A Synthesisable VHDL Model For SCI Cache Coherence Protocols

Oliver Pugh

Final Year Project 2005

B.A (Mod.) Computer Science

Supervisor: Michael Manzke

Acknowledgements

I would like to thank my supervisor, Michael Manzke, for his valuable input throughout the year. I would also like to thank Ross Brennan for his advice and help during the synthesis stage especially.[1]

Oliver Pugh

University of Dublin, Trinity College

May 2005

[1]This report has been typeset using LaTeX

Contents

List of Figures

List of Tables

Abstract

The Scalable Coherent Interface (SCI) is an ANSI/IEEE standard that defines a high performance interconnect technology, providing solutions for a wide range of applications. Among other features, the standard provides optional cache coherence, using a distributed-directory approach. The highly sophisticated cache coherence protocols, designed to be implemented in hardware, are renowned for their scalability. This report outlines a project which simulates the cache coherence protocols in a small network, using VHDL.

Chapter 1

Introduction

1.1 Project Overview

The goal of this project was to design and simulate a synthesisable VHDL[1] implementation of the SCI Cache Coherence Protocols. Directory-based cache coherence protocols are designed for DSM (Distributed Shared Memory) architectures, with NUMA (Non-Uniform Memory Access) characteristics. SCI networks use a DSM architecture, generally interconnecting nodes in a ring-like structure of unidirectional point-to-point links. This VHDL model simulates a small network (of 4 nodes) using the SCI unidirectional ring-like topology. Employing a DSM architecture, global memory is distributed among each node in the network. A local cache is included in each node, for improved performance. The simulation, using ModelSim[2], allows each node in the interconnect to perform reads and writes to the global memory address space, using the SCI Cache Coherence Protocol to maintain coherency.

1.2 Organisation Of the Report

The previous section gave a brief description of the project undertaken, mentioning some of the main topics which will be covered in Chapter 2. Chapter 2 will discuss the different types of multiprocessor architectures which are used today, including

[1] **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage
[2] ModelSim is software used to simulate HDL designs.

the role of cache coherence protocols within these architectures. Different types of cache coherence protocols are examined in detail, mainly focusing on the SCI cache coherence protocol. Some background to the SCI standard will also be given in this chapter. Chapter 3 clarifies the goal of the project, and discusses the implementation steps taken. Chapter 4 will outline how testing and synthesis were carried out on this design. Test examples are also shown in this chapter. Chapter 5 will draw conclusions and possible future work is also discussed.[3]

[3]Please note that the CD attached to the back cover contains the VHDL code for the project and a copy of this report

Chapter 2

Background

2.1 Multiprocessor Architectures

Over the years multiprocessors have been designed using one of three major architectural approaches (1). These can be categorised by the way in which the processors access global memory.

2.1.1 NORMA - NO Remote Memory Access architectures

This architecture does not allow access to memory locations other than those local to the processor.

2.1.2 UMA - Uniform Memory Access architectures

This architecture allows all processors to access any memory location directly. The general structure of UMA architectures can be seen in Figure 2.1. This structure consists of a number of processors linked up to a global memory source through a common system bus. The memory access times are the same for each processor, hence the name *uniform memory access*. These machines are also known as bus-based multiprocessors or symmetric multiprocessors (SMPs).

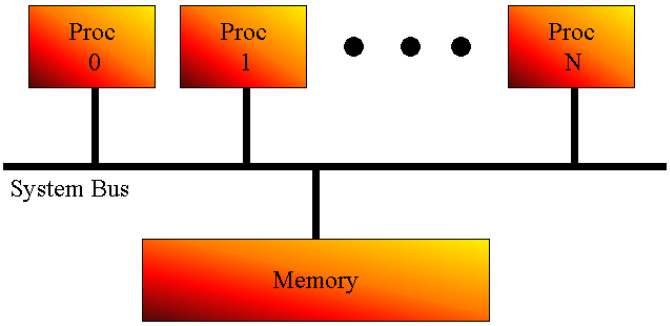


Figure 2.1: UMA Architectures

Although this architecture supports the traditional programming model, which views memory as a single, shared address space, it does however have a few major limitations. First of all, a bus is a centralised resource and therefore, an inherent serial bottleneck. Second of all, there is a fundamental limit to bus signaling (the speed of light), which limits bus lengths considerably. Bus-based UMA architectures therefore suffer from severe scalability problems and generally limit the number of processors in the network to 16 (5). Over the last few decades, efforts have been made to overcome this. Futurebus+ (2) was one which aimed to define a very high performance bus, a *superbus*, that would support a high degree of multiprocessing. PowerPath (3) bus system, used in SGI systems, was another. However, the principle problem of scalability limitation still remained. [1]

2.1.3 NUMA - Non Uniform Memory Access architectures

These architectures are similar to UMA, as all processors in the system can access any memory location directly. They differ in the fact that the memory access time is not uniform, as the memory is distributed between the processing nodes. Each processor can access the memory block within its node, performing a *local* memory

[1] SCI originated from these efforts in the late 1980's. More on this later.

access. The term *node* is used here to describe each processor and its associated local memory block. Processors in the interconnect can also access other memory blocks in other node locations, by performing a *remote* memory access. As it clearly takes less time for a processor to perform a local memory access than a remote memory access (due to the different physical distances), the memory access times are non-uniform, hence the name NUMA. The general structure of NUMA architectures is illustrated in Figure 2.2.

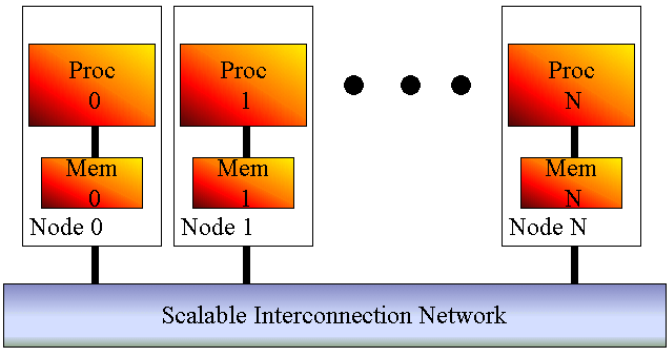


Figure 2.2: NUMA Architectures

This architecture is also known as a *Distributed Shared Memory (DSM) architecture*. NUMA/DSM architectures, overcome the scaling limitations of UMA/bus-based architectures. They also retain the traditional shared-memory programming model. DSM architectures replace the bus with a faster point-to-point interconnect, connecting pairs of nodes (generally in a ring like formation). Scalability is obviously their main advantage, but as with bus-based architectures, they have associated disadvantages too. The simplicity of the shared memory programming model is slightly compromised using this architecture, due to the non-uniform memory access times. So in order to achieve optimal performance, sophisticated software and planning are needed. The simplicity can therefore become lost with this extra overhead.

Memory access overhead can be greatly reduced (in all architecture types) through

the use of caches.

2.2 Caches

2.2.1 What is a Cache?

Caches are fast local memories which hold frequently used data. A cache exploits the temporal locality and locality of reference inherent in most programs. If memory references were random then caches would have little or no effect. If for example, a processor wishes to read an address from memory, it will first check its cache. If the information is already present in the cache, performance is improved. If the information is not present in the cache, it is then retrieved from memory and placed in the cache where it might be accessed again. As suggested above, most programs do not access memory in a random fashion, which therefore makes caches extremely useful. A cache improves performance by reducing the processor's memory access time and by decreasing bandwidth requirements to memory.

In a uniprocessor environment for example, memory access time is greatly reduced through the use of a cache. Adopting the use of caches in a multiprocessor environment is slightly more complicated, as this introduces the *cache coherence problem.*

2.2.2 Cache Coherence

The use of private caches in a shared memory multiprocessor environment, introduces an inherent cache coherence problem. If more than one processor maintains a locally cached copy of a data block from a unique memory location, any modification to the data (e.g. a write by any processor to its cached copy) will lead to data inconsistency. The cache coherence problem in a UMA/bus-based architecture is highlighted in Figure 2.3. Initially, Processor0 and Processor1 both read from memory location X=0 and store the data copy in their respective caches. Processor 0 then modifies its cached copy using a write transaction with a value of 1. Processor1

and the memory source now both have *stale* copies of location X. This is known as the cache coherence problem.

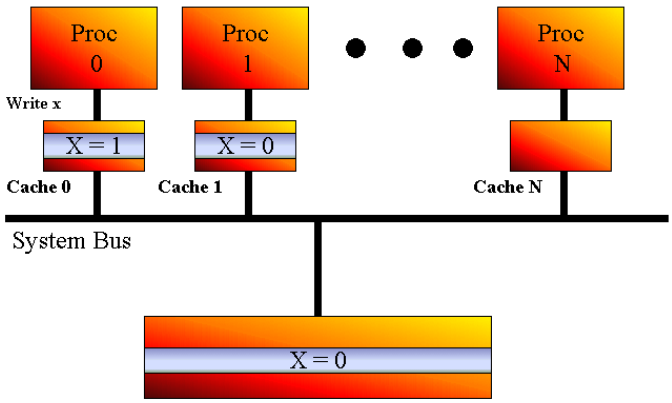


Figure 2.3: The Cache Coherence Problem in a Multiprocessor Environment

Figure 2.4. illustrates the inclusion of caches in a NUMA/DSM architecture. The cache coherence problem can either be solved in software or hardware. Solving the problem in software (i.e. leaving it up to the programmer) was the first approach used. These machines (6)(7)(8) were obviously quite difficult to program and had a lot of overhead associated with them. Today, the cache coherence problem is solved through hardware (5), using a *cache coherence protocol*.

2.2.3 Cache Coherence Protocols

A system of caches is said to be coherent if all copies of a main memory location in multiple caches remain consistent when the contents of that memory location are modified (4). In order to avoid using invalid/stale data in a cache (i.e. for caches to remain consistent), a cache coherence protocol is used. A cache coherence protocol performs certain actions when a processor writes to a specific global memory address location, which exists in other caches. These actions are usually to either invalidate or update the other copies of the block. Most cache coherence protocols use the invalidation technique, as it is easier to implement in hardware [PhD].

Cache coherence protocols fall into two principle categories (5), *snoopy*(9) and *directory-based*(10)(11).

2.2.4 Snoopy Cache Coherence Protocols

Snoopy cache coherence protocols are only suitable for bus-based multiprocessor systems, as they require the use of a broadcast medium. Since a bus allows all connected processors to observe all bus transactions, each cache can monitor all memory transactions. Each cache *snoops* the bus and if it observes a transaction which will affect the consistency of a block of data in its cache, the cache coherence protocol intervenes. As mentioned previously, the cached block may be either updated or invalidated, depending on the protocol employed.

Snoopy cache coherence protocols generally behave in the following manner. When a cache observes a write transaction on an address contained in its cache, it invalidates its cached copy. When a cache observes a read transaction on an address contained in its cache, its cache state is checked. If the cache state indicates that it has the most up-to-date copy, it intervenes before memory can reply, supplying the data to the requesting processor.

Some well known snoopy protocols are the Firefly(12) and MESI(13)(14) protocols. Since *snoopy* cache coherence protocols are only suitable for broadcast-enabled interconnects (UMA/SMP architectures), a different type of protocol needs to be used in a NUMA/DSM architecture. The type of protocol used in DSM architectures is known as a *directory-based* cache coherence protocol.

2.2.5 Directory-Based Cache Coherence Protocols

A cache coherence protocol that does not use a broadcasting interconnect/medium must store the locations of all cached copies of every block of shared data, regardless of whether the locations are centralised or distributed among a number of processors. A *directory* is the name of the structure used to store this information. Each shared block is assigned a directory entry which contains information about the

location(s) of the shared data block. A directory-based cache coherence protocol uses this structure on a cache miss, and takes appropriate action, depending on the transaction involved and the current state of the directory.

Using a single directory in a DSM architecture would only defeat the purpose of distributing the memory between the nodes in the network, therefore, the directory needs to be distributed also. Each node in a CC-NUMA (Cache Coherent Non-Uniform Memory Access) interconnection network contains a block of local memory, its associated directory and local cache. This reduces the potential bottleneck and bandwidth issues that would otherwise occur if the directory was not distributed. This structure is shown in Figure 2.5. Directory-based cache coherence was first proposed by Tang(15) and Censier and Feautrier(10). In contrast to snoopy cache coherence protocols, the cache location(s) of the shared block of data (i.e. the node location(s)) are known. The advantage of this is that individual messages can be sent to each node in the shared list and therefore, can be sent over any interconnection network.

There are two major components to every directory-based cache coherence protocol (5):

- The directory organisation
- The set of message types and message actions

The directory organization refers to the actual structure used to store the directory information. Storing directory information requires extra memory, which is not required in bus-based systems. This additional memory is known as the *directory memory overhead* (the ratio of the *additional* memory needed for the directory to the total memory).

The directory-based cache coherence protocol needs to be able to send messages over the interconnect to other nodes, containing coherence actions. Each message needs to have a message type or instruction associated with it. Each node can then take appropriate coherence action depending on the message received.

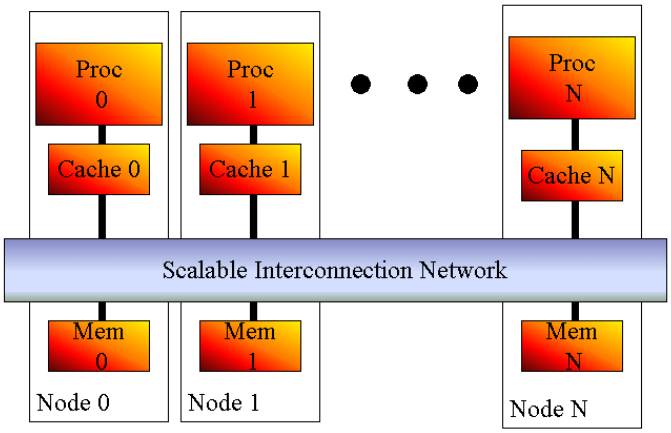


Figure 2.4: Cache Coherent Non Uniform Memory Access (CC-NUMA)

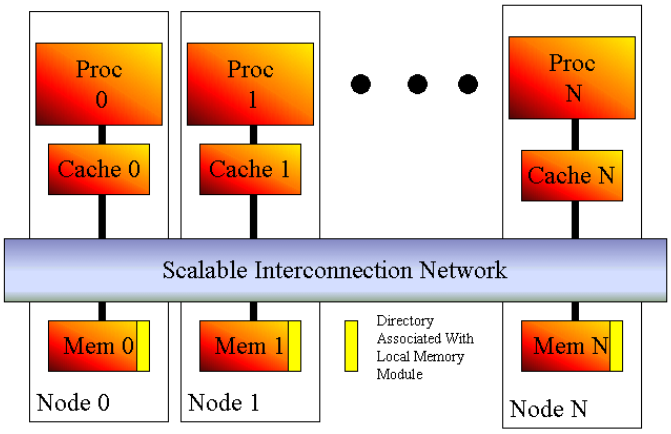


Figure 2.5: Distributed Directory Structure

Directory-based cache coherence protocols can be classified into three different categories (16): *full map directories, limited directories* and *chained directories*.

2.2.6 Full Map Directories

Full map directories store information in the directory for every node in the network, indicating whether the data block is present or not in each cache. So each directory entry contains N (groups of) indicators, where N is the number of nodes in the system. An example of a protocol that uses a full map directory structure would be the *bit-vector* cache coherence protocol (10).

Bit-Vector Cache Coherence Protocol

The bit-vector protocol associates *one bit per processor* in its directory for each shared block of local memory data. Each bit in this vector indicates whether or not the shared block is present in the corresponding nodes cache (the bit is set if present). There is also a dirty bit associated with each block in the directory. If this bit is set, it means that only one processor has a copy of the block of data. This processor therefore has exclusive ownership of the block (i.e. it can perform a write transaction to it). A cache entity in this protocol, maintains two bits per block as part of its cache line. One bit indicates whether the block is valid, while the other indicates if the block can be written to (if set then it has permission). The cache coherence protocol maintains the coherency of these indicator bits in the directories and caches of the nodes in the network. The bit-vector protocol is regarded as the simplest of all the cache coherence protocols (5) due to its speed and efficiency.

An example of three different states in the bit vector cache coherence protocol can be seen in Figure 2.6. The local memory pictured in the diagrams, represents a local memory block of an arbitrary node in the network. State (a) illustrates the situation where no node in the network has a copy of address A in their respective caches. Note that the dirty bit is not set, as no node has ownership (write permission) of the block. Three nodes (Node 0, Node 1 and Node N) then request a copy of address A, in the form of a read transaction. State (b) illustrates the resulting situation. The

memory location A updates its directory information by setting the presence bits corresponding to the node number of the requester. Node N then requests ownership of the block (write permission). State (c) shows the resulting situation. The only node that has a copy of the block is Node N, therefore the dirty bit is also set.

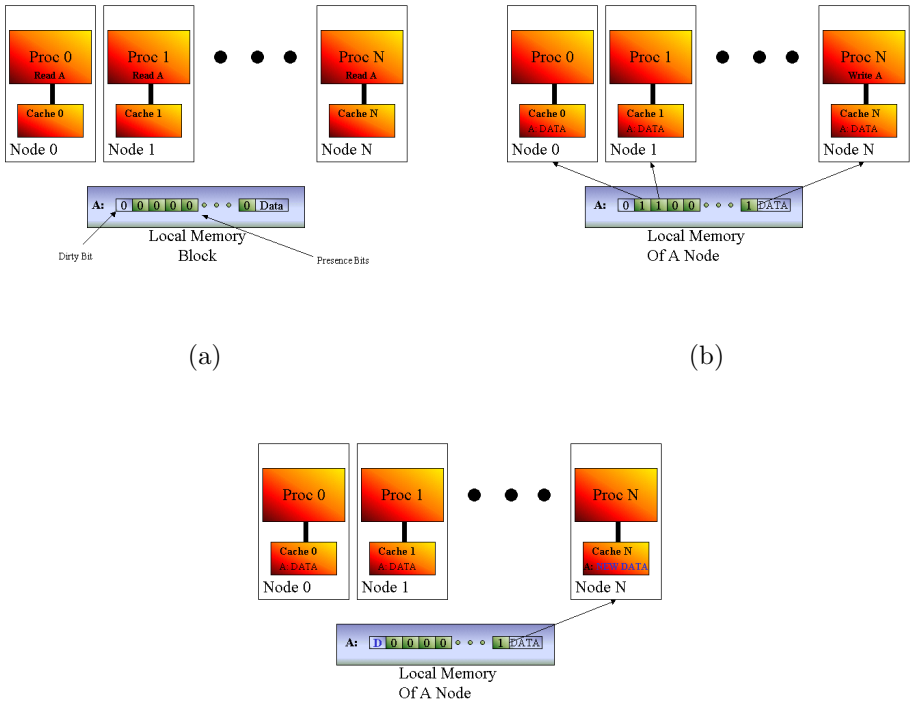


Figure 2.6: Bit Vector Cache Coherence Protocol

In order for the transition from State (b) to State (c) to occur, the cache coherence protocol takes the following steps:

1. Processor N wants to write to memory location A so it checks its cache to see if it contains a copy. It sees that a valid copy of the data is present, but it does not have write permission of the block (this is denoted by its cache exclusive bit being 0).

2. Processor N issues a write request to the Node in the network whose local memory contains address A (lets call it Node X).

3. The corresponding remote memory module sends an invalidation messages to the other members of the shared list (Node 0 and Node 1).

4. Node 0 and Node 1 receive the invalidation messages, update their caches by wiping the cache line associated with address A. Each node will send an acknowledgement back to Node X when completed.

5. Node X receives the acknowledgements, sets the dirty bit in its local memory (dirty bit = 1), clears the pointers to Node 0 and Node 1, and sends write permission to Node N.

6. Node N updates its cache by setting the exclusive bit and performs its write transaction.

By waiting for the invalidation acknowledgements from Node 0 and Node 1, the protocol guarantees that the memory system is sequentially consistent. This protocol does however have scalability limitations associated with it, as the number of indicator bits in each directory scales linearly as the number of nodes in the interconnect increases. Limited directories were designed to solve this problem.

