P. Singh, and with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishing Co., Menlo Park, CA, 1998.

[16] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-machine multicomputer: an architectural evaluation. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 224–235, New York, NY, USA, 1993. ACM Press.

[17] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 509–520, New York, NY, USA, 1998. ACM Press.

[18] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[19] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, New York, NY, USA, 2000. ACM Press.

[20] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Xilinx Inc. *Processor Local Bus (PLB) v3.4*, September 2004.

[22] Xilinx Inc. *PLB-to-IPIF Controller v2.01.a*, May 2004.

[23] Njuguna Njoroge, Sewook Wee, Jared Casper, Justin Burdick, Yuriy Teslyar, Christos Kozyrakis, and Kunle Olukotun. Building and Using the ATLAS Transactional Memory System. In *Workshop on Architecture Research using FPGA Platforms (WARFP'05) at HPCA-11*, February 2005.

Appendix A

DSPLB & PLB2Cache module

A.1 DSPLB Behavior

The data-side processor local bus (DSPLB) interface enables the PowerPC 405 data cache unit (DCU) to load and store data from any memory device connected to the processor local bus (PLB). This interface has a dedicated 32-bit address bus output, a dedicated 64-bit read-data bus input, and a dedicated 64-bit write-data bus output. The interface is designed to attach as a master to a 64-bit PLB, but it also supports attachment to a 32-bit PLB. It is capable of one data transfer (64 or 32 bits) every PLB cycle.

The same things hold for the instruction cache unit. It is also designed to connect as a master to a 64-bit or 32-bit PLB bus, and is capable of transferring one datum word every PLB cycle. Since PLB specification supports multiple masters, DSPLB and ISPLB can be connected to a single PLB bus. This approach is also followed in this study. Each processor is connected to one PLB bus to retrieve data and instructions. The arbiter of the bus is responsible to schedule the generated requests properly. In the case were both masters require to be granted the bus, the arbiter gives priority to the DSPLB interface. This generally results in better processor performance.

Data (read and write) requests are produced by the DCU and communicated over the PLB interface. A request occurs when an access misses in the data cache or the memory location that is accessed is non-cacheable. The signals used for the PowerPC 405 and the PLB to communicate can be seen in Figure A.1. Also, a short description of the signals can be found in Table A.1.

A.2 DSPLB Signal Summary

Depending on the type of the address range accessed (cacheable & non-cacheable) the PLB interface generates the appropriate request. Access to cacheable memory causes an entire cache line (8 words) to be transferred over the PLB, while non-cacheable accesses usually request only one word to be transferred. Cacheable data transferred as a cache line are eventually stored in the cache array, while single non-cacheable words remain in the fill buffer. There is also the

Signal	Function
C405PLBDCUREQUEST	Indicates the DCU is making a data-access request.
C405PLBDCURNW	Specifies whether the data-access request is a read or a write.
C405PLBDCUABUS[0:31]	Specifies the memory address of the data-access request.
C405PLBDCUSIZE2	Specifies a single word or eight-word transfer size.
C405PLBDCUCACHEABLE	Indicates the value of the cacheability storage attribute for the target address.
C405PLBDCUWRITETHRU	Indicates the value of the write-through storage attribute for the target address.
C405PLBDCUU0ATTR	Indicates the value of the user-defined storage attribute for the target address.
C405PLBDCUGUARDED	Indicates the value of the guarded storage attribute for the target address.
C405PLBDCUBE[0:7]	Specifies which bytes are transferred during singleword transfers.
C405PLBDCUPRIORITY[0:1]	Indicates the priority of the data-access request.
C405PLBDCUABORT	Indicates the DCU is aborting an unacknowledged data-access request.
C405PLBDCUWRDBUS[0:63]	The DCU write-data bus used to transfer data from the DCU to the PLB slave.
PLBC405DCUADDRACK	Indicates a PLB slave acknowledges the current data access request.
PLBC405DCUSSIZE1	Specifies the bus width (size) of the PLB slave that accepted the request.
PLBC405DCURDDACK	Indicates the DCU read-data bus contains valid data for transfer to the DCU.
PLBC405DCURDDBUS[0:63]	The DCU read-data bus used to transfer data from the PLB slave to the DCU.
PLBC405DCURDWDADDR[1:3]	Indicates which word or doubleword of an eightword line transfer is present on the DCU read-data bus.
PLBC405DCUWRDACK	Indicates the data on the DCU write-data bus is being accepted by the PLB slave.
PLBC405DCUBUSY	Indicates the PLB slave is busy performing an operation requested by the DCU.
PLBC405DCUERR	Indicates an error was detected by the PLB slave during the transfer of data to or from the DCU.

Table A.1: DSPLB PLB Interface Signal Summary

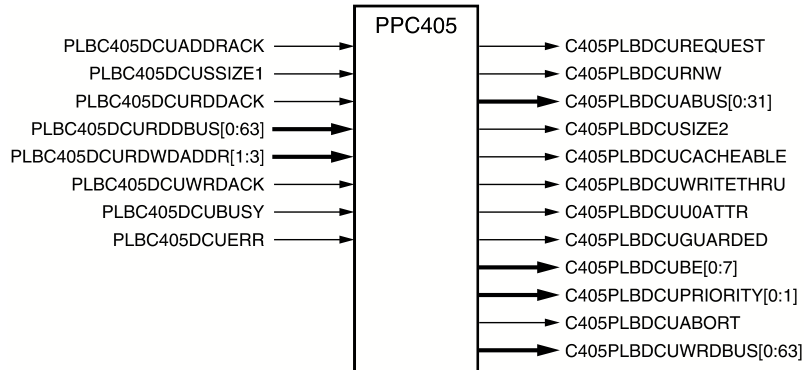


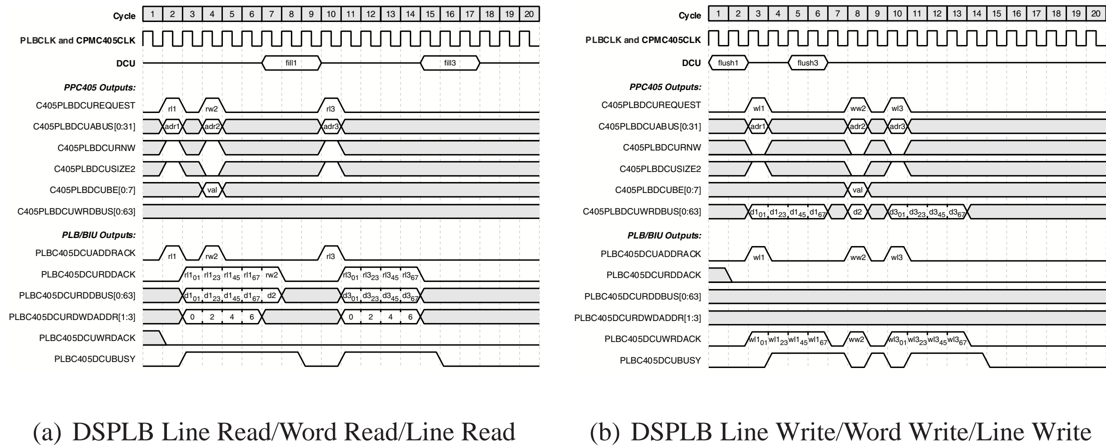
Figure A.1: Data-Side PLB Interface Block Symbol

possibility that non-cacheable reads be loaded using an eight-word line transfer, in order to take advantage of the PLB line-transfer protocol to minimize PLB-arbitration delays and bus delays associated with multiple, single-word transfers. The transferred data are placed in the DCU fill buffer, but not in the data cache. Subsequent data reads from the same non-cacheable line are read from the fill buffer instead of requiring a separate arbitration and transfer sequence across the PLB. Data in the fill buffer are read with the same performance as a cache hit. The non-cacheable line remains in the fill buffer until the fill buffer is needed by another line transfer.