2.2.7 Limited Directories

Limited directories are similar to full-map directories except each presence bit represents a group of nodes in the system. If a bit is set, it means that at least one of the nodes in the group has a cached copy of the data. An example of a protocol that uses a limited directory is the *coarse-vector* cache coherence protocol (17).

Coarse-Vector Cache Coherence Protocol

The bit-vector protocol can be converted to a coarse-vector protocol when the number of nodes in the network increases beyond a certain level. Lets say for example that this level is 32. The bit-vector and coarse-vector protocols operate the same way as long as the number of nodes in the network is less than or equal to 32. Once the number of nodes exceeds this level the protocol then adjusts its directory structure. As long as the node count is between 33 and 64, for example, each bit of the

vector represents 2 nodes. If one or both of the nodes contain a shared copy of the block then the corresponding bit is set. Similarly, if the network scales to between 65 and 97 nodes, each bit of the vector will represent 4 nodes. As you can see form this, the conversion is straightforward. The coarseness of the protocol is defined to be the number of nodes in the interconnect, where each bit of the vector represents (5). The bit-vector protocol therefore has a coarseness of one.

An example of this transformation from bit-vector to coarse-vector can be seen in Figure 2.7. State (a) shows the bit-vector being implemented for up to 32 nodes. When the nodes in the interconnect are extended beyond this (up to 64), the coarse-vector protocol is used, where each bit of the vector represents two nodes.

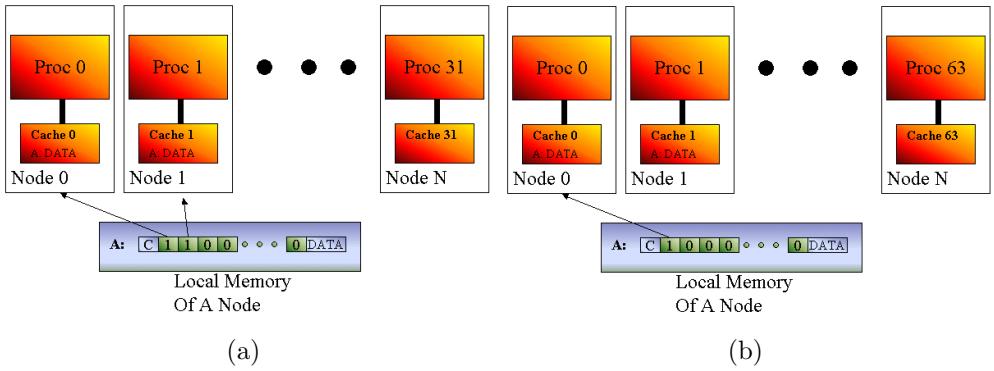


Figure 2.7: Bit Vector Cache Coherence Protocol

An example system which implements this is the SGI Origin 2000 (18), using a bit-vector protocol for processor counts less than or equal to 128. Once the processor count exceeds this number, the coarseness jumps to eight.

The advantage of this protocol is that the extra memory overhead remains fixed as the number of nodes increases (possibly up to thousands). The coarse-vector protocol adjusts the directory structure as the number of nodes increases. The protocol transitions for both the bit-vector and coarse-vector are exactly the same. The only difference is that the coarse-vector directory doesn't know how many nodes are currently sharing a given data block. This introduces inefficiency when the protocol is invalidating a shared list. An invalidation message must be sent to each node in the group, even if there is only one node in that group that contains a shared

copy of the data block. Each node also has to send an acknowledgement back to the home node upon receiving this invalidation message, generating even more traffic on the interconnect.

2.2.8 Chained Directories

Chained directory cache coherence protocols maintain a linked list (or chain) of pointers to keep track of the nodes that contain shared copies. A well-known protocol that implements this concept is the Scalable Coherent Interface (SCI) cache coherence protocol (19).

SCI Cache Coherence Protocol

Section 2.4 will discuss the general background and main topics of the SCI standard, but for now the focus will be on the functionality of the cache coherence protocol.

A shared memory block in SCI, will contain a pointer to the head of the shared list. Each node in the shared list maintains a forward and backward pointer to its adjacent neighbours in the list. The nodes in the shared list therefore maintain a doubly-linked shared list, hence the name *chained* directory structure. Figure 2.8 illustrates how the SCI protocol maintains coherence using this doubly-linked list.

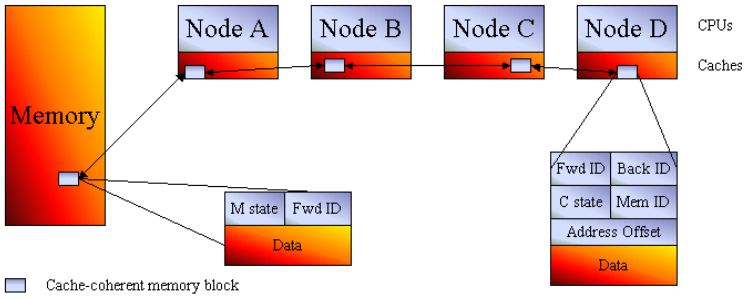


Figure 2.8: SCI's Sharing List and Coherence Tags

In SCI the memory or cache block size is 64 bytes. This results in only a few percent directory storage overhead in the memory and cache modules of each node. This overhead is fixed, regardless of the system size, highlighting the scalability advantages of this protocol. It is also worth noting that the cache coherence can

be enabled or disabled on different blocks of memory. As can be seen in Figure 2.8, each coherent memory block contains a state field and a forward pointer. The forward pointer identifies the node at the head of the shared list (if the block is shared). Each cache entry also contains a state field but has two pointers, forward and backward, to maintain the doubly-linked list. The basic functionality of the SCI cache coherence protocol is as follows.

Read Transaction

There are two cases we need to examine here. The first is when a memory block in the system is not shared. If a node A for example, wishes to read from this global memory location, it first checks its cache for the address. A cache miss occurs, and the node performs a read transaction to the remote location of the address (node B for example). Node B's memory controller checks its memory state and sees that the block is not shared. The memory controller updates its state information to the shared state, updates its forward pointer to node A's address, and sends the data block to node A. Node A writes to its cache with the data and updates its cache state information, establishing itself as head of the empty list (H.O.E.L). This final situation is depicted in Figure 2.9 (a).

The second case is when the memory block is shared. Upon receiving a read request from node A, node B's memory controller updates its pointer field to node A, but does not send the data (as it might have a stale copy). The current head of the shared list (node D for example) is then sent a message ordering it to update its backward pointer, state information and to return the data to the requesting node, node A. Node B will process this message, updating its backward pointer field, degrading itself to a R.L.E and sending the most up-to-date copy of the data block to node A. Node A can then update its state and pointer fields upon receipt of node D's message. Node A will process this message updating its backward pointer to the memory location and forward pointer to node D. Node A has now established itself as the new head of list entry (H.O.L). If other nodes were part of this shared list, there would be no need for them to be involved in these steps, thus highlighting the

scalability advantages of this protocol. Therefore the update only involves 3 nodes at most, regardless of the size of the shared list. This final situation is depicted in Figure 2.9 (b).

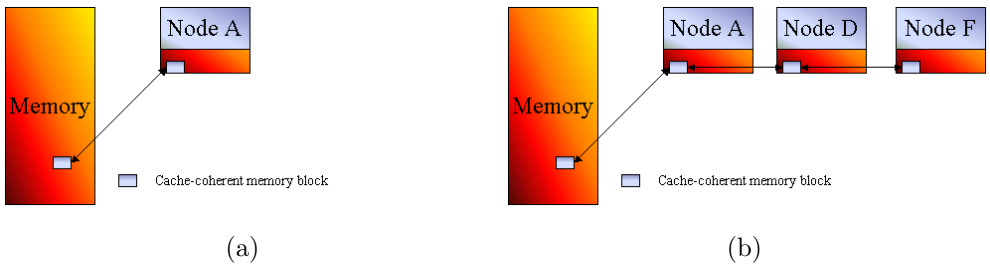


Figure 2.9: SCI Read Transaction

Write Transaction

In order for a node, node B for example, to perform a write transaction on a shared block of data, it first needs to make itself head of the shared list (if it is not already). Once it becomes head of the list it needs to delete all other entries to obtain an exclusive copy of the data block. The SCI cache coherence protocol states that only a node with an exclusive copy of a data block has write permission to that data block. When deleted, all nodes of the shared list will have discarded their cache line entries for that particular data block. Node B will now have an exclusive copy of the data block, gaining permission to modify the data.[2] This write transaction is more complicated than it sounds however.

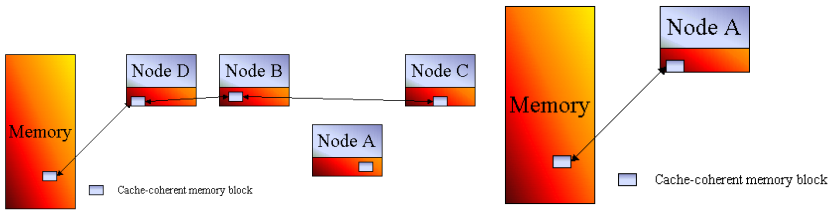
Lets look at an example where a node A is part of a shared list containing more than one element. If node A wants to perform a write transaction to a memory block, there are 3 possible scenarios that need consideration.

- The first scenario is when the node A is head of the list (H.O.L). In this case, the node simply deletes the other list entries using a purge transaction, thereby obtaining write permission to the block.

[2]It is worth noting here that some SCI implementations allow the node to modify the data before the shared list has been purged.

- The second scenario is when the node A is a tail list entry (T.L.E). In this case, the node A will update the state of the 2nd last node, whose address will be stored in node A's backward pointer field. As node A has now *popped* itself out of the shared list, it can then make itself head of the list and delete the other list entries. Now with write permission, node A can perform its write transaction.

- The third scenario is when the node is a regular list entry (R.L.E). In this case node A will update the pointers of its neighbouring nodes, *popping* itself out of the shared list. The node A will then make itself head of the list and purge/delete all other entries. An example of this can be seen in Figure 2.10. Node A updates node B's forward pointer to node C's address. Node A updates node C's backward pointer to node B's address. Node A has therefore popped itself out of the list, illustrated in (a). Node A then makes itself head of the list and deletes all other entries, finally gaining an exclusive copy of the memory block, illustrated in (b).



(a) (b)

Figure 2.10: SCI Write Transaction

If more than one node sends a read/write request at the same time, the SCI cache coherence protocol states that their requests get dealt with in the order in which they arrive at the local nodes memory controller.

2.3 Evaluating Cache Coherence Protocols

A number of studies have been carried out, evaluating different cache coherence protocols (5)(31)(33)(35). One particular study was carried out in 1998 (5) using the

Stanford FLASH multiprocessor(30). It examined the performance and scalability of each of the directory-based cache coherence protocols mentioned in this report. The results of this study found that the optimal protocol changes as the machine size scales or sometimes when architectural aspects (like cache size) are changed. None of the protocols were able to optimize their behavior in terms of:

1. Protocol Memory Efficiency
2. Direct Protocol Overhead
3. Message Efficiency
4. Protocol Scalability

At high processor counts, the bit-vector protocol has a large memory overhead, as the width of its directory entity becomes unwieldy. The coarse-vector protocol solves this problem, but due to its imprecision in the manner which information is shared, it leads to large amounts of message traffic in many applications. At small processor counts, the SCI protocol performs badly. At large processor counts however, the SCI protocol is generally the best one to use, due to its scalability advantages.

2.4 SCI

The Scalable Coherent Interface (SCI) (19)(20) is an ANSI/IEEE standard that defines a high performance interconnect technology, providing solutions for a wide range of applications. As mentioned previously, efforts were made in the late 1980's to develop Futurebus+(2). This led to the specification of the SCI standard, which implements a DSM architecture. This was approved in 1992.

The SCI interconnect, the memory system and the associated protocols are fully distributed and scalable. The main objective of SCI (28) is to deliver high communication performance to parallel or distributed applications. SCI was designed to be scalable and it is possible to connect up to 64K nodes. A node can be made up of a

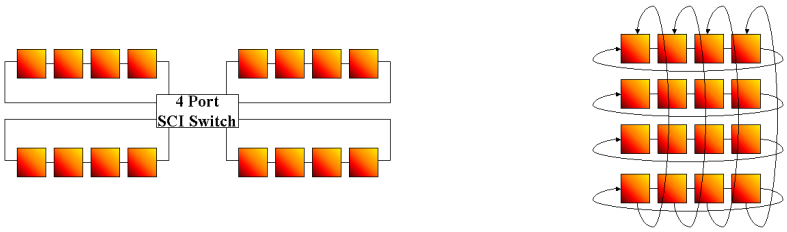
workstation or server machine, a processor, a memory module, I/O controllers and devices or bridges to other buses/interconnects. Each node attaches to the network using a standard interface. The basic transfer unit is a packet, which eliminates the overhead of bus-cycles.

SCI allows data transfer at nearly 500MHz, achieving a one gigabyte per second transfer rate. Adding nodes to an SCI network also adds bandwidth, so performance scales well. In order for SCI to be a success the developers realised that certain goals needed to be achieved (28). High communication performance, scalability, a coherent memory system and a simple interface were the key areas focused upon.

2.4.1 SCI Topologies

SCI nodes are interconnected through unidirectional point-to-point links in a ring/ ringlet topology. Certain *housekeeping* tasks such as maintaining certain timers, discarding damaged packets (so they don't circulate the ring indefinitely) and circulating ring maintenance information are assigned to one node within the ring. This is called *the scrubber*. Switches are used to connect multiple independent SCI ringlets. In systems today, there are two common topologies which are used to implement this. These are shown in Figure 2.11. The first topology is shown in (a). There are 4 ports on the switch (with 2 extra extension ports - not shown). Using the extension ports, the switch can either be expanded to a stacked switch with possibly 16 ports or configured to a non-expandable 6-port switch.

The second topology is shown in figure (b). This multidimensional tori uses small SCI ringlets at each dimension. Each node is connected to each dimension and uses a small switch integrated into an SCI adapter to provide cross-dimensional packet transmissions. It is possible to create a 3-dimensional tori with up to 10-12 nodes in each dimension.



(a) (b)

Figure 2.11: SCI Topologies

2.4.2 SCI Node Interface Structure

The SCI node interface to the network needs to be able to transmit packets, while concurrently accepting packets from other nodes. To implement this, FIFOs are used to hold symbols received while a packet is being sent. Node application logic is not expected to match the SCI link speeds; therefore input and output FIFOs are needed. So in order to match the higher link transfer rate, nodes need to ensure that all symbols within one packet are available for transmission at full link speed. In general, the SCI node maintains two queues, which serve as buffers until transmission bandwidth becomes available for outbound packets or until inbound packets can be processed by the nodes application logic. This concept is depicted in Figure 2.12.

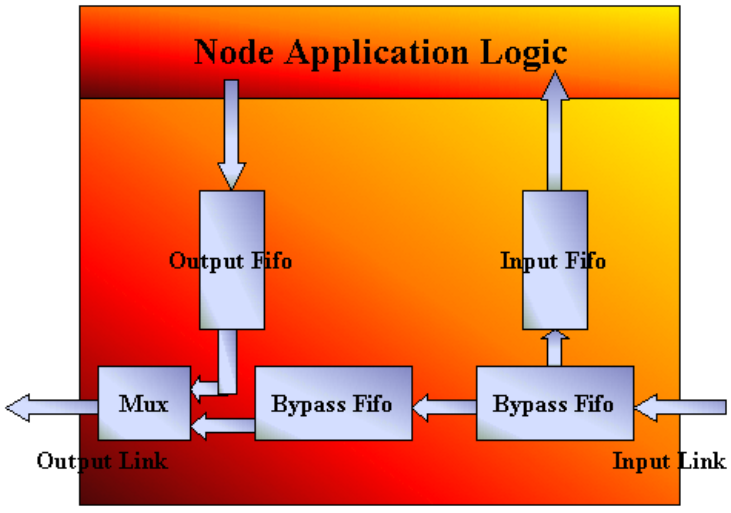


Figure 2.12: SCI Node Interface Structure

2.4.3 SCI Transaction

Transactions are split into a request and a response sub-action. Packets carry addresses, command and status information, and data (depending on the transaction). Up to 64 transactions can be outstanding per node. Each sub-action consists of a send packet (generated by the sender) and an echo (acknowledgement) packet returned by the receiver. The echo tells the sender whether the packet was accepted (stored in the receivers input queue) or rejected (due to a full input queue). In the former case, the sender can discard the send packet from its output queue. In the latter case, the sender re-transmits the packet. There are 3 types of transactions in SCI.

1. Transactions with responses (read, write and lock transactions).
2. Move transactions (for example non coherent writes). These do not have response sub-actions. They are therefore more efficient than writes.
3. Event transactions. These do not have responses and do not generate an echo. They can be used to distribute a time stamp for global time synchronisation within the SCI system.

2.4.4 Distributed Shared Memory In SCI

As mentioned previously, SCI was a solution to the inherent serial bottleneck that the bus leads to, but the standard still maintains bus-like services. Just like a bus, DSM architectures, like SCI, have the ability to support remote memory accesses for both read and writes. SCI uses a 64-bit fixed addressing scheme. The upper 16 bits specify the node on which the addressed physical storage is located [3], while the lower 48 bits specify the local physical address within the memory of the node being addressed. As with all DSM architectures, the nodes in the interconnect can access this global physical address space and hence any physical memory location within the whole network by mapping parts or segments of this memory space into their own memory.

[3] As mentioned previously, SCI can support up to 64K = 2$\hat{1}$6 = 65536 nodes

2.4.5 Cache Coherency in SCI (Revisited)

Most high performance processors use local caches to reduce effective memory-access times. Cache-coherence protocols define mechanisms that guarantee consistent data. In SCI, the cache coherence protocols are provided as options only. Therefore a compliant SCI implementation does not need not implement cache coherence. Simulations of the specification (given by the C code which is provided with the standard) have greatly helped in designing and debugging the protocols. Despite the well-devised protocols and the provided C code, it is regarded as extremely difficult to not only implement the full protocol set, but to interface it to typical processor coherence schemes (28). As mentioned on page 20, an SCI node can be a bridge to other buses or interconnects. These buses or interconnects could possibly be employing snoopy cache coherence protocols, requiring the SCI cache coherence protocol to integrate with it.

2.4.6 Commercial use of SCI

SCI is currently used in a large variety of scenarios and products(1). Nearly all applications and products are based on the technology of Dolphin Interconnect Solutions(21). In all SCI systems, nodes are connected to the SCI network using the Link Controller (LC) chip. The chip, currently in its fifth generation, allows processing speeds of up to nearly 1GBps, which was the original goal of the SCI standard in 1992. Using this component, SCI bridges to other system busses (e.g. the SUN SBus(22) and the PCI bus(23)) and SCI switches to larger systems are provided by Dolphin.

SCI chips and adaptors are being used more and more by different companies to develop their own systems. Practical applications have been developed by a number of different commercial vendors (20). Here are some examples:

1. Fujitsu Siemens have developed a product, hpcline (24), using SCI technology to achieve high performance and scalability in communication intensive applications. This product is being used as complex business solutions for different

commercial customers.

2. Philips Medical have developed a state-of-the-art ultrasound system, xSTREAM (25), using SCI technology.

3. NLX Corporation, a provider of simulation and training systems, have developed multi-channel image generators, wide field-of-view visual display systems, high-fidelity sound generation and high payload 6 degree-of-freedom motion systems, utilising the high performance aspect of the SCI technology (26).

4. Camber LTD provide products for military and commercial use. They have developed a product, using SCI's high performance technology, called Battle Vision (27). This product provides an enhanced real-time flight simulation with geographic information, meteorological effects and atmospheric effects.

The most common use of SCI however (1), is in commodity PC clusters based on the PCI-SCI bridge or adapters, developed once again by Dolphin.