From these three scenarios described above, the first and the third must be prohibited when accessing memory characterized as shared. In these two cases, subsequent memory accesses to the same addresses are being served by the internal cache structure and not from the coherent cache. This threatens memory coherency because these subsequent accesses are performed without taking into account that the specific memory is probably used by another entity, too. The only way to solve this problem is by making each processor to generate accesses that are not influenced by the internal state of the data cache in order to access shared memory. This is accomplished by addressing non-cacheable memory, transferring a single word per memory access. The rest of the private memory can be safely set as cacheable.

A PLB access can be divided into two parts. The first part has to do with the address accessed and the second part with the data returned (read access) or given (write access). Figure A.2 depicts different kinds of PLB accesses.

During the first part, the processor requests to be granted the bus, setting the C405PLBDCUREQUEST high and driving the address signal (C405PLBDCUABUS). Also control signals like C405PLBDCURNW and C405PLBDCUSIZE2 must be valid and stable until the bus acknowledge the address. Depending on the state of the bus and the state of the slave peripheral, the address can be acknowledged in the same cycle that the request is asserted. After this acknowledgement, the request signal is driven low. In case of read access, when the datum (single word access) or data (burst access) become available the slave peripheral will drive PLBC405DCURDDACK high. The number of cycles that the acknowledge signal will remain high depends on the access (1 cycle for single word access, multiple cycles for burst access). In case of a write access, the first 64-bit data word must be valid in the cycle the C405PLBDCUREQUEST is driven high. The slave peripheral can acknowledge the data even



(a) DSPLB Line Read/Word Read/Line Read (b) DSPLB Line Write/Word Write/Line Write

Figure A.2: PLB Accesses

in this cycle, by driving the PLBC405DCUWRDACK signal high. If the access writes only a single word then the access finishes in the same cycle, and PLBC405DCUWRDACK is driven low. A dummy cycle of no activity must pass before the DSPLB sends the next request. If the access writes more than one word, PLBC405DCUWRDACK remains high until all the words are acknowledged.

There are some issues that must be discussed about the accesses shown in the last figure. First of all someone can observe subsequent accesses to overlap in time. This is because the DCU can overlap an on-going request with a previous one. This process, known as address pipelining, enables a second address to be presented to a PLB slave while the slave is transferring data associated with the first address. Address pipelining can occur if a data-access request is produced before all data from a previous request are transferred by the slave. This capability maximizes PLB-transfer throughput by reducing dead cycles between multiple requests. The DCU can pipeline up to two read requests and one write request. (Multiple write requests cannot be pipelined.) A pipelined request is communicated over the PLB two or more cycles after the prior request is acknowledged by the PLB slave.

Furthermore, there have been made some timing assumptions regarding the timing diagrams shown.

- For example, requests are acknowledged in the same cycle they are presented by the DCU, if the bus interface unit (BIU) is not busy. This doesn't hold if the DCU is connected to a PLB bus. The PLB requires 3 cycles of arbitration before the peripheral becomes aware of the request.

- The first read-data acknowledgement for a data read is asserted in the cycle immediately following the read-request acknowledgement. This represents the earliest cycle a bus interface unit (BIU) can begin transferring data to the DCU in response to a read request. However, the earliest the PLB begins transferring data is two cycles after the read request is acknowledged.

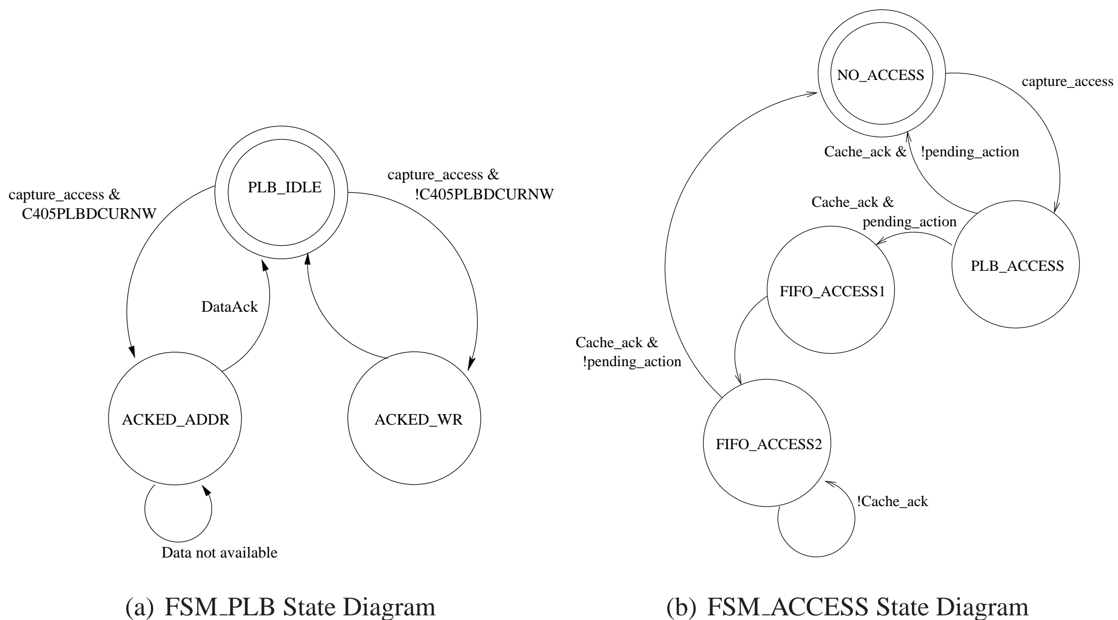
- The first write-data acknowledgement for a data write is asserted in the same cycle as the write-request acknowledgement. This represents the earliest cycle a BIU can begin

accepting data from the DCU in response to a write request. However, the earliest the FPGA PLB begins accepting data is two cycles after the write request is acknowledged.

The timing diagrams of Figure A.2 show the fastest way the DCU can operate having been connected to an ideal bus or directly (point-to-point) to an ideal peripheral. Constraints and limitations must be taken into account in order to make a safe estimate of the performance of a system that disposes a PLB bus and more than one PLB peripherals.

A.3 PLB2Cache module FSMs

Figure A.3 shows the state diagrams of the two FSMs. As it can be seen, both of them are very simple and have very few states. The FSM_PLB FSM has three states. The idle state is called PLB_IDLE. It remains in this state until a valid request for shared memory is generated. Depending on the type of the request it proceeds to state ACKED_ADDR, in case of a read request, otherwise to state ACKED_WR. When performing this transition the address of the request is acknowledged. Additionally, the address is pushed into the FIFO if the FSM_ACCESS is busy serving a previous request. The FSM remains in the ACKED_ADDR state until the requested data arrive and the PLB has finished serving any previous requests. Then it returns to PLB_IDLE. When in state ACKED_WR it stays there for a cycle and then returns to the idle state. The purpose of waiting for one cycle has to do with the constraint the DCU imposes of spending one cycle without activity after each completed write request. During this cycle the C405PLBDCUREQUEST signal is low and the PLBC405DCUBUSY high.