Chapter 3

Implementation

This chapter will discuss the approach to and implementation of this project. The first section of this chapter will clarify the goal of my project, now that the background issues have been discussed.

3.1 The Goal Of My Project

The SCI standard (19) specifies three sets of implementations for its cache coherence protocols, in order to simplify explanation.

1. The Minimal Set
2. The Typical Set
3. The Full Set

The goal of this project was to implement a Synthesisable VHDL Model of SCI's Cache Coherence Protocols. Therefore all parts of the SCI standard were not required to be implemented. In order to implement the distributed directory structure which the protocols are based on, a DSM architecture was used. It was not necessary however to use the exact SCI node structure or its split transaction method of communication between nodes in the interconnect. These features would have been extremely difficult and time consuming to implement in VHDL, so as a result they were not used. The full SCI cache coherence protocols use a vast

amount of memory and cache states. These states are mostly based on the SCI split transactions, making the majority unsuitable for this project. In other sources researched on the protocols (5)(28), a number of key states are used. The design of the protocols were therefore based on these key states. The following memory states were used in this design:

1. Shared
2. Not Shared

The cache states for this design are as follows:

1. Head Of Empty list (**H.O.E.L**)
2. Head Of List (**H.O.L**)
3. Regular List Entry (**R.L.E**)
4. Tail List Entry (**T.L.E**)

3.2 Project Approach and Implementation

This project was implemented in two main stages. The first stage was to design a simple DSM architecture in VHDL, allowing each node in the interconnect to perform read and write transactions to global memory. As a DSM architecture was to be employed, each node was to consist of a local memory module (which was part of the global memory of the system). The interconnect used was to model SCI's simple ring-like structure of unidirectional point-to-point links.

The next stage was to add a cache component to each of the nodes, requiring the design of the VHDL-based SCI distributed directory cache coherence protocols. From October to the end of December 2004, the design and coding of the simple interconnect was carried out. From January to March 2005, the final stage was implemented,

3.2.1 VHDL

As suggested by the title of the project, the VHDL programming language was used to implement the design. Coding was performed using Xilinx Integrated Software Environment (ISE) 6.3i software. VHDL is a language used to describe and simulate complex digital systems. A digital system in VHDL consists of a design *entity/module* that can possibly contain other entities, which are then considered components of the top-level entity (29). An entity in VHDL consists of an *entity declaration* and an *architecture body*. The entity declaration defines the relationship to the outside world and defines the input and output signals/wires. The architecture body contains the functional description of the entity. The architecture body of the top level entity consists of all the other entities interconnected, along with its own possibly additional functionality, all operating concurrently.

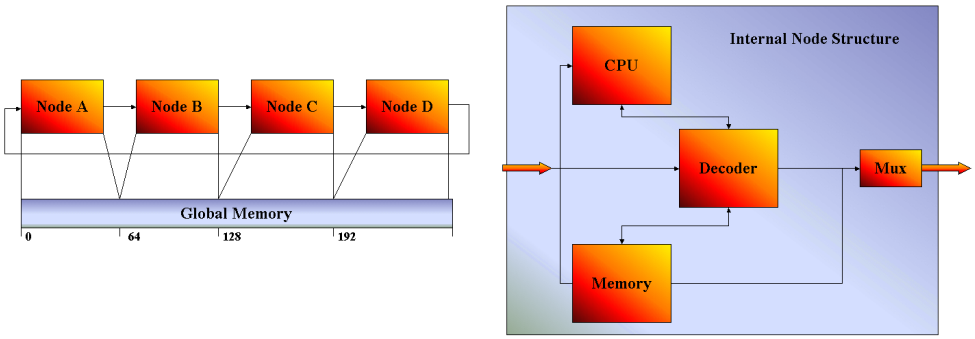
A VHDL entity can be instantiated into another higher level entity through the use of a feature called *port-mapping*. This report will discuss each of the entities implemented in the design process, using a high level description. Knowledge of how a VHDL entity is written is therefore not required.

3.3 Initial Implementation

As mentioned previously, the goal for this part of the design was to simulate a number of nodes in an interconnect of unidirectional point-to-point links, performing read and write transactions to a global address space. This global address space was to be distributed among the nodes in the network - conforming with the DSM architecture used by SCI. An interconnect of 4 nodes was decided upon for simplicity. Global memory was to have an address space of 256 elements, allowing each node's local memory module to hold 64 addresses. Figure 3.1 (a) illustrates this concept.

3.3.1 Internal Node Structure

Before the coding could begin, a node structure needed to be designed. Each entity that was designed implements a different piece of functionality. The following



(a) SCI Interconnect (b) Internal Node Structure

Figure 3.1: Initial Implementation

components were decided upon for each of the 4 nodes:

1. A CPU entity, which generates read and write transactions to the global address space.

2. A Local Memory module which, as part of the DSM structure, stores addresses in an array structure.

3. A Decoder entity, which decodes the addresses sent to it by the CPU, deciding whether a local or remote transaction is to be performed.

The following sections outline the functionality of these entities. Figure 3.1 (b) depicts the internal structure of each node in the interconnect. The bidirectional links are actually implemented as two unidirectional wires/signals. Each pair of connected entities communicate with each other by assigning values to their output ports (to send data), and by reading data from their input ports (receiving data).

The CPU Entity

The CPU entity is responsible for generating read and write transactions to the global address space of the interconnect. A pseudo random number generator (PRNG) was used to generate the 8 bit addresses [1]. The algorithm behind the PRNG used in this project is described in Appendix A. Using this simple concept,

[1] This was suggested by Michael Manzke

the code was formulated in VHDL. The extract from the CPU entity which implements this code is shown below.

```
1 --generate pseudo random number for address:
2 temp_addr:="000" & nxt_addr(7 downto 3);
3 temp_addr2:= temp_addr xor nxt_addr;
4 temp_addr:=temp_addr2(2 downto 0)&"00000";
5 nxt_addr:=temp_addr xor temp_addr2;
```

The basic functionality of the CPU can be summarised in the following steps:

1. On a reset the CPU will set the initial address or seed for the pseudo random number generator.

2. The CPU will send a 41-bit transaction to the decoder when an enable signal is set on its input port. The structure of this packet format can be seen in Figure 3.2. The most significant bit signifies whether the transaction is a read or write. This bit is followed by an 8-bit address, selecting one of the 256 global addresses in the interconnect. The address is followed by a 32-bit data value. Unless a write transaction is being sent, this data field is negligible[2].

3. The CPU is informed by the Decoder entity as to which input line it will be receiving a reply on. The CPU will either receive data directly from memory (on a local transaction) or from the decoder (on a remote transaction).

4. The CPU will compare the address it sent out to the address it receives on the correct input line (bits 39 down to 32). When the addresses match, the CPU has received the correct data (if its a read transaction) or received confirmation that a write took place (if its a write transaction).

5. The CPU will then use the current address as the seed to generate the next random number, which represents the next address.

[2]If certain bits of a signal are negligible in VHDL, they must still be assigned a value

Figure 3.2: CPU-Decoder Packet Format

The Local Memory Entity

Each local memory module contains 64 addresses of the global address space. In VHDL an array can be used to represent a memory structure. For node A, the address lookup is straightforward as it will be passed addresses in the range of 0 to 63, thus satisfying the array bounds. For all other nodes, an offset needs to be subtracted from each address in order to access the correct location in the array. Taking node B for example, when an address is sent to the memory module from the Decoder, it needs to subtract an offset of 64 from it. Likewise, node C needs to subtract an offset of 128 from the address and node D needs to subtract an offset of 192.

The memory was designed with a 2 bit control signal input, controlled by the Decoder entity. The memory performs one of the following 4 transactions, which are explained in greater detail in the Decoder Entity section:

1. A read transaction, returning the data to the CPU (local read).
2. A write transaction, confirming completion to the CPU (local write).
3. A read transaction, sending the data out onto the interconnect (remote read).
4. A write transaction, confirming completion to the remote node that performed the write (remote write).

This sums up the local memory module functionality of the initial implementation. More detail on the 4 different functions provided by memory will be discussed in the next section.

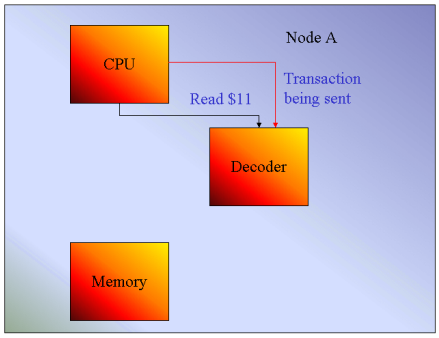
The Decoder Entity

The decoder acts as an interface to the node. The Decoder entity is also responsible for deciding whether its CPU is performing a local or remote transaction. The CPU entity initiates communication with the Decoder by asserting a control signal (not shown in Figure 3.1 (b)[3]). It will then assign data to the output port which feeds into the Decoder entity. This data will specify a transaction for the Decoder to process. The Decoder will first examine the address field of the packet. A decision is made based on this address, as to whether a local or remote transaction will take place.

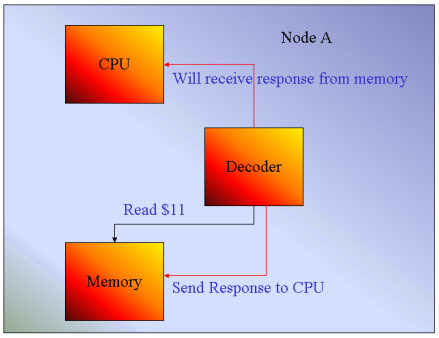
If the address indicates a local transaction (for Node A the address would need to be between 0 and 63), the Decoder sets the memory control signal to the local read state. This asks memory to perform the local read/write and reply to the CPU. The transaction will also be forwarded to the memory via the correct output port. The Decoder simultaneously informs the CPU, through the CPU control signal, that it will be receiving a reply from the memory entity. As discussed earlier, this reply will either be a data block (if a read transaction took place) or confirmation of write completion (if a write transaction took place). An example of a read transaction is illustrated in Figure 3.3. The red lines denote the control signals being used.

If the address indicates a remote transaction (i.e the address is not contained in the nodes address space), the Decoder sends a packet out onto the interconnect and waits for a reply (i.e. a return packet). The packet structure for the initial implementation is depicted in Figure 3.4. When set, the S bit indicates that a Node has sent a transaction out onto the interconnect, and that the packet is still trying to find its destination. Each node the packet arrives at, checks whether the address is located in its address space. The packet is forwarded from node to node until it finds the correct destination. The Decoder in that node will then assert a control signal to its memory module. This control signal tells memory to process the transaction (read or write) and to then forward the response out onto

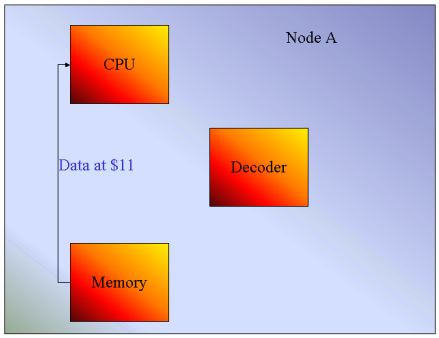
[3]It should be clear by now that control signals are used to decide which piece of functionality the receiver is to implement



(a) CPU sends read request



(b) Decoder sends request to memory



(c) Memory returns data to CPU

Figure 3.3: Local Read Transaction

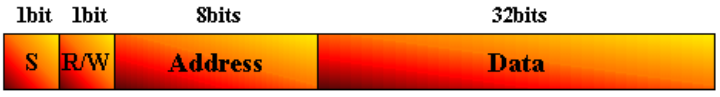


Figure 3.4: Initial Interconnect Packet Structure.

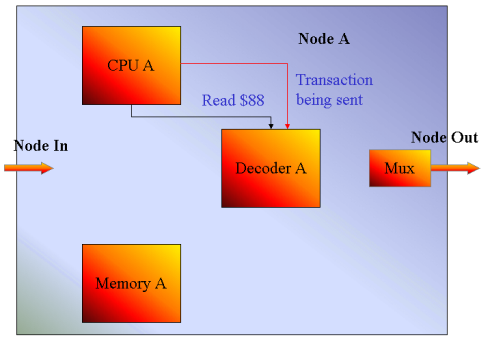
the interconnect. Memory processes the transaction. Before forwarding the return packet out onto the interconnect, the memory entity sets the S bit of the packet to 0. If a read took place then the data field is updated with the block value. The address field is not modified. This new packet is the transaction response and is forwarded out onto the interconnect. The original node eventually receives the response packet and checks to make sure the S bit is not set. Once this requirement has been fulfilled, the Decoder then sets the CPU control signal low, indicating that the response will come from the Decoder. The data is then sent to the CPU, completing the transaction. This remote transaction is illustrated in Figure 3.5. The 2:1 mux entity has not been mentioned yet however. It has a single bit control signal from the decoder. The mux either outputs the 42-bit data packets from memory or from the decoder, depending on the Decoder's control signal.

This sums up the overall functionality of the Decoder entity. As suggested by the CPU entity section on page 28, each node would execute transactions concurrently, sending packets out onto the interconnect for remote transactions. If this were allowed, some sort of flow control would be needed. In SCI this is done using input and output FIFO's. This however, was not the focus of the project, so another solution was used.

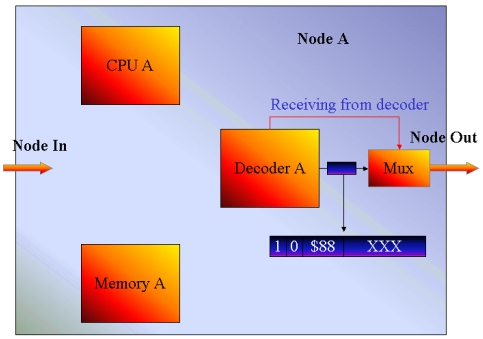
In order to add control to the top level design (i.e. the four nodes interconnected in the ring-like structure), another entity was required. This additional functionality allows only one CPU to perform a transaction at a time. To implement this control element, a Finite State Machine was used inside a *Scheduler* entity. This Scheduler entity is described in the next section.

The Scheduler Entity

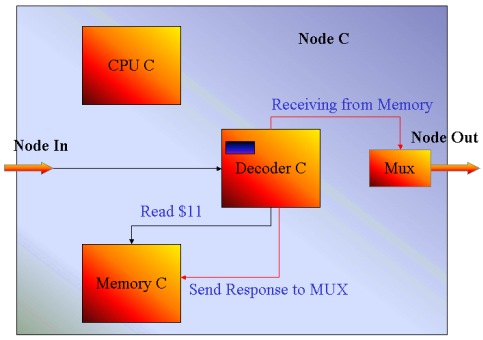
This scheduler entity feeds a control signal into each node, enabling one CPU at a time. Each CPU has a request line output, feeding into the scheduler entity. On a reset, node A is given the first *time slot* by default. Once it has completed its transaction, it de-asserts its request line temporarily, allowing the scheduler to allocate the next time slot to node B. Node C will receive the next time slot, followed by node D. This functionality is implemented using a simple Finite State Machine. The final top-level design for the initial implementation is illustrated in Figure 3.6.



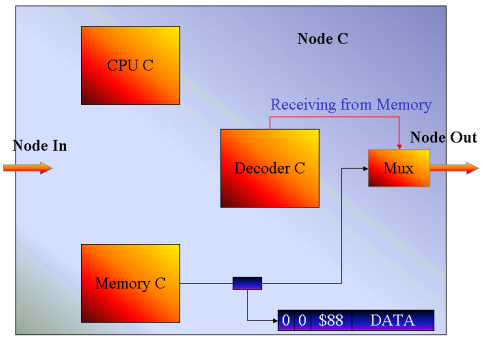
(a) CPU sends read request



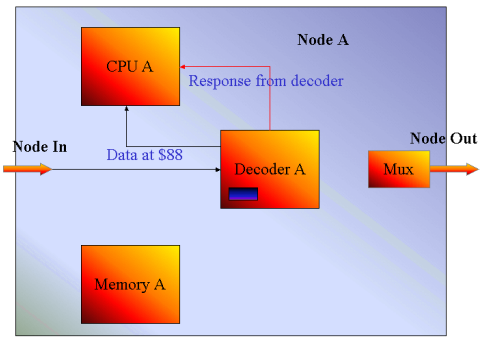
(b) Decoder sends packet out onto the interconnect



(c) Packet arrives at destination Node C and processing begins



(d) Memory sends packet out onto the interconnect



(e) Decoder returns response to CPU

Figure 3.5: Remote Read Transaction

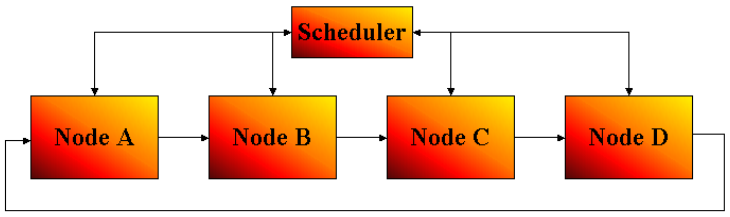


Figure 3.6: The Scheduler

3.3.2 Evaluation of the Initial Implementation

In VHDL, *port-mapping* is a function used to connect entities together. This was done to instantiate each entity into each of the 4 nodes. Each of the four nodes were then instantiated into the final topology as shown in Figure 3.6. Upon completion, each entity was thoroughly tested, exploring all corner edges (all possible combinations of the inputs)[4]. In order to test this implementation fully, each CPU randomly performed transactions to each of the 256 memory locations in turn. This was done by modifying the Scheduler code slightly. Once the test results were all correct, the Scheduler functionality was returned to normal, selecting each node in turn to perform a transaction. Each nodes CPU entity was given a different seed to start off with.

Once the design had passed this test final test, the final stage of project was addressed. The next section describes the steps taken in implementing the final design.

3.4 Final Implementation

This section details how the SCI Cache Coherence Protocols were simulated in VHDL. Figure 3.7 illustrates the final internal node structure. The design of each of these entities will now be discussed in detail, beginning with the Cache entity.

[4]Testing is discussed in detail in the next chapter.

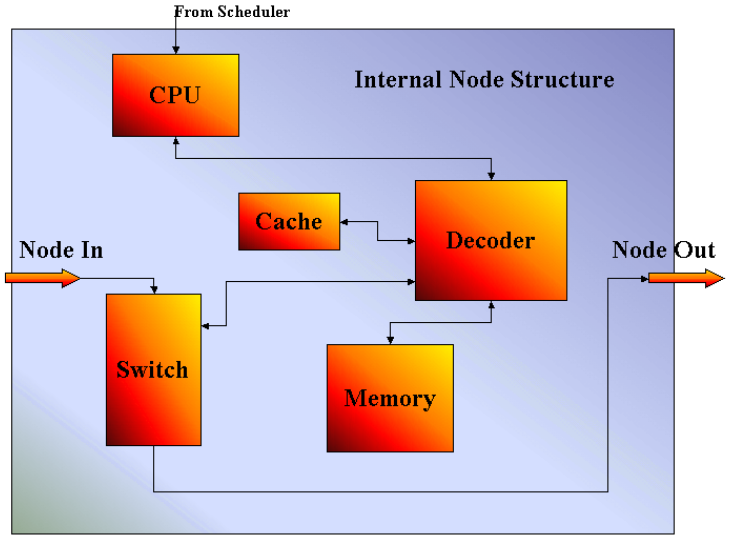


Figure 3.7: Internal Node Structure

3.4.1 The Cache Entity

A cache size of 128 elements and implementation of a least recently used cache replacement policy were used. Along with the data and corresponding addresses, additional directory information fields are associated with each cache line. These include :

1. A cache state field
2. A forward pointer field (to the next node in the list)
3. A backward pointer field (to the previous node in the list)
4. An l.r.u indicator field

The l.r.u indicator field was used as part of the least recently used cache replacement functionality. These bits indicate how old the cache line is. On a read or write transaction to a cache line X, these bits are cleared (i.e. set to 0). Once a transaction is performed in the cache, all cache lines not directly involved increment their indicator bits by 1. Once the cache is full, the line with the oldest indicator value is chosen to be replaced on the next write.

The cache block layout is illustrated in Figure 3.8. The state field indicates one of the previously mentioned cache states:

1. Unused = 000
2. Head Of Empty list (**H.O.E.L**) = 001
3. Head Of List (**H.O.L**) = 010
4. Regular List Entry (**R.L.E**) = 011
5. Tail List Entry (**T.L.E**) = 100

Two bits are needed for each of the pointer fields, which when used, contain the address of one of the four nodes in the system. The indicator field (10 bits), the address field (8 bits) and the data field (32 bits) make up the rest of the cache line block.

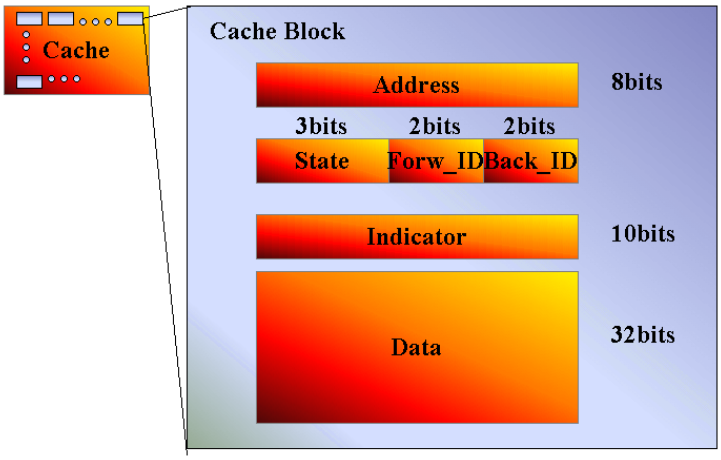


Figure 3.8: Cache Block layout

Cache Line Replacement

After a cache transaction takes place, the indicator field of each block gets incremented. The oldest cache line is also selected. In the case of a tie, the cache line with the smallest index value in the array is chosen. If the cache is full, this l.r.u

block will be replaced on the next cache write transaction. The cache coherence protocol needs to take action before this next "write" takes place, performing one of the two possible actions:

1. If the cache line is in the H.O.E.L state, the block needs to be flushed to memory.

2. If the cache line is in any other state, the shared list needs to be updated.

If the cache wishes to replace an existing block, it will assert a *flush* control signal to the Decoder. The Decoder will then take action and maintain coherency[5].

Cache Functionality

The communication packet structure between the Cache and Decoder entities is illustrated in Figure 3.9. The only difference between the two packet structures is in the most significant bit (m.s.b) field of each. When the Decoder sends a packet to the Cache entity, the m.s.b indicates whether a cache read or cache write is to be performed. When the Cache sends a packet to the Decoder, the m.s.b indicates whether a cache hit or cache miss took place.

(a) Decoder to Cache Packet Format

(b) Cache to Decoder packet format

Figure 3.9: Decoder-Cache packet structure

The Decoder controls the Cache entity using a two bit control signal, selecting one of four possible functions.

1. **Read or write**

[5]This action is explained later in the Decoder Entity section

The decoder selects this function when performing a cache read or write transaction. The m.s.b of the packet sent from the Decoder, indicates whether a read or write is to take place.

- In order to perform a read, the cache indexes through its array searching for the corresponding address. If successfully found, the indicator bits within the block are cleared (i.e. set to 0). The data, cache state and pointers are returned to the Decoder with a cache hit packet (i.e. m.s.b = 1).
- In order to perform a write transaction, the cache will either write to the next free index in the array (when the cache is empty or partially full), or write over the least recently used line (when the cache is full). This least recently used line is picked by the update loop, discussed in the Cache Replacement section. Using the packet fields sent by the Decoder, the cache block fields are set. The 10-bit indicator field value is set to 0.

2. **Update an existing cache line**

   This function updates an existing cache line. The decoder uses this for reasons which will become apparent in the Decoder Entity section. The cache line is updated and the indicator bits are cleared.

3. **Flush a cache line**

   This is part of the cache flushing process mentioned earlier. In response to this control signal, the cache sends the required data fields of the corresponding cache line to the Decoder.

4. **Wipe a cache line**

   The Decoder will use this function when it is required to wipe/purge a cache line. The cache sets all bits of this line to 0.

This sums up the functionality of the Cache Entity. The next section will discuss the Switch entity.

3.4.2 The Switch Entity

This entity acts as a node interface to the interconnect (like in SCI). This interface is primarily responsible for two things:

1. Examining the destination field of incoming packets.

   If a packet is not destined for the current node, the switch forwards it to the output link, bypassing all other entities in the node. Otherwise the packet is sent to the Decoder entity for processing (in SCI the packet will be sent to the node application layer).

2. Forwarding packets out onto the interconnect from the Decoder.

   The Decoder sends packets out onto the interconnect via this Switch interface. The interconnect packet structure can be seen in Figure 3.10. The transaction# field will be explained in detail in the Decoder entity section.

Figure 3.10: Interconnect Packets

Extra functionality was added to the Switch entity initially, which enabled all or some nodes in the system to perform transactions simultaneously. In order to do this, the CPU entity needed to be modified, making it ignore the Schedulers input and therefore send transactions continuously to the Decoder. In this situation, the switch would possibly receive a packet from both sources (i.e. the SCI interconnect and the Decoder entity), therefore requiring it to buffer the Decoder packet(s) until the incoming SCI link becomes idle. The Switch entity could therefore act as the combined input and output buffers used in the SCI node model. It was later decided that this *simultaneous* transaction processing would not be used. The main reason for this decision was that during the testing phase, the test-bench waveform was quite difficult to follow and understand, due to the amount of transactions being executed. It therefore restricted the amount of control that could be enforced on

the design. Implementing this was never part of the project goal, so the CPU code was re-adjusted, allowing the Scheduler to enforce the queue system once more.

In order for the switch to be able to route packets correctly, it requires knowledge of its node address. One way of implementing this in VHDL is to make a different switch entity for each node, setting a `node_id` variable to a unique value. This method was used for the initial implementation entities (Memory, Decoder and CPU). A more efficient approach was taken for the final implementation, using VHDL *generic* functionality [6]. In the entity declaration part of the VHDL module, a *generic map* can be used to pass in a variable upon instantiation. The CPU, Switch and Decoder entities [7]required the use of this generic mapping. The functionality of each of these entities requires knowledge of the node in the interconnect of which it is a part of. The Decoder entity also requires knowledge of which part of the distributed address space it has access to. The switch entity only requires knowledge of its node number, in order to forward the packets to the right output port. The code extract below shows the generic map function being used. A default value is specified in case a generic instantiation is omitted in the higher level entity. A value of 0 was chosen, which is Node A's integer address value. In VHDL, the **std_logic** type is used for bit values. It was therefore necessary to convert this integer value to std_logic type[8]. The integer value was converted to `std_logic` type using the `to_unsigned` function.

```
1  entity Switch is
2          Generic(n_id : integer := 0);--default = 0;
3          Port(   Clk,Reset:in std_logic;
4                  SCI_in,deco_in : in std_logic_vector(47 downto
                       0);
5                  SCI_out,to_deco : out std_logic_vector(47 downto
                        0));
6  end Switch;
7  architecture Behavioral of Switch is
8
9  begin
10
11 process(.....)
12
```

[6]Ross Brennan, a PhD student in Trinity College, was responsible for the suggestion of this approach.

[7]The CPU and Decoder entities are detailed in the next two sections

[8]As 4 nodes were used in this design, the integer range is 0-3, requiring two bits

```
13 variable our_node:std_logic_vector(1 downto 0):=(others=>'0');
14 --the node identifier variable
15 ............
16 ............
17 ............
18
19 begin
20
21 if(Reset='1')then
22          ..................
23          ..................
24          our_node:= std_logic_vector(to_unsigned(n_id,2));
25          --assign the generic value to the our_node variable
26          ..................
27          ..................
```

When each Node entity (A, B, C and D) instantiates the generic Switch entity, it passes in an integer value as the node identifier. The following code is used to pass in the node id value for node A:

```
GENERIC MAP(n_id=>0)
```

This sums up the switch entity functionality. In this final implementation the Local Memory module was updated for reasons which will now be discussed.

3.4.3 The Local Memory Entity

In the initial implementation, the Decoder entity passed a global address to the memory module, which then subtracted an *offset* before accessing the memory array. A more optimal way of implementing this is giving the Decoder entity the responsibility to perform the subtraction of this offset. With this approach, the same entity can be instantiated into each node. Along with the data block, the associated directory information is also stored in memory. According to the SCI Cache Coherence Protocol, memory state and forward pointer fields make up this fixed overhead, maintaining the distributed directory structure. Figure 3.11 illustrates the block layout for each memory module. The *State* field indicates whether the block is shared or not, requiring a single bit. An extra bit is needed to implement *idle signal* functionality. This will be discussed in a later section.

The memory entity performs one of three different actions upon request from the Decoder: a read, a write or a flush. A 2-bit control input from the Decoder entity

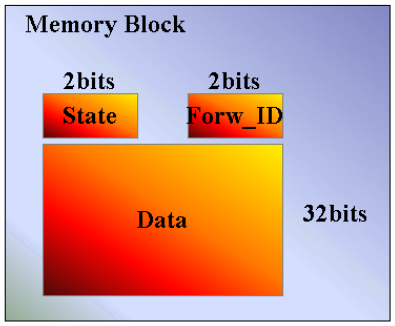
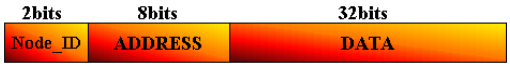


Figure 3.11: Memory Block Layout

is used. The memory and Decoder communicate using the packet structures shown in Figure 3.12. In (a), the *Node_ID* field is used to identify the node performing the read transaction (i.e. which CPU is reading the data).



(a) Decoder to Memory Packet Format

(b) Memory to Decoder packet format

Figure 3.12: Incoming and Outgoing Packet Formats for the Local Memory Entity

1. **A Memory Read Transaction**

   On a read, the Memory entity will retrieve a data block, using the address field of the received packet as the index to the memory array. The state and pointer information, along with the 32-bit data value are returned to the Decoder. The Decoder, discussed in the next section, implements the SCI cache coherence protocol, maintaining coherency. The *Node_ID* field indicates which node is performing the read transaction. The *Forw_ID* field of the block is updated

with this address value, and the memory state is set to the shared state (if not already). This completes the updating of the directory information.

2. **A Memory Write Transaction**

   On a write, Memory will return a packet to the Decoder, giving it the current directory information. It then updates its directory information.

3. **A Flush Transaction**

   The flush transaction is used when the H.O.E.L needs to overwrite a cache line. When the memory module receives the incoming packet, the 32-bit data is flushed to its location, returning the (possibly updated) data. Memory then sets the state field to *not shared* and clears the forward pointer. This clearing of the forward pointer is not essential, as the Decoder will only ever look at the forward pointer field if the memory state is shared.

This sums up the functionality of the Local Memory Entity. The initial implementation of the Decoder entity was quite straightforward as there was no cache involved, and therefore no cache coherence protocol.

3.4.4 The Decoder Entity

As the Decoder entity implements the SCI Cache Coherence Protocol, it proved the most difficult to design, code and test. Introducing the caches and the SCI cache coherence protocol required the design of the Decoder to go back to the drawing board! Upon receipt of a CPU request transaction, the Decoder must process the transaction, abiding by the SCI Cache Coherence Protocol rules. As mentioned in the Switch Entity section, the Decoder makes use of the generic map functionality. So once instantiated with a node address, each Decoder can identify which node it belongs to. As specified in the previous section, the Decoder entity must subtract an offset from the address it sends to Memory. As each node has a unique integer address which gets assigned upon instantiation, this value can be used to calculate the address offset within the node. This is achieved using the following line of code:

```
1 offset:=std_logic_vector(to_unsigned((n_id*64),8));
```

When carried out, this assigns an offset of 0 to node A, an offset of 64 to node B, an offset of 128 to node C and an offset of 192 to node D. In the initial implementation, if node A sends a packet over the interconnect to node B, it was performing one of the following two transactions:

1. Sending a read or write request to node B's address space.
2. Returning data from its address space / confirming a write took place to its address space.

As mentioned in Chapter 2, the two major components of a directory-based cache coherence protocol are:

1. The directory organisation.
2. The set of message types and message actions.

In the SCI standard, the transactions used are based on the split-transaction method of message passing and the different cache and memory states. In order to apply the SCI cache coherence protocol to this design, 16 transactions were designed. These transactions are based on different coherence actions used in the SCI protocol. Each of these transactions are listed in Table 3.1. and will be discussed in a later section. The steps the decoder takes when processing a read or write transaction will first be outlined. Until the CPU asserts a control signal to the Decoder (indicating a transaction is waiting to be processed), the Decoder remains in idle mode, dealing with request packets from other nodes in the interconnect.

Read transaction

Once the CPU asserts a control signal to the Decoder entity, the Decoder reads the incoming transaction [9]. As in the initial implementation, a read transaction is

[9]The packet structure for the communication between the CPU and decoder did not change for the final implementation. The packet structure was illustrated on page 35.

Transaction#	Description
0	Head Update
1	Remote Read to Memory
2	Remote Write to Memory
3	Return of Data to Source/Confirmation of Write
4	Read Request to H.O.L
5	Write Request to H.O.L
6	Return of Data from Old Head to New Head
7	Purge of List Complete: Permission to Write Cache Line
8	Purge R.O.L
9	Update Backward Pointer
10	Update Forward Pointer
11	Update Memory Pointer & (Optionally) Send Purge Transaction To Rest of List
12	Pointer Update Confirmation
13	Purge Completion
14	Tale Update
15	Flush to Memory

Table 3.1: Internodal Transactions

denoted by the most significant bit being 0. The Decoder will first perform a cache read using this address [10]. This is done by setting the cache control signal to indicate that a read or write transaction is taking place [11]. The cache replies indicating a cache hit or miss. If a cache hit occurs, the Decoder will send the retrieved data to its CPU. If however a cache miss occurs, the Decoder then calculates the location of the address in global memory, and performs one of the following actions:

- If the address is located in the current nodes address space [12], a read from local memory is performed[13]. The returning packet from memory indicates whether the block is shared or not. If not shared, the cache can be updated and the data can be returned to the CPU, completing the read transaction. If shared however, the `forw_id` field will point to the node at the head of the list. A packet is then sent out onto the interconnect using transaction#4. This

[10]The Decoder and Cache communication packet structure was illustrated on page 39.
[11]This was discussed in the Cache Entity section. There are four possible functions. A read/write is one of these.
[12]The current node is the node whose CPU is performing a transaction.
[13]The Decoder and Memory communication packet structure was illustrated on page 44.

packet will be addressed to the current head, *node X* for example. Node X will then process this transaction (processing steps are explained later), eventually resulting in the return of the data block to the current node. The current node will then update its cache (becoming H.O.L) and return the data to the CPU, completing the transaction.

- If the address is located in another nodes address space, *node X* for example, a packet is sent out to node X using transaction#1. Node X will process this transaction, eventually resulting in the return of the data block to the current node. The current node can then update its cache and return the data to the CPU, completing the transaction.

Write Transaction

Conversely, a write transaction is denoted by the most significant bit of the packet sent from the CPU being set. The Decoder proceeds the same way as the read transaction, checking its cache for the address (cache read). The cache replies with the return packet.

Write Transaction : Cache Hit

If a *cache hit* occurs (i.e. m.s.b = 1), the cache state field is examined. Depending on which state the cache line is in, different steps are taken.

- **H.O.E.L**

  If the current node is head of an empty list, it already has exclusive ownership of the block, and the Decoder can therefore update its cached copy (setting the cache control line to *update*), using the data sent from the CPU. The Decoder can then return confirmation to the CPU, signifying that the transaction is complete.

- **H.O.L**

If the current node is head of a shared list, it needs to obtain an exclusive copy of the block before it can be given write permission. The only way to do this is to delete the rest of the list. Using transaction#8 (purge rest of list), the Decoder sends a packet to the next node in the list. Each member of the list will wipe its cache line entry for the address and forward the purge transaction to the next node in line. When the tail list entry receives the purge packet, it will send a confirmation packet back to the head of the list indicating that the purge has been completed (transaction#7). Once the current node receives this confirmation, it can then update its cache line and send confirmation to the CPU, indicating that the transaction is complete.

- **R.L.E**

  If the current node is a regular list entry, it will need to:

  1. pop itself out of the list
  2. make itself the new head of the list
  3. send a purge transaction to the rest of the list

  In order to pop itself out of the list, it will need to update the backward pointer of the node in front of it (using transaction#9), and update the backward pointer of the node in front of it (using transaction#10). Each of these nodes will send back a pointer update confirmation packet (transaction#12). Once these updates have been confirmed, the current node can make itself head of the list and delete all other entries. This is done by sending transaction#11 to the local memory or source of the address (unless of course the current node is the source of the address. In which case a memory write transaction is used, followed by a purge transaction to the head of the list). As in the previous case, the T.L.E will eventually send a confirmation packet to the current node, indicating that the purge has been completed. The current node's Decoder can then update its cache line and send confirmation to the CPU.

- **T.L.E**

Similarly, if the current node is a tail list entry, it will need to:

1. pop itself out of the list
2. make itself head of the list
3. send a purge transaction to the rest of the list

In order to pop itself out of the list, the tail list entry need only send a tail update transaction (transaction#14) to the node before it. This tells the receiving node that it needs to update its cache state to T.L.E (if its current state is R.L.E), or H.O.E.L (if its current state is a H.O.L). Once confirmation of this update has been received by the current node, it can update memory and purge the shared list (explained previously). The current node will eventually receive confirmation from the T.L.E that the list has been successfully purged, giving it permission to update its cache and send confirmation to the CPU.

Write Transaction : Cache Miss

If a cache miss occurs, the Decoder first calculates the location/source of the address.

- If the address is a located locally, the Decoder will perform a read transaction to its memory module. Once memory returns the necessary information, the state field is first checked.

  1. If the memory state is not shared, the Decoder performs a cache write, initialising itself as H.O.E.L. Confirmation is then sent to the CPU.
  2. If the memory state is shared however, the forward pointer field is examined. Using this forward pointer, the current node will purge the list. Once this has been confirmed (by the T.L.E), the current node can write to its cache and send confirmation to its CPU.

- If the address is a remotely located, the current node will send a write request (transaction#2) to the remote location. Confirmation will eventually be sent

back to the current node, giving it exclusive ownership of the block. Once this occurs the decoder writes to its cache and sends confirmation to the CPU.

Cache Flush Transaction

As mentioned in the Cache Entity section, the cache asserts a flush signal to the Decoder entity when it needs to flush a cache line. When idle, the Decoder deals with this request by setting the cache control signal to *flush* [14]. The next steps are carried out according to the cache state of the line.

- **H.O.E.L**

  If the current node is head of an empty list, it will need to flush the possibly dirty data to memory. If the memory location is local, the Decoder performs a flush transaction to its Local Memory module. Otherwise the Decoder will perform a remote flush, using the flush transaction (transaction#15).

- **H.O.L**

  If the current node is head of a shared list it needs to send a head update transaction (transaction#0) to the next node in the shared list. This node will update its cache state to either

  - H.O.L (if it is a R.L.E)
  - H.O.E.L (if it is a T.L.E)

  Once this is completed (i.e. when a *pointer update confirmation* packet is sent back to the current node using transaction#12), the current node will need to update memory. If the address is located locally, the Decoder will perform a write to memory. Otherwise transaction#11 is sent to the remote node with a slight variation. If a node receives a packet with transaction#11, it examines the data field. If the data field has all bits set, then it is treated as a memory update transaction only (the purge command is not sent to the list). So the

[14] This was discussed in the cache entity section

current node sets all bits to 1 in the data field of the packet, sending it to the remote location, completing the update.

- **R.L.E**

  If the current node is a regular list entry it will need to *pop* itself out of the list. It does so by updating its surrounding neighbours pointer values, which was explained in the previous Cache Write section.