(a) FSM_PLB State Diagram (b) FSM_ACCESS State Diagram

Figure A.3: PLB2Cache module FSMs

FSM_ACCESS is also very simple, consisting of only four states. The idle state is called NO_ACCESS. It remains in this state until an access to shared memory is captured. When this happens, the access is instantly passed to the coherent cache, and the FSM proceeds to state

PLB_ACCESS. It stays in that state until the request that it serves has completed. Depending on the occupancy of the FIFO, the FSM_ACCESS transitions either to FIFO_ACCESS1 in order to serve requests that have been queued in the FIFO or back to the idle state, if there are no available requests. In FIFO_ACCESS1 state stays for one cycle, in order to pop the address of the access from the head of the queue, and then proceeds to state FIFO_ACCESS2. State FIFO_ACCESS2 is equivalent to state PLB_ACCESS with the difference that data are read by the FIFO and not by the FSM_PLB. It remains in that state until the request is completed. When this happens, it transitions back to FIFO_ACCESS1, if there are available requests in the FIFO or back to the idle state.

FSM_ACCESS is responsible for communicating with the coherent cache, which is the next step in the hierarchy. Each couple of states (NO_ACCESS PLB_ACCESS and FIFO_ACCESS1 - FIFO_ACCESS2) corresponds to serving one request. The minimum this can hold is 2 cycles, which corresponds to cache hit and completion of the request. In the first state of the couple the address of the request is sent to the cache in order to read the tag memory. In the second state a decision on a hit or a miss is taken. Until the cache signals the end of the request being served control, address and data signals towards the cache must be valid and stable.

A.4 Returning Data

An issue that must be cleared has to do with the way the DCU receives data when load requests are issued. As mentioned above, the DCU supports two reads and one write accesses to wait for completion at the same time. This comes from the ability of the DCU to pipeline requests. There are two entities that serve DCU requests, the PLB bus that responds to requests accessing private memory and the coherent memory system that responds to requests accessing shared memory. The problem arises, when the DCU overlaps subsequent load requests that access shared and private memory. These two requests will be served in parallel by two different entities. Thus, no assumptions can be made about the time required for each request to complete. As a result, there is the likelihood that these two requests would not complete in program order. In systems, where the DCU is connected directly to the PLB bus without any other entity standing in between, this problem was solved because the PLB bus and the slave peripherals attached to it were serving requests in a first-come-first-served fashion. Now, care must be taken in order to impose this kind of completion on the load requests. In order to achieve this, all the possible combinations of load requests must be taken into account. Since there must be a load request to shared address space, this request must be non-cacheable (with respect to the DCU). Thus, the possible combinations are the following:

1. burst access to private memory followed by access to shared memory
2. non-cacheable access to private memory followed by access to shared memory
3. access to shared memory followed by burst access to private memory
4. access to shared memory followed by non-cacheable access to private memory

Figure A.4 provides an example of the fastest speed at which the PowerPC 405 DCU can request and receive single words over the PLB. The DCU is designed to wait for the current single-word

read request to be satisfied before making a subsequent read request. This requirement results in the delay between requests shown in the figure.

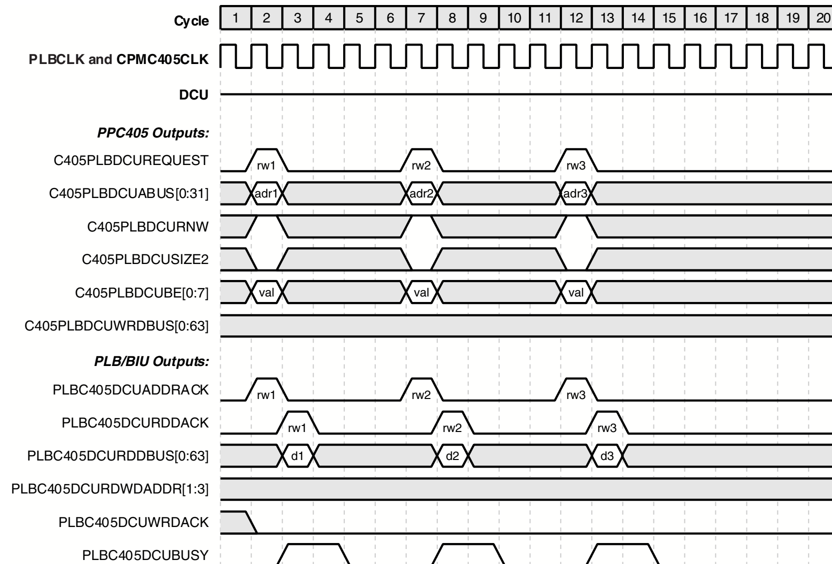


Figure A.4: DSPLB Three Consecutive Word Reads

Thus, combinations 2, 3 and 4 can be considered safe, since there is no DCU activity until the first request of the couple to complete (Figure A.2 doesn't show an example for combination 2, but same thing happens in that case, too). The only combination that must be taken under consideration is combination 1. Enforcing program order on load request completion for combination a is rather simple. When PLB2Cache module is about to return data to the DCU, it checks signals PLBC405DCUBUSY and PLBC405DCURDDACK. If both of them are high, this means that a burst read access has preceded. The PLB2Cache module waits for PLBC405DCURDDACK to go low before it returns the data.

Appendix B

Details about the Coherent Cache

B.1 Part A FSMs

B.1.1 FSM_CPU_ACCESS

Figure B.1 shows the state diagram for FSM_CPU_ACCESS. Six states constitute the FSM. CPU_READ_TAGS is the idle state. The PLB2Cache module provides the address and drives the control signals to initiate the access. Provided that part A is not busy evicting another block, the access can proceed, otherwise the FSM is blocked in the initial state. When part A becomes available, there are 3 possible states to which FSM_CPU_ACCESS may proceed. If the access is a non-cacheable read it transits to NC_RD state. It stays there until the data become available and then returns to the idle state. If the access is a non-cacheable write then it transits to NC_WR state. It stays there until the bus acknowledges the transmission of the data to the external memory. Then it returns to the initial state. Finally, if the access is a cachable (read or write) access it transits to CHECK_EQUALITY. The transition from the idle state to CHECK_EQUALITY incurs a read access to each of the tag memories. During the CHECK_EQUALITY cycle the tags are compared for equality with the tag-part of the address accessed. In case equality is found and the existence of the proper privileges for accessing the specific cache block, the access is considered to be a cache hit. It completes by the end of the second cycle. At the same time, data are returned to the processor, in case of a read access, or data are written in the cache, in cache of a write access. Additionally, tag memory is updated, in case of a write, and also LRU information about the accessed set are stored. In the next cycle, the FSM returns to the idle state.

Apart from the "cache-hit" scenario, there is also the possibility of the absence of the required cache block or the absence of privileges to use it. In the former case, the FSM transits to the MISS_BLOCK state, issuing the appropriate BusRd or BusRdX message (depending on the type of the access, read or write). It stays in that state until all the requirements are met for the access to complete. When that happens the Cache_Ack signal goes high for one cycle and the FSM transits back to the idle state. In the latter case the FSM transits to the MISS_COHERENCE state, issuing an Invalidate message. The only possibility of not having the proper privileges to