- **T.L.E**

  Finally, if the current node is a tail list entry, it will need to send a tail exchange transaction to the previous list entry. This previous list entry has two ways of dealing with this, depending on its current cache state (once again, this was explained in the previous Cache Write section).

This outlines the Decoder's processing steps for a cache flush transaction.

Interconnect Transactions

The Decoder entities use 16 different transactions to maintain coherency in the distributed global address space. A detailed description of each will now be given. [15]

Transaction#0: Head Update

When a H.O.L entry is processing a flush transaction, the Decoder will update the cache state of the next node in line, node X for example. This is done using a *Head Update* transaction. Upon receipt of this transaction, node X will perform a cache read using the address from the received packet. A cache hit will take place (as expected), and the Decoder will examine the cache state field, acting accordingly.[16]

1. If the cache state indicates that the node is a R.L.E, the Decoder will update the cache to the H.O.L state.

[15] Chapter 4 will more clearly outline the use of each transaction, as simulation examples are given.

[16] There are only two possible states the cache could be in.

2. If the cache state indicates that the node is a T.L.E, the Decoder will update the cache to the H.O.E.L state.

Node X will then send a packet confirming this update to the original node (transaction#12).

Transaction#1: Remote Read to Memory

When a node wishes to read from a remote memory address not contained in its cache, it will send a *Remote Read* to the node which *owns* this address space, node X for example. Node X will first perform a memory read transaction. Memory will return a packet in the format discussed in the Local Memory Entity section. If the memory state is not shared, node X's Decoder will return the data to the requesting node using transaction#3. If the memory state is shared however, further processing is required:

The address of the forward pointer is examined. If node X is head of the (possibly empty) list, a local cache read is performed resulting in a cache hit (as expected). The state field is examined for two different possibilities.

1. **H.O.E.L**

   If node X is head of an empty list, it will update its state to the T.L.E state.

2. **H.O.L**

   If the node X is head of a list containing more than one node, it will update its state to the R.L.E state.

Once this is complete, node X's Decoder will return the data to the requesting node using transaction#6. If node X's local memory pointer points to a different node in the interconnect however, a read request is sent to the head node using transaction#4.

Transaction#2: Remote Write to Memory

When a node wishes to write to a remote memory address not contained in its cache, it will send a *Remote Write* to the node which *owns* this address space, node X for example. Node X will first perform a write to memory and will receive the old state and pointer information in return. The Decoder will examine the state field. If the memory state is not shared, node X's Decoder will return confirmation to the requesting node using transaction#3. Alternatively, if the memory state is shared, further processing is required.

The address of the forward pointer is examined next. If it is node Xs address that is head of the (possibly empty) list, a local cache read is performed resulting in a cache hit (as expected). The state field is examined for two different possibilities.

1. **H.O.E.L**

   If node X is head of an empty list, it will wipe its cache line and send confirmation back to the requesting node using transaction#7.

2. **H.O.L**

   If the node X is head of a list of nodes, it will wipe its cache line and purge the rest of the list (using transaction#8). In order for the T.L.E to be able to send a confirmation packet to the requesting node, node X will insert the requesting nodes ID in the source field.

On the other hand, if the node X's local memory pointer points to a different node in the interconnect, a write request is sent to the head node using transaction#5.

Transaction#3: Return of Data to Source/Confirmation of Write

The situation where this transaction is used has been outlined previously. Upon receipt of this type of packet, the cache is updated and the CPU is either sent confirmation that the write has taken place (if it was a write transaction originally), or the requested data is returned (if it was a read transaction originally).

Transaction#4: Read Request to H.O.L

A packet, using this transaction type, is sent to the head list entry. The receiving node will first perform a cache read, checking which one of two possible states its cache line is in.

1. **H.O.E.L**

   If the cache line is head of an empty list, the Decoder will update the cache state to the T.L.E state.

2. **H.O.L**

   If the cache line is head of a list, the Decoder will update the cache state to the R.L.E state.

Once the cache update is complete, the Decoder in the receiving node will send the corresponding cached data to the requesting node using transaction#6. The requesting node will then make itself head of the list.

Transaction#5: Write Request to H.O.L

A packet, using this transaction type, is sent to the head list entry. Like in transaction#4, the cache state of the receiving node will need to be checked for two possibilities.

1. **H.O.E.L**

   If the cache line is head of an empty list, the Decoder will wipe its cached copy and then return confirmation to the requesting node using transaction#7, indicating that the list has been purged.

2. **H.O.L**

   If the cache line is head of a list, the Decoder will wipe its cached copy and purge the rest of the list using transaction#8. The address of the requesting node will be inserted into the source field of the outgoing purge packet, enabling the T.L.E to send confirmation to it, once the deletion of the list has been completed.

Transaction#6: Return of Data from Old Head to New Head

Upon receipt of a packet with this transaction type, the receiving Decoder will write to its cache with the data received, establishing itself as head of the list. The source field of the received packet is used to identify the node next in line (i.e. used in the forward pointer field of the cache block). The Decoder will then return the data to the waiting CPU, thus completing the transaction.

Transaction#7: Purge of List Complete: Permission to Write Cache Line

Once the Decoder receives this transaction, it will either:

1. Update its cache line (if a cached copy of the address already exists).
2. Perform a cache write, creating a new entry for that particular address.

Once this update has been completed, the Decoder then sends confirmation to the CPU, completing the transaction.

Transaction#8: Purge R.O.L

For this transaction, the Decoder will need to first perform a cache read. As expected, a cache hit will occur, and the cache state field will be examined. Depending on the cache state, the Decoder will act in one of two ways.

1. **R.L.E or H.O.L**

   If the receiving node is a regular list entry or a head of list entry, it will wipe its corresponding cache line and forward the purge transaction to the next node in the list.

2. **T.L.E**

   If the receiving node happens to be a tail list entry, it will wipe its cache line and send confirmation to the requesting node, indicating that the deletion of the shared list is complete. [17] The requesting nodes address is contained in the source field of the packet.

[17]The requesting node in this case is the node which is waiting for permission to write to the cache line.

The above are the only possible cache states that the receiving node could be in. The reason for this is that if a node is head of an empty list, it already has write permission, as it contains an exclusive copy of the data.

Transaction#9: Update Backward Pointer

The receiving node of this transaction, will update its cache backward pointer using the node address encapsulated in the source field of the packet. Once this update has been completed, the receiving node will send confirmation back to the requesting node. In order to allow the receiving node send this confirmation, the requesting node saves its node address in the first two bits of the data field, as the data field is not being used to store any *useful* data for this transaction. Using these two bits, the receiving node will send a pointer update transaction back to the requesting node (transaction#12).

Transaction#10:Update Forward Pointer

This transaction is similar to #9, differing only in the fact that the *forward pointer* is updated. All other functionality is identical.

Transaction#11:Update Memory Pointer & (Optionally) Send Purge Transaction To Rest of List

The receiving node will update its memory state (if its not already shared) and pointer field (using the node address from the source field of the incoming packet). The next step depends on the contents of the data field of the packet. If the 32 bits of the data field are all set to zero, the shared list pointed to by the memory pointer is purged. If the data field bits are all set to 1 however, no purge transaction is needed. This is used when processing a flush transaction. Here, the head list entry is updated, requiring a memory update only. Transaction#12 outlines the case when a purge is needed after a memory update.

Transaction#12:Pointer Update Confirmation

When a regular or tail entry node is performing a write transaction, it will need to send 1 or 2 pointer updates to its neighboring members in order to pop itself out of the shared list, depending on its position in the shared list (R.L.E or T.L.E). Once the update(s) has been completed, a pointer update confirmation packet will be received from the updated node(s).

As mentioned in the above paragraph, the processing of this transaction depends on the cache state of the current node. The only two possible states are R.L.E and T.L.E. [18]

1. **R.L.E**

   If the current node is a R.L.E, it will check which node has sent the pointer update confirmation (either the node at its forward pointer or backward pointer). If the source field indicates the node at its forward pointer, the Decoder will then send a pointer update transaction to the node at its backward pointer. If source field indicates the node at its backward pointer however, the current node is now *popped* out of the shared list. Once this has happened, the Decoder will check the source location of the global address.

   If the address is located locally, memory is updated, and a purge transaction is sent to the list pointed to by memory. If the address is located remotely however, a memory update and purge list transaction (#11) is sent to the remote node. [19]

2. **T.L.E**

   If the current node is a T.L.E, a pointer update transaction could only have come from its only neighbour in the list. Once confirmed, the Decoder checks

[18] If the cache had been in the H.O.L state, a purge transaction would have been sent to delete the rest of the list, giving the current node exclusive write permission. A H.O.E.L entry already has exclusive ownership of the block.

[19] Please note that the data field bits will need to all be set to 0, indicating that the receiving node is to purge the list after updating memory.

the source location of the global address. If the address is located locally, memory is updated and a purge transaction is sent to the list. Otherwise a transaction#11 packet is sent to the remote node which contains the address.

Transaction#13:Purge Completion

This transaction is sent from the tail list entry to indicate that the list has been purged. Once received, a node will have exclusive ownership of the address block. It can then perform a cache update or write and send confirmation back to its CPU, completing the transaction.

Transaction#14:Tale Update

The receiving node will first check its cache state for the corresponding address, as the update depends on the current cache state. If the current cache state is H.O.L, then an update to the H.O.E.L state will be performed. The only other possible cache state in this case is a R.L.E, resulting in an update to the T.L.E state. Once the cache update is complete, a pointer update transaction is sent in reply.

Transaction#15:Flush to Memory

Upon receipt of this transaction, the Decoder simply flushes the data to its local memory entity, asserting the *flush* control signal.

This sums up the functionality of the Decoder entity. The initial implementation of the CPU entity needed slight modification for control and demonstration purposes. These modifications are discussed in the following section.

3.5 CPU Entity

In the initial implementation, a pseudo random number generator was used to generate random addresses within the global memory address space. It was realised however, that in order to clearly demonstrate the SCI Cache Coherence Protocols in action, a number of carefully selected transactions would need to be executed by

each of the CPU's. The generic function outlined previously was used to assign different transactions to each CPU. The transactions are stored in an array structure. With this slight modification, it is possible to use specific transactions which highlight the interesting actions of the SCI cache coherence protocol. The interaction with the Scheduler entity remained the same, and is summarised as follows. Once a node has completed a transaction, it de-asserts its request line to the Scheduler. If the CPU still has transactions left to process (i.e. the index of the array has not come to the last element), it re-asserts the request line.

3.6 Instantiation / Port Mapping Stage

Once all the individual entities were coded and tested (testing is discussed in the next chapter), the next step was to instantiate each of the them into 4 different nodes (A, B, C and D). The internal node structure can be seen in Figure 3.13. The red lines are the control signals used by the different entities.

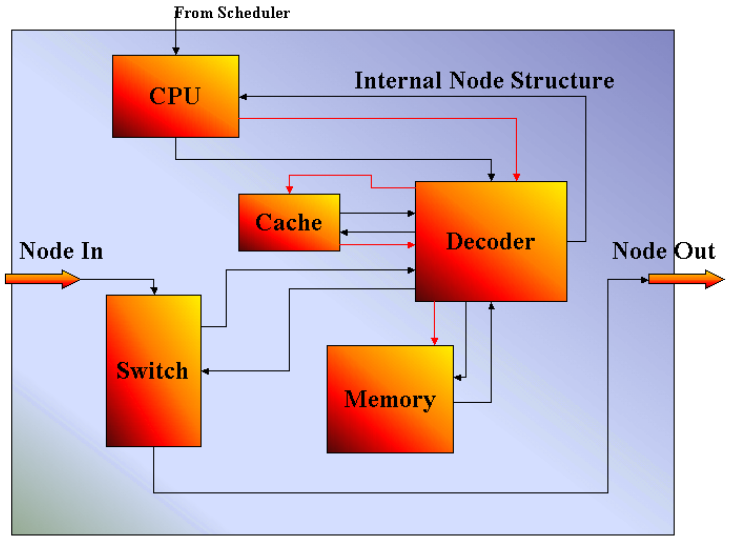


Figure 3.13: Internal Node Structure

For the generic entities, the *node_ID* is passed in as an integer value (0-3), as explained earlier. Test outputs were assigned to all wires within the node. This allowed observation of every internal signal within the node, for testing purposes. Each node was then fully tested and then instantiated into the top-level design,

along with the Scheduler entity. This final topology was illustrated on page 36.

The next chapter will discuss how the design was synthesised and tested.

Chapter 4

Synthesis and Testing

This chapter discusses how synthesis and testing were performed. Test examples are also provided, demonstrating the protocol in action through the use of diagrams and simulation screen-shots.

4.1 Synthesis

Synthesis is the translation of a high-level design description into specified hardware. Logic synthesis translates and optimises the behavioural or RTL code into gate level code, producing what is known as a *net list*. A logic synthesis software tool in VHDL can be thought of as a compiler tool in C. It is then possible to map this net list to hardware. It was a requirement of this project that the design be synthesisable. This meant that the design could potentially be downloaded to hardware. Synthesis was performed using the Xilinx Synthesis Technology (XST) suite of HDL compilation tools. Each entity in this design was compiled and synthesised using this tool, targeting the FPGA[1] device XSA-3S1000FT256-4 [2]. The command used to compile and synthesise each entity targeting this device was:

```
xst -ifn sci_cache.xst
```

The sci_cache.xst file listed each of the entities in hierarchial order. This command

[1]Field Programmable Gate Array

[2]This was the FPGA board available to final year students.

generates a .srp file, which displays the synthesis results. The synthesis results of this design made it impossible to download onto the FPGA device targeted. An FPGA consists of an array of 3 kinds of programmable elements:

1. Configurable Logic Blocks (CLB's)
2. Interconnect Resources
3. Input/Output Blocks

The CLB's consist of LUT's (Look-Up Tables), muxes and flip-flops. According to the device utilization summary in the .srp file, this design uses 468% of the available logic within the device. 434% of the available 4-input LUT's were being used. LUT's are used to store combinational logic. The Decoder entity, due to its complexity, would use most of these L.U.T resources to implement the SCI Cache Coherence Protocols. Each of the 4 nodes instantiates a Decoder, CPU, Cache, Memory and Switch entity. Each of these nodes are instantiated into the overall top-level design. The code written for each of the entities gets replicated 4 times therefore, which explains why the top-level design uses up too many board resources. Downloading the design to the FPGA however, was not considered one of the project goals.

4.2 Testing

All testing was performed using ModelSim-XE II v5.8c[3], which was used through the Xilinx project navigator tool. Testing using ModelSim is carried out using a *test-bench*. Test-benches allow one to manipulate all possible inputs at different *intervals* in the test-bench. For a synchronous design, the rising edge of a clock denotes the beginning of a new interval. Once set up, a test-bench can be simulated using ModelSim. This simulation displays a waveform, which allows one to observe the output value of each signal at every interval. If the design is functionally correct, these values will be expected. In the next chapter, different examples of the SCI

[3]Available from http://model.com

Cache Coherence Protocol in action will be demonstrated. These examples are illustrated through the use of diagrams and test-bench screen-shots. The next section will briefly discuss some of the problems encountered during the testing phase.

4.2.1 Issues

Most of the difficult problems which were encountered, presented themselves during the top-level testing phase. It proved difficult to test the individual entities for more than basic functionality [4]. Errors and problems were relatively easy to debug at this level. Once the top-level model was created however, more complex problems arose which were quite difficult to debug. As most of the complex functionality was implemented by the Decoder entity, this proved the most problematic.

During top-level functional testing, each CPU was programmed with various transactions, testing each piece of functionality in the Decoder entity. To begin with, some small problems were highlighted, which were relatively easy to fix. For example, a minor error encountered was that some of the packets sent out on the interconnect had incorrect fields (data, source, destination, address or transaction#).

The biggest problem faced proved initially quite difficult to identify. The Decoder bases all decisions on the state of its input signals. The Decoder appeared to be performing illegal transactions at different intervals, when in idle mode. The input signals (from the Switch, Cache, Memory), seemed to be affecting decisions made later on in the test-bench. Introducing additional control signals only solved the problem for some cases. The introduction of idle signals on the wires feeding into the Decoder after a certain time-period solved the problem [5]. This required additional functionality in the Switch, Cache and Memory entities. All output bits of the signal sent to the Decoder entity are set high after a certain time period (leaving enough time for the Decoder to retrieve all information needed from the packet).

[4] It was not feasible to cover all combinations of inputs as this would have been too time consuming. It would also be quite difficult to interpret what the correct result should be in each case.

[5] In SCI, message passing protocols use idle symbols on the interconnect lines.

This explains why the *extra bit* was not removed from the Memory entity, as if all bits are set, the Decoder knows it is an idle signal. The Decoder can identify an idle signal from the Switch entity, as the destination and source fields will never both be set to 11 (i.e. Node D). Finally, the Decoder can identify an idle signal from the Cache entity as the cache state 1111 does not exist.

In order to test all *corner-edges*, each node randomly performed read and writes to all global addresses, using the pseudo random number generator from the initial implementation. Once successful, specific examples of the protocol in action were set up for demonstration purposes. These examples are discussed in the next section.

4.3 Test Examples

This section demonstrates, using examples, the VHDL-based SCI Cache Coherence Protocol in action. The examples here were used during project demonstration on the 5th of April 2005. These examples were carefully selected in order to demonstrate the overall functionality of the cache coherence protocol.

The test-bench waveform contains two (single-bit) input signals: clock and reset. Every other signal in the test-bench is a test output. As mentioned earlier, these test outputs are not only used to display the communication between the 4 nodes in the interconnect, but also to display the internal entity communication within each node. Different signal colours have been chosen to differentiate some of the signals and to aid in the explanation of each step. Please note that this project implements a synchronous design, as each entity performs an action on the rising edge of the clock only. The design was implemented this way for demonstration and testing purposes only. An asynchronous design would have proven extremely difficult to test and almost impossible to demonstrate effectively. Constructing realistic CPU, cache and memory processing times was not one of the goals for this project. The CPU in this design, acts as a simple transaction generator. Associated with the following test-bench screen-shots is a timing bar situated below the test signals. This was included to allow accurate reference to each part of the screen-shot.

To begin with, each CPU is given 7 transactions to complete. Once a node has completed its assigned transactions, it does not re-assert its request line, indicating that it does not require another time slot from the Scheduler. This is illustrated in Figure 4.1(a). Each time a node completes a transaction, it de-asserts its request line. If there are still transactions for it left to process however, it will re-assert the request line once the scheduler has given the next inline node the time slot. The light blue lines represent the scheduler allocating a time-slot to each node. These signals are named `goA, goB, goC, goD`. The request signals are coloured dark blue (`rqA, rqB, rqC, rqD`). As can be seen from the screen-shot, once a node completes all 7 transactions, the request line for that node is not re-asserted.

The next set of signals are the interconnect lines (48 bits). Each node has an input wire (`nodeAin, nodeBin`...) and an output wire (`nodeAout, nodeBout`...). These input and output wires are coloured orange and green respectively. Shown in Figure 4.1(b), are the internal test output signals of node A. Highlighted in blue, are the 41-bit communication signals between the Decoder and CPU entities, `cpu_toDecoA` and `deco_toCPUA`. The next two green coloured signals represent the 42-bit Decoder and Memory entity communication signals, `deco_toMemA` and `mem_toDecoA`. These are followed by two yellow signals representing the 48-bit Decoder and Cache entity communication signals, `deco_toCacheA` and `cache_toDecoA`. The next two signals represent the 48-bit communication signals between the Decoder and Switch entities, `deco_toSwitchA` and `switch_toDecoA`. The last four represent control signals within the node (i.e. CPU to Decoder control, Decoder to Cache control...). Excluding the first transaction example by node A, different stages in each transaction will be illustrated in a step-by-step manner.