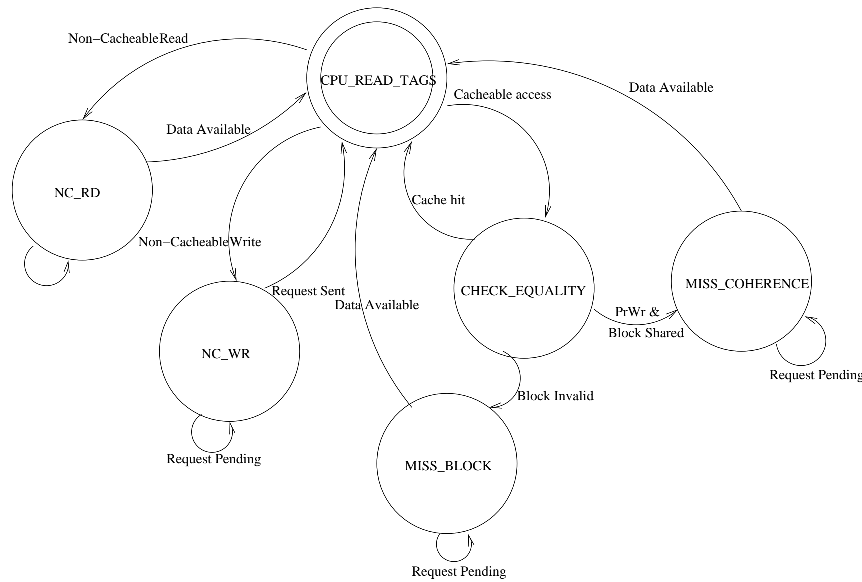


Figure B.1: FSM_CPU_ACCESS State Diagram

access a cache block is when a write access is issued for a block that is shared. All the other write-cases fall in the case where the required block is missing. The FSM stays in that state until all the requirements are met for the access to complete. When that happens the Cache_Ack signal goes high for one cycle and the FSM transits back to the idle state.

B.1.2 WB_FSM

The second FSM, WB_FSM as shown in Figure B.2, in part A is responsible for handling write back activity. Write back activity corresponds to transferring modified blocks back to main memory. This transfer takes place when an access that misses in the cache requires a new cache block to be loaded in. The new block that comes in conflicts with two blocks that are already present in the cache. These two blocks occupy the whole specific set, in which the new block is mapped. The replacement algorithm is called to resolve this conflict by choosing a block to be evicted from the cache. The block that is least recently used is chosen to leave the cache. If that block is 'clean' then the incoming block just over-writes it. If, however, the chosen block has been modified, it must be written back to main memory.

The logic identifies the need to evict a cache line, in order to make some space for the incoming block, during the second cycle of the access. At that time FSM_CPU_ACCESS is in the state CHECK_EQUALITY and is going transit to MISS_BLOCK, since the requested block is not present in cache. The 'replace_dirty' signal rises to notify the start of the eviction process and unblock the WB_FSM to transit from WB_IDLE to WB_REQ. In the positive clock edge between these two states, FSM_CPU_ACCESS generates a bus message to request the corresponding block.

WB_FSM cannot take advantage of this clock edge, since the path towards the bus is busy receiving the first of the two bus messages. Giving priority to loads over stores, that is sending first the bus message that requests the missing block, is well known, and it's proven to increase

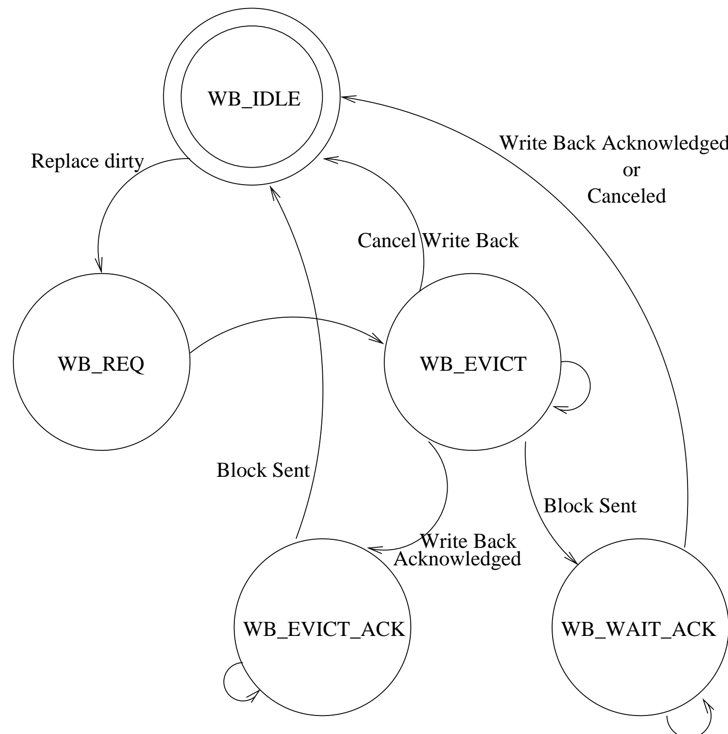


Figure B.2: WB_FSM State Diagram

the performance of the cache by reducing the miss penalty. Following this approach, WB_FSM waits for one cycle, moving to state WB_REQ. In the next positive clock edge it sends the request to notify the initiation of a write-back action and then moves to WB_EVICT state. For each cycle WB_FSM remains in the WB_EVICT state it sends a word that resides in the evicting block. The word is sent to the logic within part B that is responsible for the communication with the bus. The FSM leaves this state if all the words of the block have been sent, and transits to WB_WAIT_ACK. It can also leave from this state if a write-back acknowledgement is received. In that case it transits to WB_EVICT_ACK. The purpose of acknowledging the write-back transmission is to assure that the action can safely proceed. There are two possible scenarios when a write-back should not complete. In the first one, a remote cache invalidates the copy of the cache block that is about to be evicted. The remote cache intends to write the specific block, which also means that it undertakes the responsibility of holding the most up-to-date state of the block. Thus, the local cache doesn't need to write back the block to the main memory. If the local cache wrote back the block then coherency of the data would not be harmed. However, a useless request would have been issued, wasting DDR bandwidth. The second scenario involves receiving an update message for the block that is about to be evicted. In this case the write back action must be canceled. An update message is designed to clear the dirty bit if the block is found to be in a cache. Upon completion of the update message, all the caches and the main memory have an up-to-date copy of the block. Thus, no-one has the responsibility to flush it in case of a conflict, since the block isn't dirty in any cache. However, processor's and bus' accesses happen in an asynchronous way. Part A may have already decided to evict a block that in the near future will be updated. This conflict is recognized and the eviction process is canceled, notifying the bus

to also ignore any data that may have transferred. When WB_FSM is found in WB_WAIT_ACK it waits for a write-back acknowledge or write-back invalidation and then transits to WB_IDLE state. When in WB_EVICT_ACK the FSM is sure that the eviction will not fail. The rest of the words within the block are sent to part B and then WB_FSL transits to WB_IDLE.

A write-back transfer is not considered to be a separate bus transaction. It is hidden behind the block transfer that generated the eviction. That is feasible because the implementation of the bus offers different sub-buses for transferring data from and to the cache. Immediately after the bus request has been transferred, the write-back action is initiated. The address of the evicted block is first transferred and then the data in a critical-word-first fashion. The first word out is either the one that has the same offset with the address generated the miss, if the offset is even, or the previous word if the offset is odd. Either way, the first word out must lie in even offset due to constraints imposed by the implementation of the bus. It is crucial for the consistency of the memory the data to be transferred in this fashion. When the access that missed in the local cache is served by the remote cache, the incoming block arrives in the local cache before the end of the eviction process. If the process has started by writing back offset 0 to offset 7 then there is the possibility some data to be overwritten by the incoming block.

B.2 Completing Requests and Cache Status

The sequence of actions taken to resolve the conflict between a processor's access and an update bus request are similar to those needed to implement the critical-word-first feature. The dependency check module is used in this case, too. A processor's access is blocked waiting for the requested block to arrive. When that happens part B updates immediately the tag memory, but also forwards the proper information to the dependency check module. In the following cycle, part A reads the updated tag line, obtaining the belief that the requested block is available. However, the dependency check module identifies that both parts of the cache access the same cache block, driving the EarlyRestartMiss signal high. The FSM_CPU_ACCESS ignores the updated tag line and waits for the corresponding bit of the signal ValidBits to be driven high. This happens by part B when the specific word is arrived by the bus and is written in the cache.

Including the scenario just described, there are six possible ways for an access to be completed. Some of them have been mentioned earlier, but it would be more clear for the reader to be presented all together.

- Cache hit: An access may hit in the cache and complete in to cycles time. During the last cycle that acknowledge signal is driven high. If the processor access is a read access then the requested data are read directly by the data memory of the cache.

- In case of a read access which misses in the cache due to absence of the requested block, the FSM blocks in the MISS_BLOCK state. The dependency check module will signal the arrival of the data, as described above. However the data are not read by the data memory of the cache. When part B receives the word of the target address (regardless of the type of the request, read or write), it writes it to the data memory and also to the ReturnData register. Part A gets the requested data from this register and forwards them to the processor.

- In case of a write access which misses in the cache because of the absence of the requested block, the FSM blocks in the MISS_BLOCK state. The dependency check module will signal the arrival of the block and specifically the arrival of the target address. When that happens the request is considered to have been served. However, the data of the write access are not written to data memory by part A. While part A is blocked waiting the cache block to arrive, the data are forwarded to part B. When the word of the target address arrives it is ignored. Its place is taken by the new data provided by the processor. This is done because part A, although blocked, it might be busy evicting a cache block. In this case the data memory is busy and cannot handle the write access.

- In case of a write access which misses in the cache because of absence of privileges on the specific cache block, the FSM is blocked in the MISS_COHERENCE state. While blocked, it constantly reads the tag memories (without luck). When the cache eventually acquires privileges on the specific cache block part B updates the tag memory. The next read from part A on the tag-line of the requested block will notify the existence of the proper privileges for the access to complete.

- In case of a non-cacheable write, part A forwards to part B the address and the data to be written. The access completes when part A gets notified that the request has been sent over the bus.

- In case of a non-cacheable read, part A is blocked in NC_RD state. Part B signals the availability of the non-cacheable word by placing it in ReturnData register and also setting Non_Cacheable_Data_Valid register high. This triggers the completion of the access. Data are read by the ReturnData register and are forward to the processor.

B.3 Part B FSMs

B.3.1 BUS_FSM

Figure B.3 depicts the state diagram of the FSM. BUS_IDLE is the initial state. BUS_FSM remains in that state until a new request is forwarded by part A. When this happens it transits to either BUS_CHECK_CMD or BUS_SEND_CMD. BUS_CHECK_CMD is chosen when there must be a comparison between the address of the request to be sent and the address of a request that has just been arrived. The reason to do so is that there may be the need to change the type of the outgoing request. This need comes from the fact that an Invalidate request doesn't cause the transfer of a block. The scenario that threatens the coherency of the data is the case where the local processor sends an Invalidate message for a certain block, which is shared and wishes to write to it. At the same time a message broadcasted on the bus invalidates (Invalidate or BusRdX) the specific block. The message from the remote cache has come before the local message, and thus the remote processor will use the up-to-date copy of the block. If the local Invalidate message is not changed to BusRdX message then the local processor will operate upon the old copy of the block.

If a scenario like this doesn't occur, then the FSM will transit from BUS_IDLE to BUS_SEND_CMD. However, there is also the possibility of changing the outgoing request while

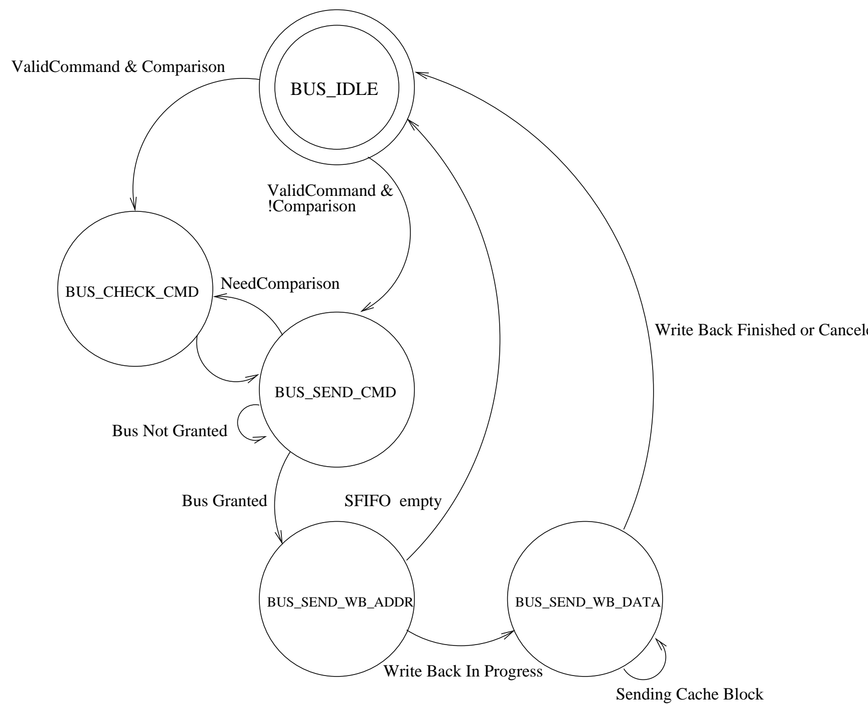


Figure B.3: BUS_FSM State Diagram

in BUS_SEND_CMD, too. This may happen only if there are three participants on the bus, the local cache is granted the bus last and the previous request served conflicts with the local request. In this case, the FSM transits to BUS_CHECK_CMD for one cycle, and then returns to BUS_SEND_CMD. While in BUS_SEND_CMD the FSM requests to be granted the bus. It stays there until it receives an acknowledge signal, which means that the request has been broadcasted. Then it transits to BUS_SEND_WB_ADDR. The cycle dedicated to this transition corresponds to sending the address of the block that is about to be written back or the data for a non-cacheable write. The FSM then transits to BUS_SEND_WB_DATA, if there is a cache block to be evicted or to BUS_IDLE in all other cases. The FSM stays in BUS_SEND_WB_DATA state until all the words of the cache block are transmitted or the decision to cancel the write back is taken. Then it returns to BUS_IDLE. A small improvement in the request procedure is that in the case where there is no need to transit from BUS_IDLE to BUS_CHECK_CMD, then the output signal B_Req is set high before this transition. The bus registers the request signals, and thus a cycle is saved in the common case.

Finally, this part of the logic is responsible to notify when an Invalidate message can be safely considered to have completed. This is done when the FSM is in the BUS_SEND_CMD state trying to send an Invalidate message and receives an acknowledge signal from the bus. Having taken control over the shared medium can safely assume that the Invalidate message will be translated by anyone on the bus in the same manner (order). Thus, it is safe to assume its completion. At the next cycle the privileges of the block in this cache are upgraded.

B.3.2 FSM_REQ_IN

Figure B.4 shows the state diagram of FSM_REQ_IN. BUS_RD_TAGS represents the idle state of the FSM. The arrival of a request drives the address pins of the tag memories, and also makes the FSM to transits to either BUS_CHECK_EQ or FETCH or NC_READ. If the request is one of the BusRd, BusRdX, Invalidate and Update, the FSM goes to BUS_CHECK_EQ and uses the output of the tag memories to check if the requested block is cached. If the request comes from an incoming refill message or an incoming non-cacheable read then the FSM transits to FETCH state or NC_READ state, respectively, and ignores the output of the tag memories. While in the FETCH state the incoming block is written in the data memory and the tag information gets updated. The address that had come before the block declares the sequence on which the block will be written in the memory. The word corresponding to the target address is also written to the ReturnData register and forwarded to part A. As far as the non-cacheable read requests are concerned, the FSM transits to NC_READ, signaling that the requested word has become available. The transition is triggered by the start of the response message. The non-cacheable word follows back to back the start of the message, which is the address of the datum. In this case the address has no use, but also doesn't cost anything. The implementation of the bus is aware of this detail and makes sure to send the address one cycle before it becomes able to send the datum.