Each of the 7 transactions processed by each node will now be outlined. In order for the reader to easily follow each step of a transaction, the entity communication packet structures can be found together in Appendix B. The `reset` signal is asserted at the beginning of the test-bench, initialising all variables, instantiating the node identification integer values and displaying idle signals on the communication links. On a `reset`, the scheduler will give the first time slot to node A by default. So node

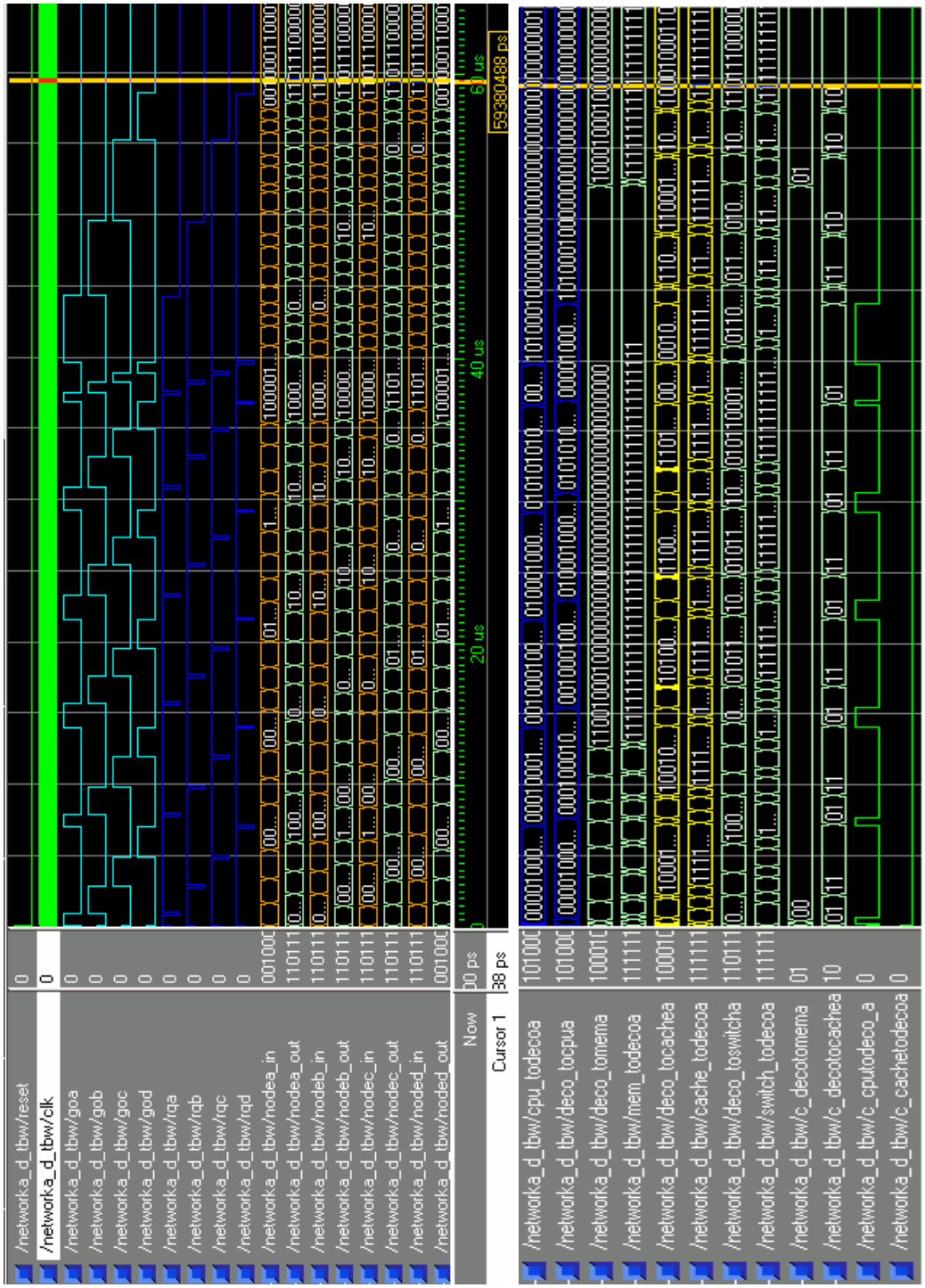
A will get to process its first transaction[6] once the `reset` is de-asserted.

4.3.1 Node A: Transaction#1: Read $11

Node A's first transaction is a read from memory location $11, illustrated by Figure 4.2. This first transaction will be discussed in greater detail than others, in order to explain all signals changes and general behavior of the test-bench screen-shot. At 100ns the reset signal is set low, allowing node A to process its first transaction. On the next rising clock edge, node A's CPU asserts the control signal to the Decoder and sends the first transaction for processing. As detailed by the CPU-Decoder packet structure, the m.s.b set to 0 indicates a read transaction, while the next 8 bits encapsulate the address.[7]. The first action the Decoder takes is a cache read, performed on the next rising edge of the clock. As detailed by the Decoder-Cache packet structure, the m.s.b being set to 0 indicates a read. As part of the reset, the Decoder sets the cache control line to `01` by default, which is the cache read/write state. This does not need to be adjusted therefore. The only other field the Decoder requires is the address field (set to $11). On the next clock cycle, the cache replies with a cache miss packet, denoted by the m.s.b being 0. All other data in the packet is negligible.[8]

The next step the Decoder takes is to calculate the node location of address $11. Address $11 is located in node A's address space, requiring the Decoder to perform a local memory access on the next clock cycle. The memory control signal is set to 00, indicating a read transaction. Memory will return the data along with the associated directory information on the next clock cycle (this can be seen at ∼500ns on the `mem_toDecoA` test output signal). As illustrated by the Memory-Decoder packet structure, the first 8 bits represent the address. This is followed by a two bit state field indicating whether the block is shared or not. In this case, the block is not shared (state field = `00`), indicating that the data field contains the most up-to-date copy of the block.

[6] a reset will set the CPU's index variable in the transaction array to point to the first transaction.
[7] The data field is negligible here as a read transaction being performed.
[8] All caches are cleared on a reset, hence the cache miss.



(a) Scheduler and Interconnect Signals (b) Internal Node Signals

Figure 4.1: High-Level Signals Overview

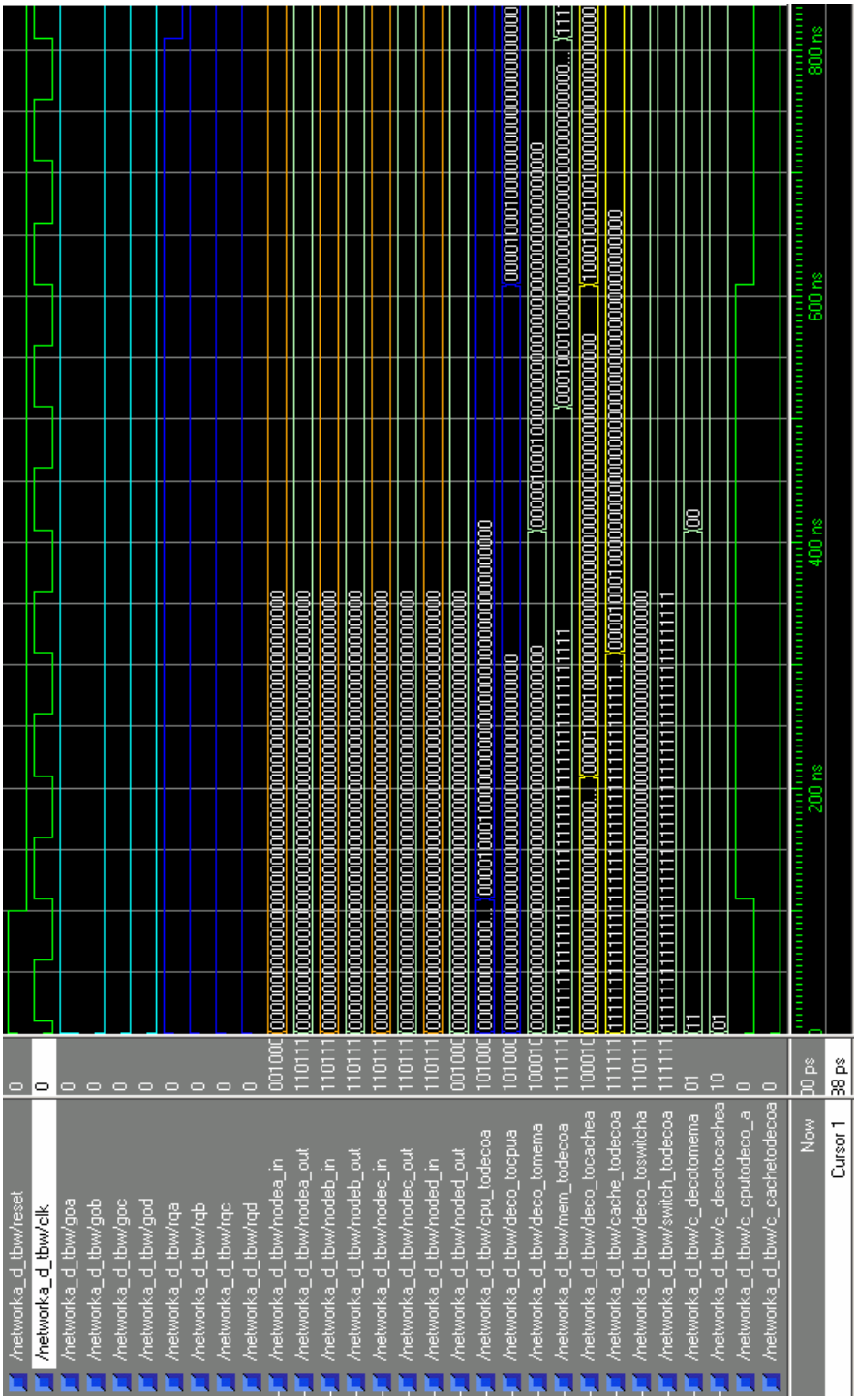


Figure 4.2: Node A: Read $11

The data is written to the cache at ~600ns by the Decoder, signifying the write with the m.s.b set to 1. The cache state field is set to H.O.E.L (0001) and the data field contains the 32-bit value received from memory. The data is also returned to the CPU at this time interval. The CPU will then de-assert its request line temporarily, thus ending the transaction. The current shared list for address $11 is illustrated in Figure 4.3.

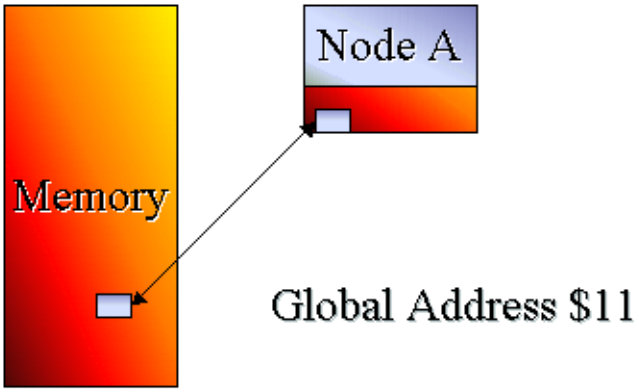


Figure 4.3: $11 Shared List

4.3.2 Node B: Transaction#1: Read $11

Node B is given the next time slot from the Scheduler. This *handover* at the scheduler is illustrated by the screen-shot in Figure 4.4 (a). Node B's CPU will then send its first transaction to its Decoder for processing. A cache read will take place first. Node B's cache (like the others after a reset) is empty, resulting in a cache miss. The Decoder will then calculate the location of address $11, identifying node A as the remote source. A packet is then formulated and sent to the Switch at roughly 1300ns, using the following fields[9]:

1. The *destination* field is set to node A's address (00)
2. The *address* field is set to $11
3. The *transaction#* field is set to trans#1 (0001)

[9]Interconnect packet structure is illustrated along with the others in Appendix B

4. The *source* field is set to node B's address (01)

5. The *data* field is negligible but needs a value, and is therefore set to $00000000.

The communication in node B just discussed is shown in the screen-shot of Figure 4.4(b). The Switch then sends the packet out onto the interconnect at ∼1300ns, which eventually finds node A. This packet forwarding is illustrated in Figure 4.5(a). Each node's switch examines the destination address field. Node A will receive the packet, matching its node address with the packet destination address, and forwarding the packet to its Decoder for processing.

Node A Processes Read Request

The processing steps within node A are illustrated in Figure 4.5(b). The Decoder will receive this packet and perform a memory read transaction to address $11 (∼1900ns). Memory will update its forward pointer to node B's address, returning the old directory information and data (which is negligible in this case, as the data is possibly stale). The state bits indicate that the block is shared. The next two bits identify the node at the head of the shared list (00 = node A). Node A's Decoder therefore performs a cache read. The cache returns with a hit [10], providing the corresponding state and pointer information, along with the data block. Node A's Decoder updates the cache state to the T.L.E state, as there is now 2 entries in the shared list and the backward pointer field is set to node B's address (01). Decoder A will then send a packet to node B returning the data to the new head, using transaction#6, at ∼2200ns.

Node B Receives Data From Old Head

Node B's final processing steps can be seen in Figure 4.6. This packet is delivered to node B and forwarded to its decoder (∼2400ns). Node B writes to its cache, initialising itself as H.O.L., and setting its forward pointer field to node A's address (00). The Decoder then returns the data to its CPU, completing the transaction.

[10] This is expected as we now know that node A is head of the shared list.

The current shared list for address $11 is illustrated in Figure 4.7. The Scheduler will give the next time slot to node C, allowing it to process its first transaction.

4.3.3 Node C: Transaction#1: Read $11

Node C performs the same initial steps as node B. These steps are summarised as follows:

1. Decoder performs a cache read for address $11.
2. A cache miss occurs so the decoder calculates location of address $11.
3. A packet is sent to node A (i.e. the remote location of $11) using transaction#1.

Node A Processes Read Request

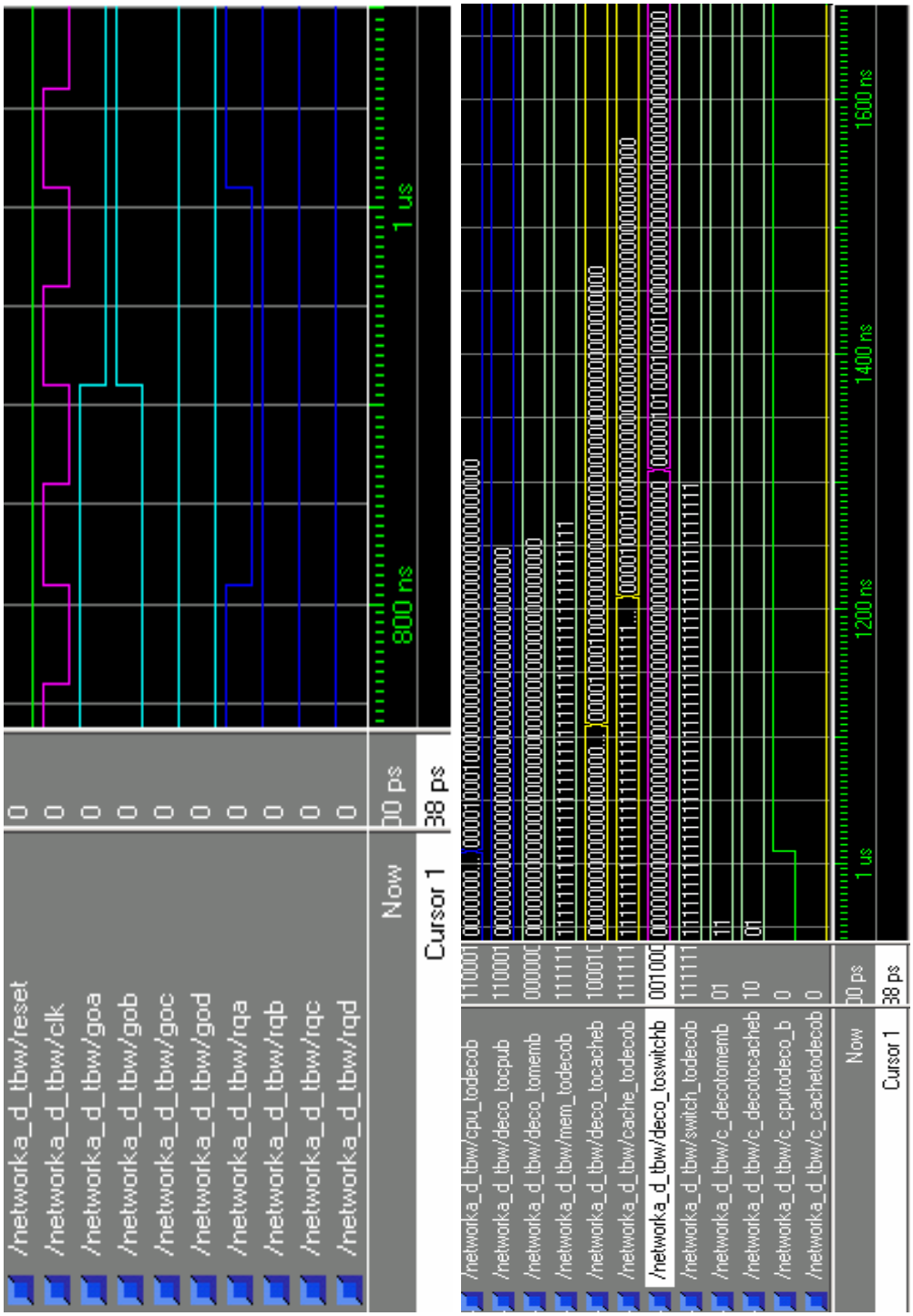
The screen-shot showing node A's processing steps is illustrated in Figure 4.8(a). Decoder A performs a memory read transaction (∼3600ns) upon receipt of the packet. This time the memory's forward pointer indicates node B as the head list entry. Node A will then send a packet to node B using transaction#4 (∼3800ns).

Node B (H.O.L) Processes Read Request

The screen-shot showing node B's processing steps is illustrated in Figure 4.8(b). Decoder B performs a cache read transaction, resulting in a cache hit (as expected). Decoder B will update its cache to the R.L.E state, and will update its backward pointer to node C's address (10). Once the cache line has been updated, the cached data can be forwarded to the new head, node C. The updating of the cache and the packet forwarding to the switch by node B occurs at ∼4300ns.