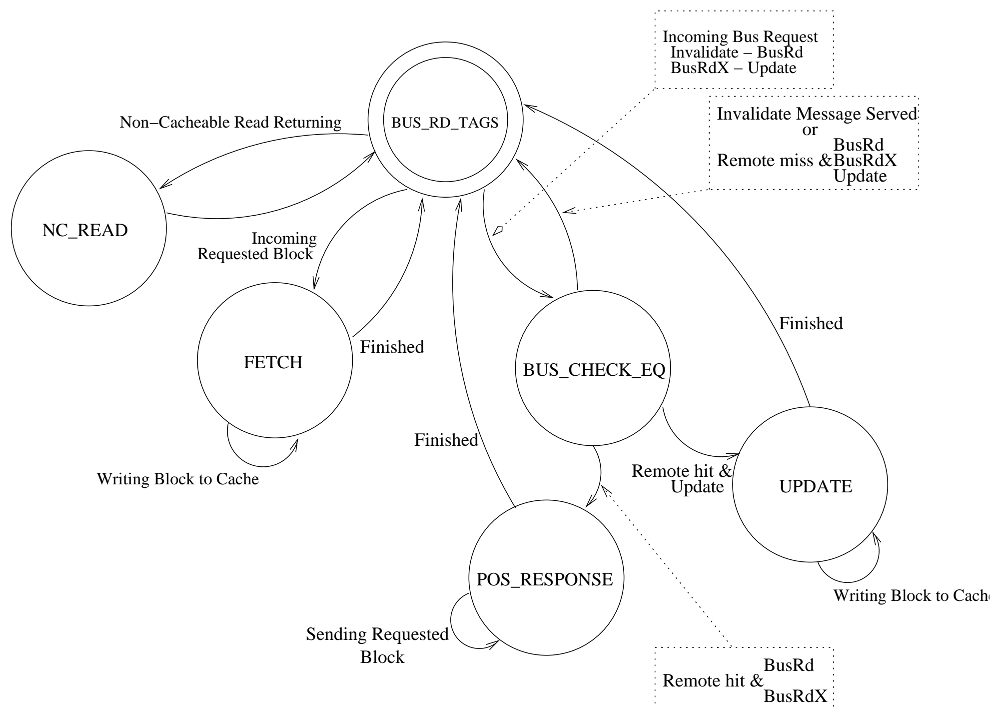


Figure B.4: FSM_REQ_IN State Diagram

While in BUS_CHECK_EQ it compares the tag-part of the incoming address with the outputs of the tag memories. If equality occurs then it is said that the request had a remote hit (in the remote cache). Depending on the type of the request a response, either positive or negative, must

be sent back. Both BusRd and BusRdX are required to notify the bus about the status of the request in the specific cache. The bus then should know if it should forward the request to the DDR controller or data from another cache will be sent to the initiator cache, instead. On the other hand, Update and Invalidate messages don't cause any additional flow of information to the bus. If the requested cache block is found in the cache it gets either updated or invalidated. If the requested cache block isn't cached the message is just ignored. Any of the following messages, for which the remote cache doesn't hold a copy of the requested address, make the FSM to transit back to the idle state. During the transition the Request_miss signal is driven high, in order the bus to be notified for the miss. On the other hand, if a BusRd or a BusRdX message hits in the remote cache the FSM transits to POSITIVE_RESPONSE and responds with the shared data. An Invalidate message that hits in the remote cache doesn't need more processing. It transits back to the idle state. During the transition it modifies the tag information concerning the requested block. Finally, an Update message that hits in the cache moves to UPDATE state.

While in POSITIVE_RESPONSE data from the cache are sent to the bus. During the transition from BUS_CHECK_EQ to POSITIVE_RESPONSE the cache has already sent the address (or the head of the response message) of the cache block. The data of the block follow back to back. After the transmission of the whole block the FSM returns to the idle state. While in the UPDATE state, data coming in from the bus are placed into the data memory. After having received the entire cache block the FSM returns to the idle state.

B.4 Communication with the dependency check module

The behavior of the dependency check module and the communication between the two parts of the cache has already been described. However, a small description follows regarding the specific timing FSM_REQ_IN uses in order to serve the incoming requests. BusRd and BusRdX messages that hit in the cache modify the tag memory during the first of the eight cycles of the POSITIVE_RESPONSE state. Thus, in order the dependency check module to receive in time the required information, FSM_REQ_IN forwards it when transiting from BUS_CHECK_EQ to POSITIVE_RESPONSE. The same holds for successive Update messages. The eight valid bits, though, are updated cycle by cycle. An invalidate message that hits in the cache clears the tag line as soon as the hit is resolved. This is done at the end of the BUS_CHECK_EQ cycle. Thus, proper information is being forwarded as soon as the message is received. A refill message updates the tag-line upon reception (at the end of the BUS_RD_TAGS before transition to FETCH). At the same time the proper information is forwarded to part A and also the valid bits are updated cycle by cycle, while FSM_REQ_IN is in FETCH state.

Appendix C

Coherent Bus Module FSMs

C.1 FSM_Arb

Figure C.1 depicts the state diagram of the FSM_Arb that handles the transmission of requests and data through the bus. ARBITRATE corresponds to the idle state. The FSM remains in this state until it receives a new request. The request signals that are sent by the participants are masked before they reach the FSM's logic. The FSM recognizes a new request only when it is capable of serving it. The incapability of the FSM to serve a request comes from FIFO's finite capacity. The amount of memory dedicated to the FIFOs is statically partitioned between the different types of messages (one different FIFO module for each type). Consequently, there is totally enough memory to keep pending 4 non-cacheable write accesses, 2 block evictions, and 2 Update messages. Whichever part of the memory is full the bus stops accepting requests until a free position be created in that part. The FIFO-memory was chosen to be partitioned statically in order to separate the memory dedicated to storing evicted blocks, from all the other kinds. The need to do so arises from the fact that a write back can be canceled. A part of the block that is supposed to be written back may reside in the FIFO's when the action is canceled. In order to clear these words the FIFO module is being reset. No other kind of data must be present during this FIFO reset.

When, eventually one or more requests can be served by the bus, the FSM decides which one to serve (arbitration) sending an acknowledgement to the corresponding cache, and transits to the next state. A BusRd or BusRdX message produces the transition towards the GATHER_REPLIES state, while a non-cacheable write the transition towards the NON_CACHEABLE_WR. An Update message produces the transition towards the UPDATE state, and all the other requests the transition to the NULL state. In parallel with transiting to one of these states, the head of the message, which is the address, is either broadcasted or written to the corresponding FIFO. A non-cacheable write pushes the address to be accessed in the commands FIFO during the transition from the idle state to NON_CACHEABLE_WR. While in NON_CACHEABLE_WR it pushes the data to be written into the corresponding FIFO (Commands_fifo) and returns to idle state. An Invalidate message follows a similar course. While in the idle state the address and the type of the request are being broadcasted to all remote caches.