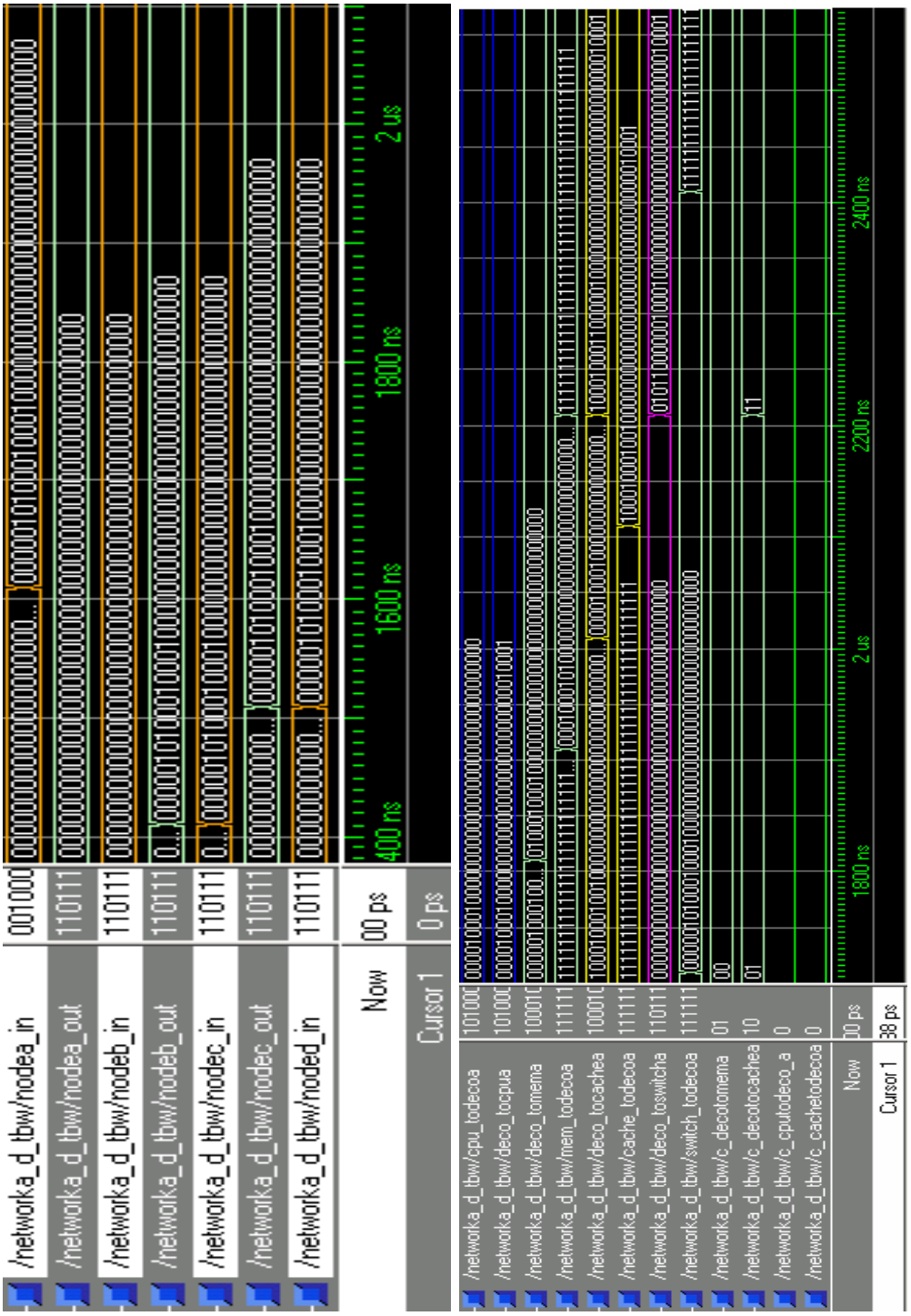
Node C Receives Data From Old Head

This packet is delivered to node C and is forwarded to its decoder at ∼4500ns. Node C performs a cache write, initialising itself to H.O.L. and setting its forward pointer field to node B's address (01). The decoder then returns the data to its CPU,



(a) Scheduler Allocation (b) Node B: Send read request to Node A

Figure 4.4: Node B's first steps



(a) Packet Forwarding to NodeA

(b) Node A: Update cache & return data from cache to Node B

Figure 4.5: Test-Bench Signals Overview

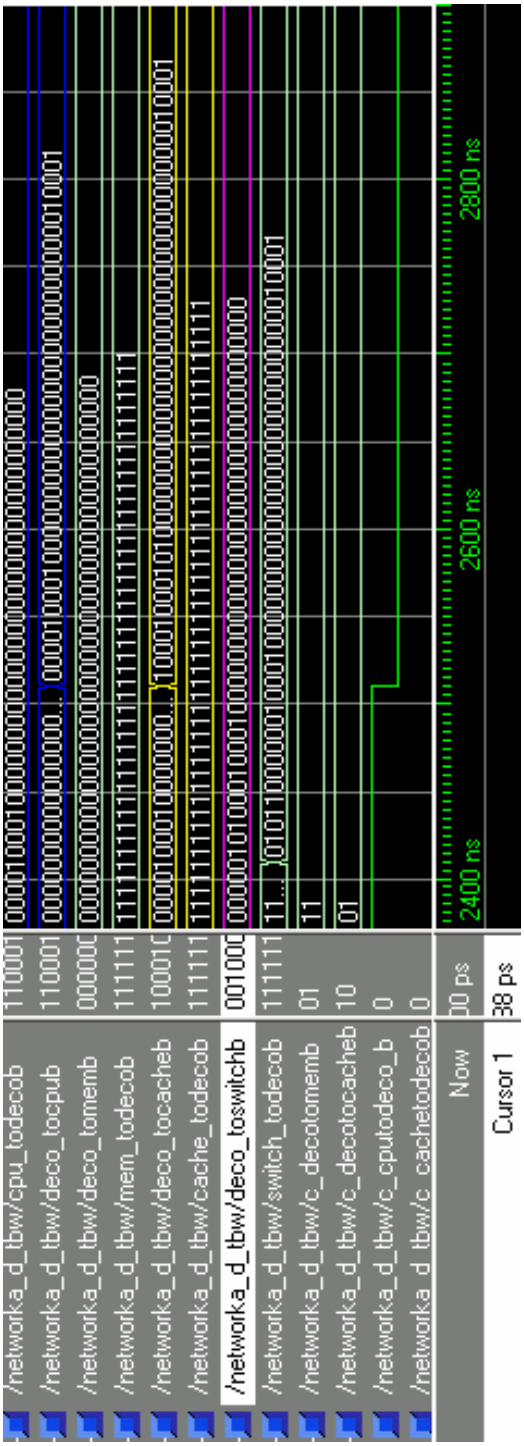


Figure 4.6: Node B updates cache and sends confirmation to CPU

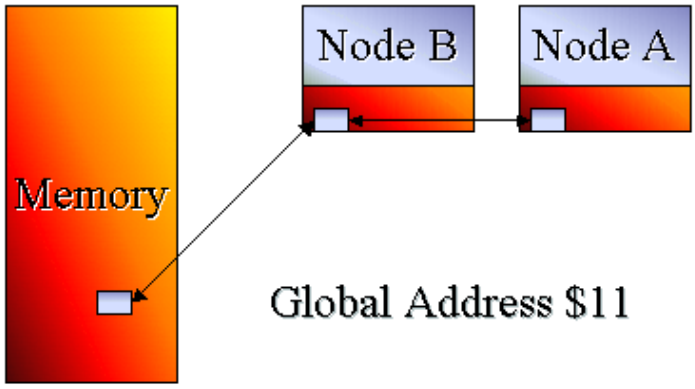


Figure 4.7: $11 Shared List

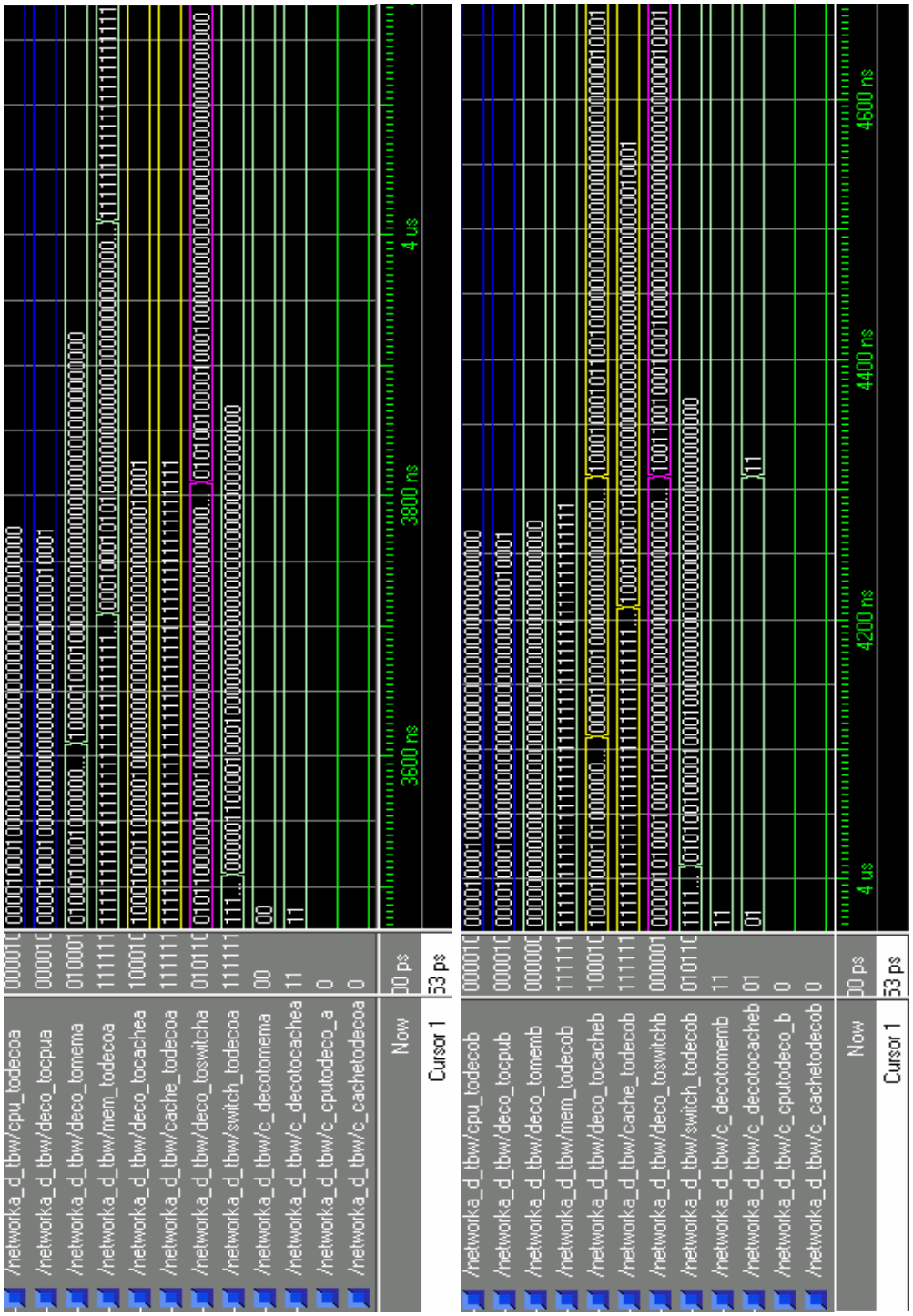
completing the transaction. This final processing step by node C is illustrated in Figure 4.9. The current shared list for address $11 is illustrated in Figure 4.10(a). The Scheduler will give the next time slot to node D which will process its first transaction.

4.3.4 Node D: Transaction#1: Read $11

This transaction will be processed in similar stages to the previous example, the main difference being that the data will be returned from the current head, node C. Node C will degrade itself to a R.L.E, update its backward pointer to point to node D and return the 32-bit data block. Node D will write to its cache, initialising its state to H.O.L and its forward pointer to node C. The current shared list for address $11 is illustrated in Figure 4.10(b).

4.3.5 Transactions 2-6

The first 4 transactions of this test-bench set up a shared list of 4 nodes for the address $11, as depicted in Figure 4.10(b). The steps taken in each case show the protocol functioning correctly. The each CPU then performs further reads from addresses $22, $44, $88 and $AA. This sets up shared lists of the same order for each address. The only difference in these transactions is the source location of the addresses. $22 is located in node A's address space, $44 is located in node B's



(a) Node A: Send read request to Node B(Head)

(b) Node B: Update cache & send data to Node C(New Head)

Figure 4.8: Node C sends read request to Node B

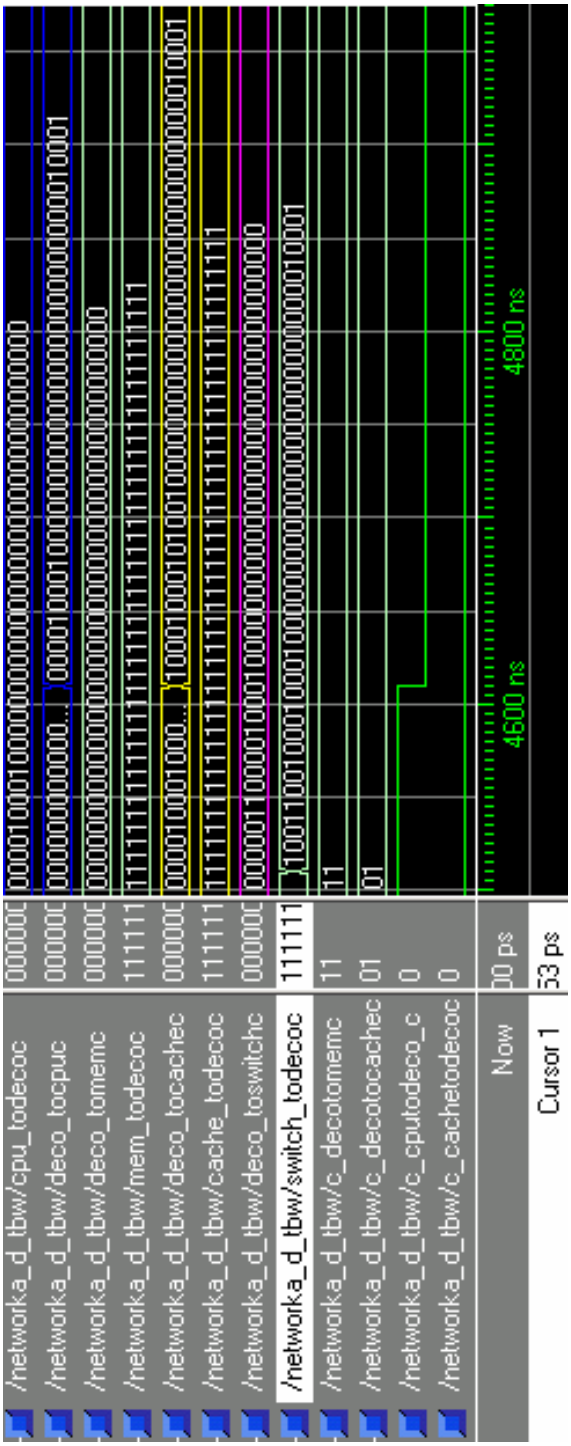
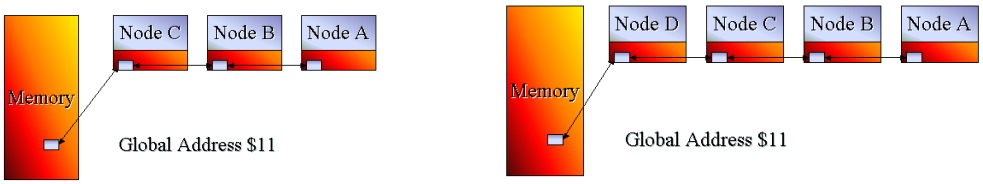


Figure 4.9: Node C updates its cache and returns data to CPU



(a) Shared List after Node C's Read (b) Shared List after Node D's Read

Figure 4.10: Shared List for address $11

address space, $88 is located in node C's address space and $AA is located in node D's address space.

The sixth transaction performed by each node simulates a cache read from one of the cached addresses, $11, $22, $44, $88 or $AA. This results in a cache hit.

Node A: Cache Hit $11

Figure 4.11(a) shows the screen-shot where node A performs a read of address $11. As can be seen from the test-bench, the Decoder will first check its cache for the address at ∼37us. A cache hit occurs and the Decoder can return the data to the CPU immediately, improving performance. Examining the return packet from the cache at 37,100ns, we notice that the cache is (correctly) in the T.L.E state (cache state field = 100), and the backward pointer field contains node B's address (01). The forward pointer field here is negligible as we are dealing with a tail list entry.

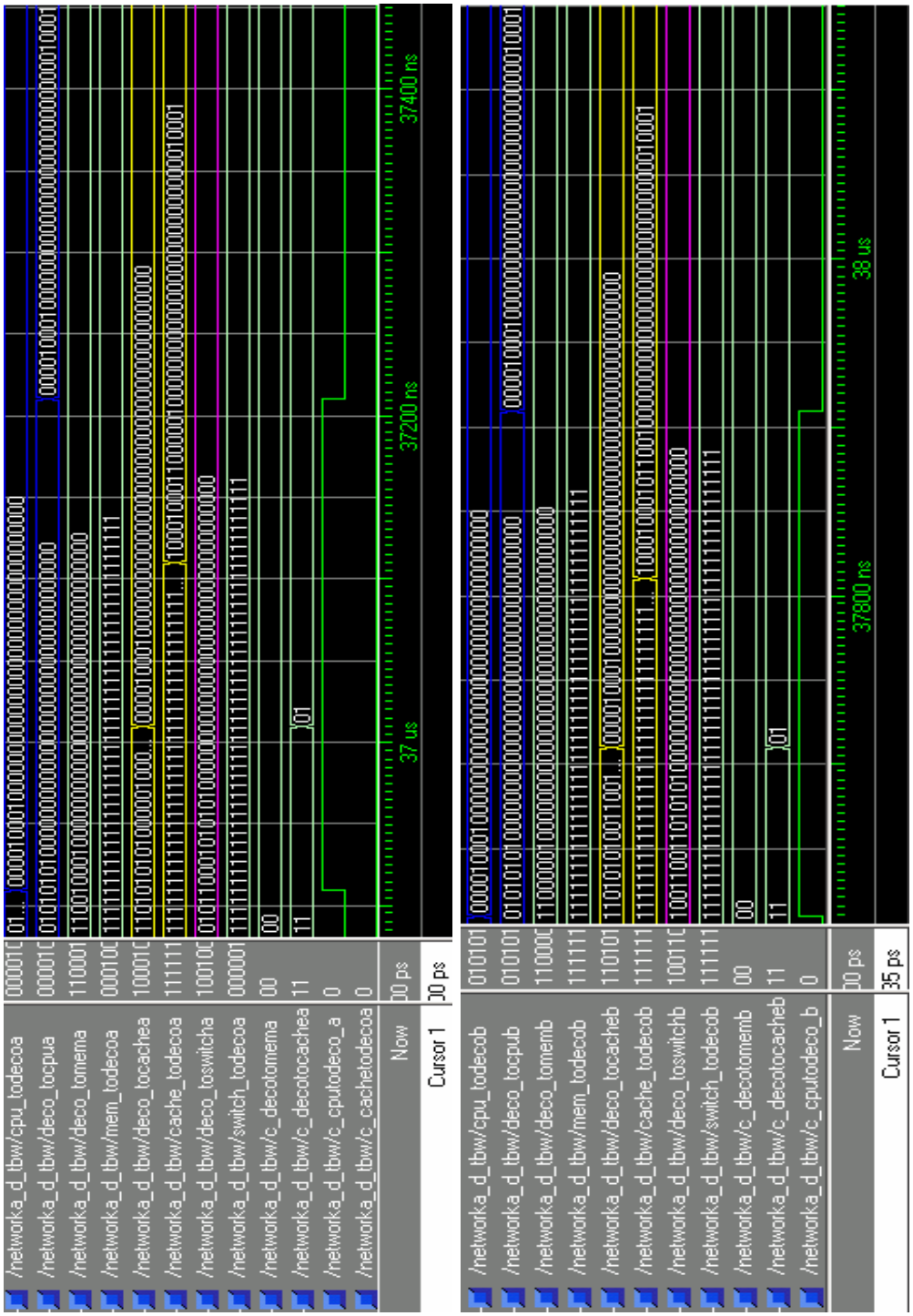
Node B & C: Cache Hit $11

Figure 4.11(b) shows the screen-shot where node B performs a read of address $11. Examining the return packet from the cache at ∼37,800ns, we can see that the cache is (correctly) in the R.L.E state (cache state field = 011), the forward pointer field contains node A's address (00) and the backward pointer field contains node C's address (10). Node C will have the same cache state as node B, R.L.E, but will have a forward pointer value set to node B's address and a backward pointer value set to node D's address.

Node D: Cache Hit $22

Figure 4.12 shows the screen-shot of the test-bench where node D performs its second read of address $22. Examining the return packet from the cache at ∼39,200ns, we can see that the cache is (correctly) in the H.O.L state (cache state field = 010) and the forward pointer field contains node C's address (10). The backward pointer field is negligible here, as node D is the H.O.L entry, which means that its backward pointer field is to memory.

The final transaction each node performs is a write. Each node will now perform a write transaction to one of the addresses previously mentioned. This will require the protocol to act differently in each case, depending on the current nodes cache state. The following transactions will illustrate the efficient use of nearly all of the 16 transaction types.