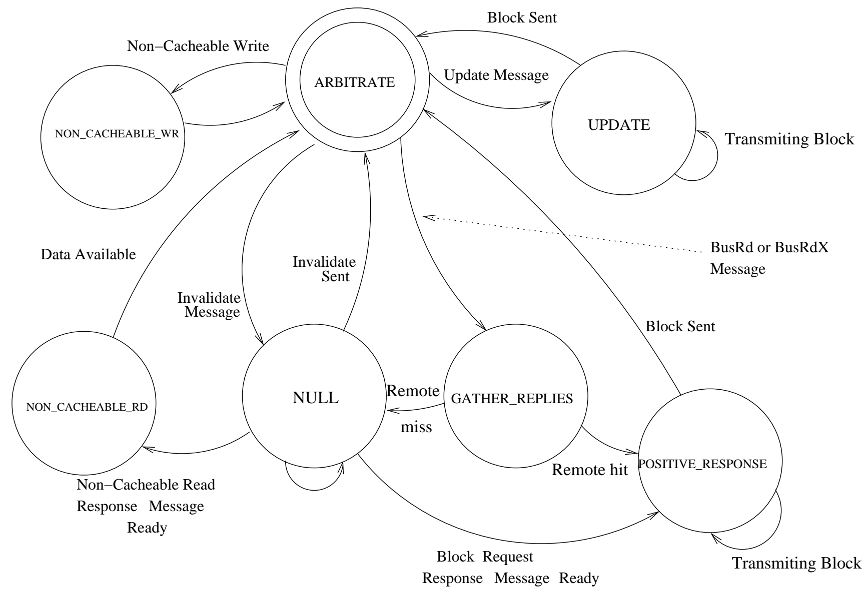


Figure C.1: FSM_Arb State Diagram

The FSM transits to NULL state, where it stays there for one cycle only. Then it returns to the idle state to serve the next request. The existence of that dummy cycle is necessary, because during this period the remote caches are evaluating the equality check between the incoming address and the accessed tag-line. If a hit occurs then Part B of each cache that resolved equality, is busy updating the tag memory and cannot accept another request. Update messages follow a straightforward course, too. While in the idle state, the address of the block to be updated is broadcasted, along with the type of the message. Furthermore, the update command and the address are pushed into the Commands_fifo, in order the request to reach the main memory. The FSM transits to UPDATE state and stays there until all the words of the block are broadcasted to the participants. In parallel with this, each word is also written to the Update_fifo. The FSM returns to the idle state after having sent the entire block. The actions taken for BusRd and BusRdX messages are more complex. During the transition from the idle state to GATHER_REPLIES state the address and the type of the request are being broadcasted. While in GATHER_REPLIES the FSM waits for either one or more positive replies, or a number of negative replies, which are enough to identify that the request failed in the remote caches and should be served by the main memory. Assertion of signal B_CoherenceData__x notifies a positive response from cache ’x’, while assertion of signal B_Request_miss__x notifies a negative response. If one or more positive responses are found the FSM transits to POSITIVE_RESPONSE state, and stays there until the whole cache block is transferred. Any cache that has a copy of the requested block will try to transmit the cache block. Only one of them will be chosen to make the transmission, and this will be the first cache, in relation to the one being served, in the round-robin order. Depending on the type of the request, the block is loaded either in shared or exclusive state. In case of a BusRd message, signal B_SharedState__x is driven low if the requested block is not shared, otherwise it is driven high. In case of a BusRdX message, signal B_SharedState__x is always driven low in order the block to be loaded to Exclusive state. The signal is valid for one cycle only, when the head of the message arrives at the cache that initiated the request. After the transmission of the

whole block the FSM returns to the idle state. As far as the negative responses are concerned, the FSM transits to NULL state when it resolves a total remote miss. During the transition, the address and the corresponding command are written in the Commands_fifo. The DDR controller becomes aware of the request and some cycles latter the cache block is available in the DataIn_fifo. The corresponding address is sent back to the cache, as the head of the message that caries the requested block. The B_SharedState__x signal is driven to the proper value and the FSM transits to POSITIVE_RESPONSE state to send the data. After the whole cache block is sent it returns to the idle state.

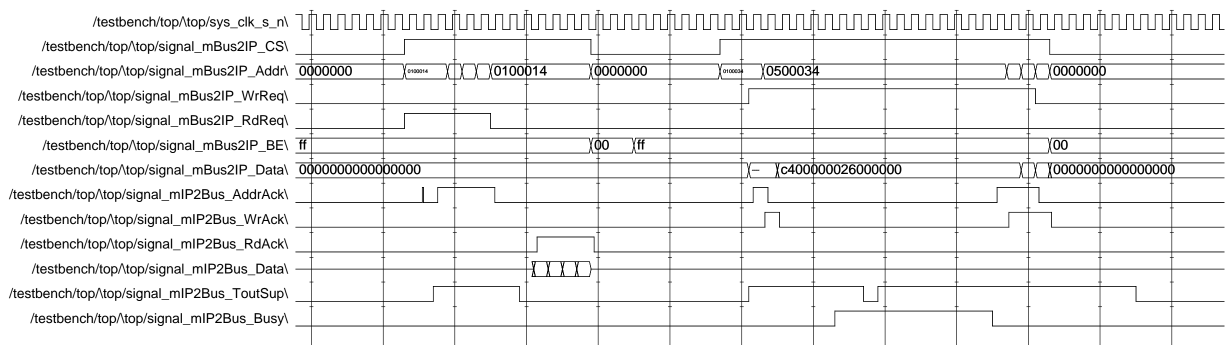
C.2 FSM_BUS_WB

In the same clock domain with the FSM_Arb there is also a less complex FSM, named FSM_BUS_WB that handles the write back activity. BUS_WB_IDLE is the idle state, and the FSM stays in that state until the initiation of a block eviction is signaled. If so the FSM transits to ACCUMULATE_DATA, where it receives the data sent by the cache and push them into one of the two available write back FIFOs. If the write back is canceled the FSM returns to the idle state, otherwise when half the block has been transferred to a FIFO the FSM transits to WB_CONFIRMED. Having transferred half of the block it is then certainly that the eviction process has no reason to be canceled. While in the WB_CONFIRMED state the rest of the block is received and written to the same FIFO. After the end of the process the FSM returns to the idle state. The crucial part of the process is the time chosen to notify the DDR controller that there is a write back to be carried out. This is done only after confirmation of the eviction process. Specifically, while transiting to WB_CONFIRMED state, the address of the block and the corresponding command are written in the Commands_fifo. The DDR controller will serve the write back request after having finished with the BusRd or BusRdX request that came before.

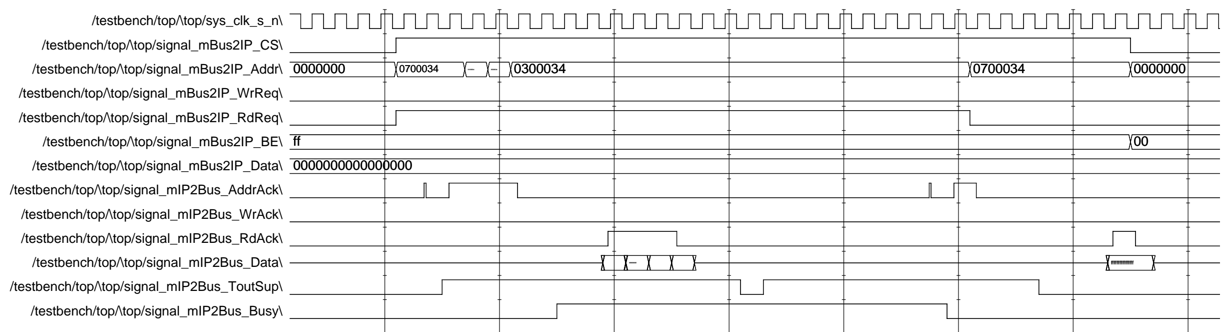
Appendix D

Detailed Reports

D.1 Waveforms for Halted IPIF Burst Accesses



(a) IPIF Burst Write Accesses Halted



(b) IPIF Burst Read Accesses Halted

Figure D.1: IPIF Burst Accesses Halted

Source:	top/MemorySystem/cache_0/tags_mem/tagsmem0/B6.A (MEM)	
Destination:	top/ppc405_0/ppc405_0/PPC405_i (CPU)	
Data Path Delay:	9.820ns (Levels of Logic = 8)	
Clock Path Skew:	0.000ns	
Source Clock:	top/plb_bram_0_connector_BRAM_Clk rising	
Destination Clock:	top/plb_bram_0_connector_BRAM_Clk rising	
Clock Uncertainty:	0.000ns	
Delay type	Delay(ns)	Logical Resource(s)
Tbcko	1.401	top/MemorySystem/cache_0/tags_mem/tagsmem0/B6.A
net (fanout=4)	0.932	top/MemorySystem/cache_0/A_DataOut0_tags¡17¿
Tilo	0.274	top/MemorySystem/cache_0/comp21A/Eq161
net (fanout=1)	0.551	top/MemorySystem/cache_0/comp21A/Eq161/O
Tilo	0.254	top/MemorySystem/cache_0/comp21A/Eq238
net (fanout=4)	0.575	CHOICE6949
Tilo	0.274	top/MemorySystem/cache_0/Ker753521
net (fanout=3)	0.585	top/MemorySystem/cache_0/Lru_bits_associate_in
Tilo	0.254	top/MemorySystem/cache_0/Cache_hit33
net (fanout=4)	0.311	top/MemorySystem/cache_0/CHOICE1663
Tilo	0.254	top/MemorySystem/cache_0/Cache_hit54
net (fanout=11)	0.272	top/MemorySystem/cache_0/Cache_hit
Tilo	0.274	top/MemorySystem/cache_0/Cache_ack25_2
net (fanout=33)	0.650	top/MemorySystem/cache_0/Cache_ack25_2
Tilo	0.254	top/vag_0/Ker668951
net (fanout=64)	0.902	top/vag_0/N66897
Tilo	0.254	top/vag_0/PLBC405DCURDDBUS_OUT¡51¿1
net (fanout=1)	1.516	top/PLBC405DCURDDBUS_OUT_0¡51¿
Tpdck_PLB	0.033	top/ppc405_0/ppc405_0/PPC405_i
Total	9.820ns	(3.526ns logic, 6.294ns route) (35.9% logic, 64.1% route)

Table D.1: Time Consumption at the Read Hit Path

D.2 Detailed Timing Reports for Critical Paths

Source:	top/MemorySystem/cache_0/B_Non_Cacheable_Data_Valid (FF)	
Destination:	top/ppc405_0/ppc405_0/PPC405_i (CPU)	
Requirement:	5.000ns	
Data Path Delay:	4.970ns (Levels of Logic = 4)	
Clock Path Skew:	0.000ns	
Source Clock:	top/sys_clk_s_n rising at 5.000ns	
Destination Clock:	top/plb_bram_0_connector_BRAM_Clk rising at 10.000ns	
Clock Uncertainty:	0.000ns	
Delay type	Delay(ns)	Logical Resource(s)
Tcko	0.370	top/MemorySystem/cache_0/B_Non_Cacheable_Data_Valid
net (fanout=5)	0.234	top/MemorySystem/cache_0/B_Non_Cacheable_Data_Valid
Tilo	0.274	top/MemorySystem/cache_0/Cache_ack6
net (fanout=6)	0.209	top/MemorySystem/cache_0/CHOICE1609
Tilo	0.274	top/MemorySystem/cache_0/Cache_ack25_2
net (fanout=33)	0.650	top/MemorySystem/cache_0/Cache_ack25_2
Tilo	0.254	top/vag_0/Ker668951
net (fanout=64)	0.902	top/vag_0/N66897
Tilo	0.254	top/vag_0/PLBC405DCURDDBUS_OUT¡51¿1
net (fanout=1)	1.516	top/PLBC405DCURDDBUS_OUT_0¡51¿
Tpdck_PLB	0.033	top/ppc405_0/ppc405_0/PPC405_i
Total	4.970ns	(1.459ns logic, 3.511ns route) (29.4% logic, 70.6% route)

Table D.2: Time Consumption at the end of the Read Miss Path

Delay:	2.230ns (data path)	
Source:	top/MemorySystem/cache_0/BUS_FSM_FFd4 (FF)	
Destination:	top/MemorySystem/Switched_Bus/REQ_received_R_0 (FF)	
Data Path Delay:	2.230ns (Levels of Logic = 0)	
Source Clock:	top/sys_clk_s	
Destination Clock:	top/sys_clk_s_n rising	
Delay type	Delay(ns)	Logical Resource(s)
net (fanout=4)	1.689	top/MemorySystem/cache_0/BUS_FSM_FFd4
Tsrck	0.541	top/MemorySystem/Switched_Bus/REQ_received_R_0
Total	2.230ns	(0.541ns logic, 1.689ns route) (24.3% logic, 75.7% route)

Table D.3: Time Consumption when requesting the Bus

Source:	top/MemorySystem/Switched_Bus/NextNext_to_Serve_0 (FF)	
Destination:	top/MemorySystem/cache_0/data_mem/mem0/B6.B (RAM)	
Requirement:	10.000ns	
Data Path Delay:	9.871ns (Levels of Logic = 9)	
Clock Path Skew:	-0.095ns	
Source Clock:	top/sys_clk_s_n rising at 5.000ns	
Destination Clock:	top/sys_clk_s_n rising at 15.000ns	
Clock Uncertainty:	0.000ns	
Delay type	Delay(ns)	Logical Resource(s)
Tcko	0.370	top/MemorySystem/Switched_Bus/NextNext_to_Serve_0
net (fanout=15)	0.479	top/MemorySystem/Switched_Bus/NextNext_to_Serve¡0¿
Tilo	0.274	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿28_SW0_1
net (fanout=1)	0.777	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿28_SW0_1
Tilo	0.254	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿28_SW0
net (fanout=1)	0.540	N168349
Tilo	0.274	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿56
net (fanout=3)	0.080	CHOICE6301
Tilo	0.254	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿70
net (fanout=12)	0.494	top/MemorySystem/Switched_Bus/Arbitrate_serving¡1¿
Tilo	0.274	top/MemorySystem/Switched_Bus/Ker89704_SW0
net (fanout=4)	0.334	N147546
Tilo	0.254	top/MemorySystem/Switched_Bus/Ker89704_2
net (fanout=14)	0.297	top/MemorySystem/Switched_Bus/Ker89704_2
Tilo	0.274	top/MemorySystem/Switched_Bus/Sender¡23¿1
net (fanout=18)	1.543	top/MemorySystem/Switched_Bus/Sender¡23¿
Tilo	0.254	top/MemorySystem/Switched_Bus/Mmux_B_Data2Cache__0_Result¡23¿1
net (fanout=5)	1.601	top/MemorySystem/B_DataIn_0¡23¿
Tilo	0.274	top/MemorySystem/cache_0/Mmux_B_DataIn2mem_Result¡23¿1
net (fanout=2)	0.770	top/MemorySystem/cache_0/B_DataIn2mem¡23¿
Tbdck	0.200	top/MemorySystem/cache_0/data_mem/mem0/B6.B
Total	9.871ns	(2.956ns logic, 6.915ns route)
		(29.9% logic, 70.1% route)