(a) Node A: Cache Hit : T.L.E

(b) Node B: Cache Hit : R.L.E

Figure 4.11: Cache Hits on $11

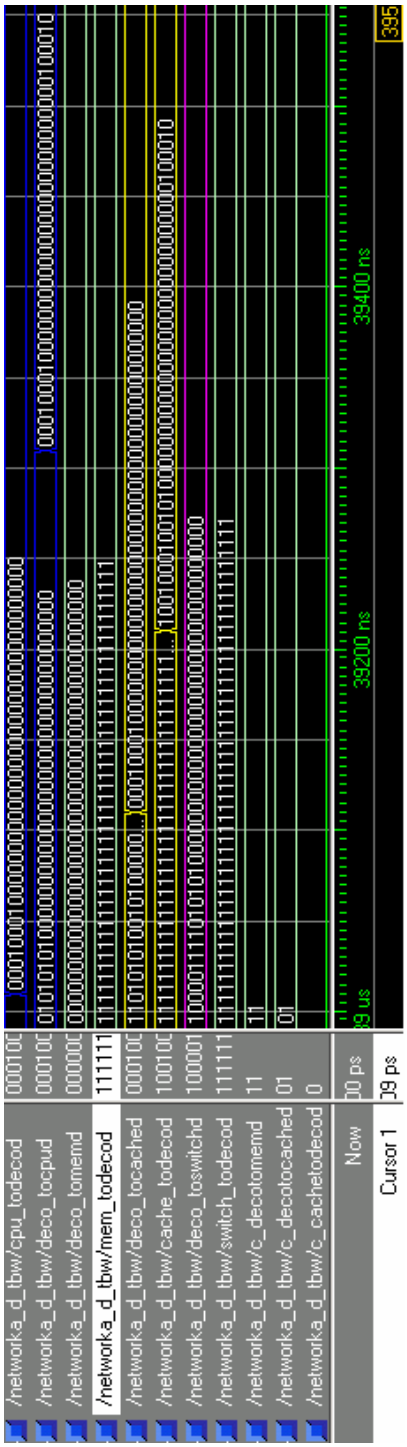


Figure 4.12: Node D: Cache Hit on $22 : H.O.L

4.3.6 Node A: Write $44 with $00000004

Node A performs this transaction, specifying the write by setting the m.s.b. of the packet. The Decoder will perform a cache read, resulting in a cache hit. Node A is a T.L.E and will therefore send a *Tail Update* transaction (transaction#14) to node B (its backward pointer field = 01). This is illustrated in Figure 4.13(a). When the packet reaches node B, its Decoder will perform a cache read. It is currently a R.L.E, and therefore updates its cache state to the T.L.E state.[11] A *pointer update confirmation* packet is then returned to node A (transaction#12). This is illustrated in Figure 4.13(b).

Upon receipt of this confirmation packet, node A will now have *popped* itself out of the shared list for address $44. The next step is to update memory and purge the shared list, giving node A an exclusive copy of the shared data. Address $44 is located in node B's address space, so a packet is sent to node B using transaction#11. This is illustrated in Figure 4.14(a). Node B updates its memory, receiving the current head of the list, node D, in the return packet. Node B then sends a *purge rest of list* transaction (transaction#8) to node D. This is illustrated in Figure 4.14(b). As node D is the H.O.L (a cache read is performed before the cache wipe), it will wipe its cache line entry for address $44 (the Decoder sets cache control signal to wipe, i.e. 10) and forward the purge rest of list transaction to the next in line node, node C (found using its forward pointer field). This is illustrated in Figure 4.15(a). Node C performs the same action as node D, forwarding the purge transaction to the T.L.E, node B.

As node B is the T.L.E (once again a cache read is always performed before the cache wipe), it is responsible for returning confirmation of the purge to the new head. This is done using transaction#7, *purge of list complete, permission to write cache line.* This is illustrated in Figure 4.15(b). Node A will receive this packet, write to its cache and send confirmation to the CPU. This indicates that the write has taken place, thus completing node A's final transaction (its request line to the Scheduler

[11] If this node had been a H.O.L entry, its state would be update to the H.O.E.L. This has been discussed previously in the Implementation chapter.

will not be re-asserted). This final step is illustrated in Figure 4.16, resulting in node A becoming H.O.E.L for address $44, with modified data in its cache.

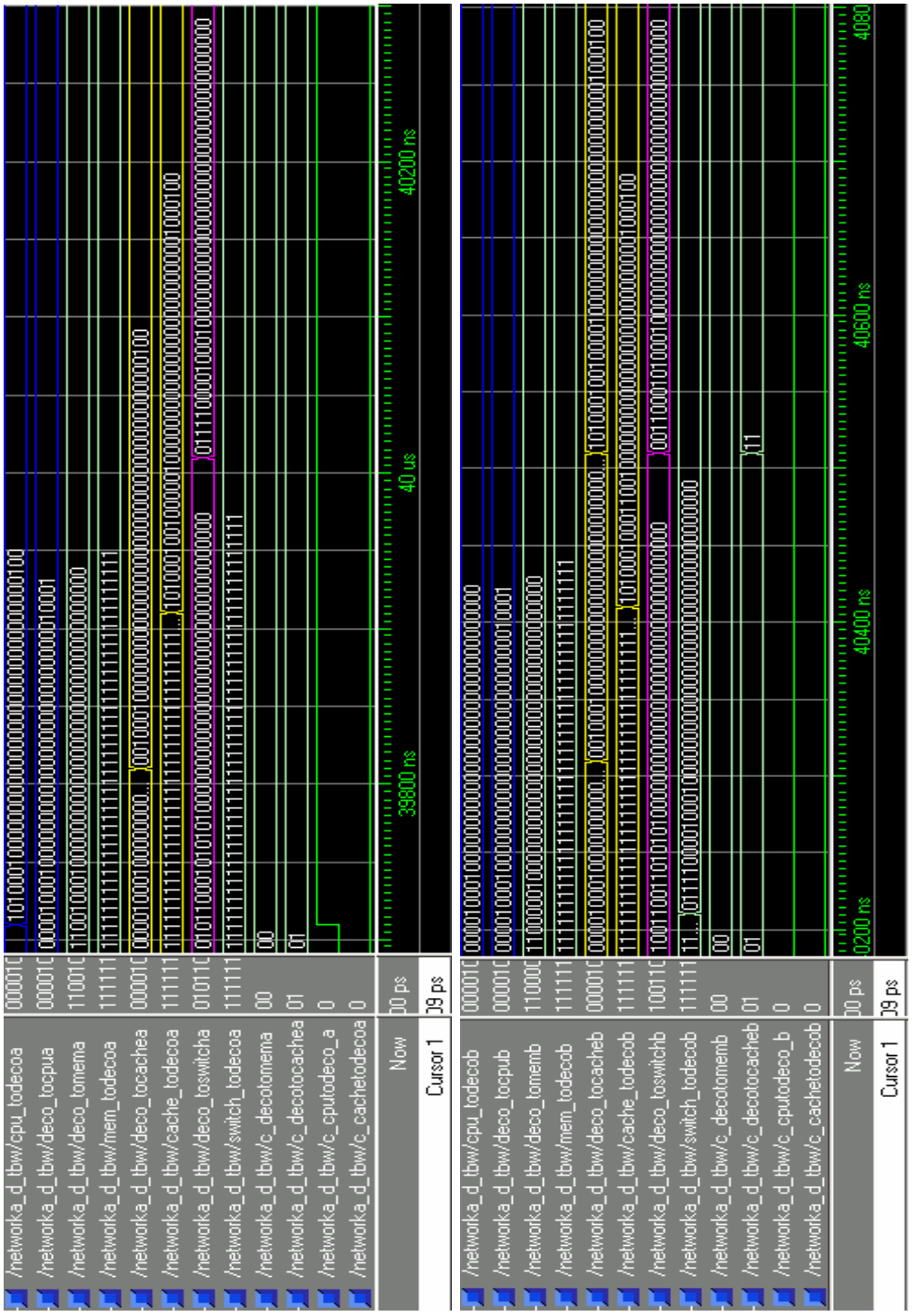
4.3.7 Node B: Write $88 with $00000008

Node B's cache performs a hit when queried with the address $88. As node B is a R.L.E, both of its neighbouring nodes in the list need a pointer update transaction sent to them. Node A is sent an *update backward pointer* packet (transaction#9), giving it node C's address as the new pointer value. Node B's address is stored in the 2 m.s.b's of the data field, so that node A can reply with confirmation of this update. This first step is illustrated in Figure 4.17(a)'s screen-shot. Node A will update its pointer accordingly and send confirmation back to node B. This can be seen in Figure 4.17(b). Upon receipt of this confirmation packet, node C is then sent an *update forward pointer* packet (transaction#10), giving it node A's address as the new pointer value. This second step carried out by Node B is illustrated in Figure 4.18(a). Node C will return a confirmation packet upon completion. Node C's processing steps are illustrated in Figure 4.18(b).

The next step for node B is to update memory and purge the shared list of address $88. $88 is located in node C's address space. The screen-shot for this step is shown in Figure 4.19(a). Node C will update its memory and send a purge r.o.l transaction to the head entry, node D. This processing step is illustrated in Figure 4.19(b). Node B will eventually receive confirmation of the shared list deletion from the T.L.E node A, giving it exclusive ownership of the address $88 block and allowing it to write over its cached copy. Node B then sends its CPU confirmation that the write has taken place, completing its final transaction. This final step is shown in Figure 4.20.

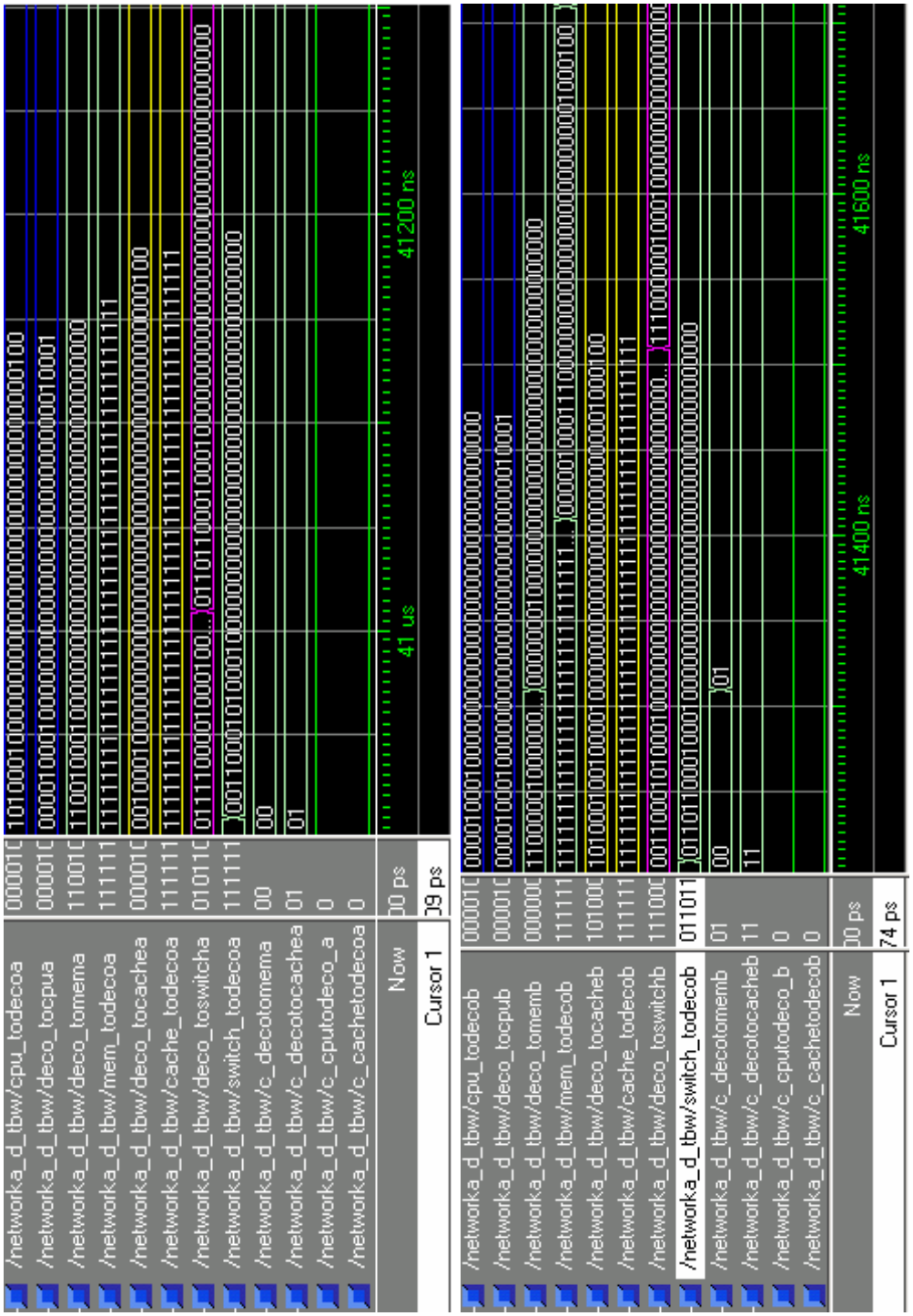
4.3.8 Node C: Write $22 with $00000002

Node C is a R.L.E for this address block. The steps involved in the execution of this transaction are almost identical to node B's write to $88. A R.L.E needs to pop itself out of the shared list by sending pointer update transactions to its neighbouring



(a) Node A sends Tail Update to Node B (b) Node B sends Pointer Update Confirmation to Node A

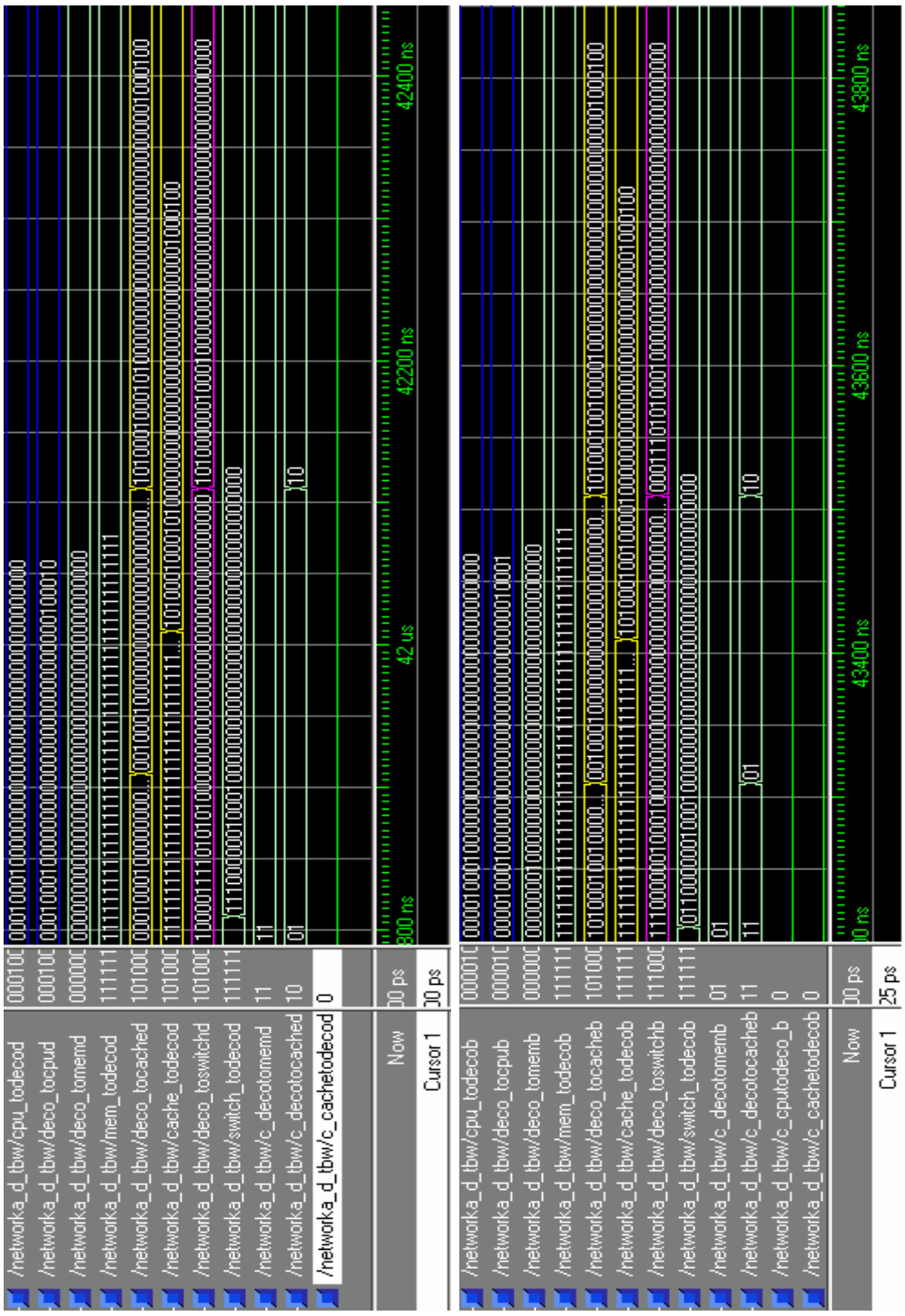
Figure 4.13: Node A: Write $44 with $00000004



(a) Node A sends Memory Update and Purge Command

(b) Memory updates and sends purge to head

Figure 4.14: Node A: Write $44 with $00000004



(a) Head wipes and forwards to next node (b) T.L.E sends confirmation to new head in list

Figure 4.15: Node A obtains exclusive write permission of address $44

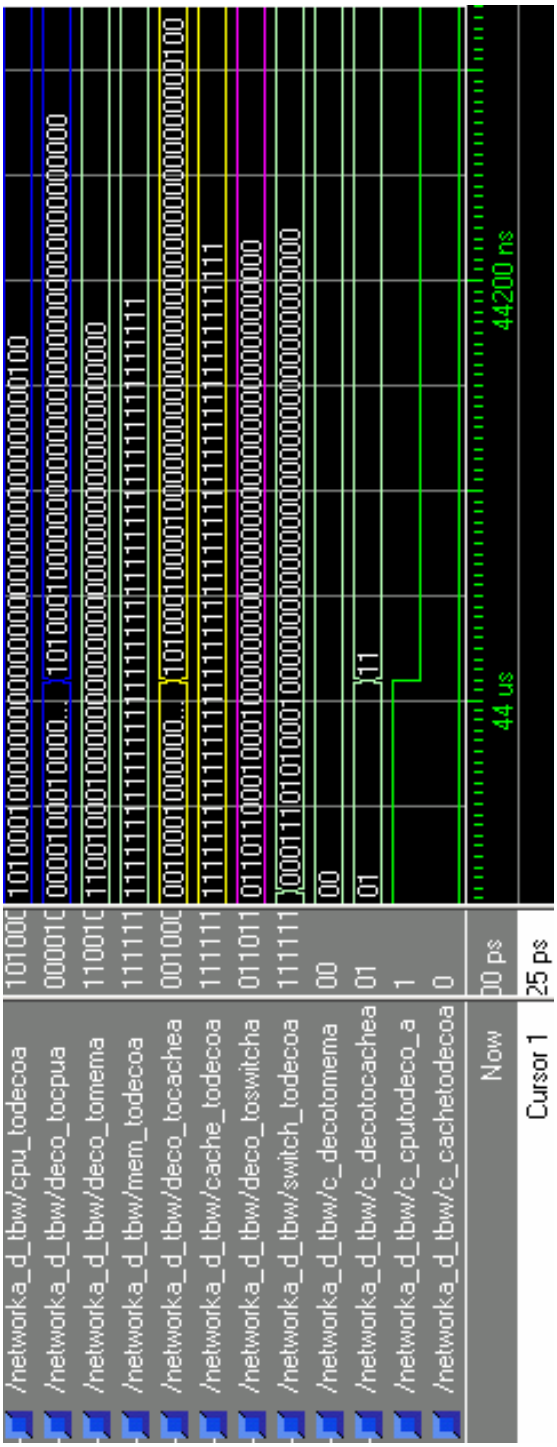
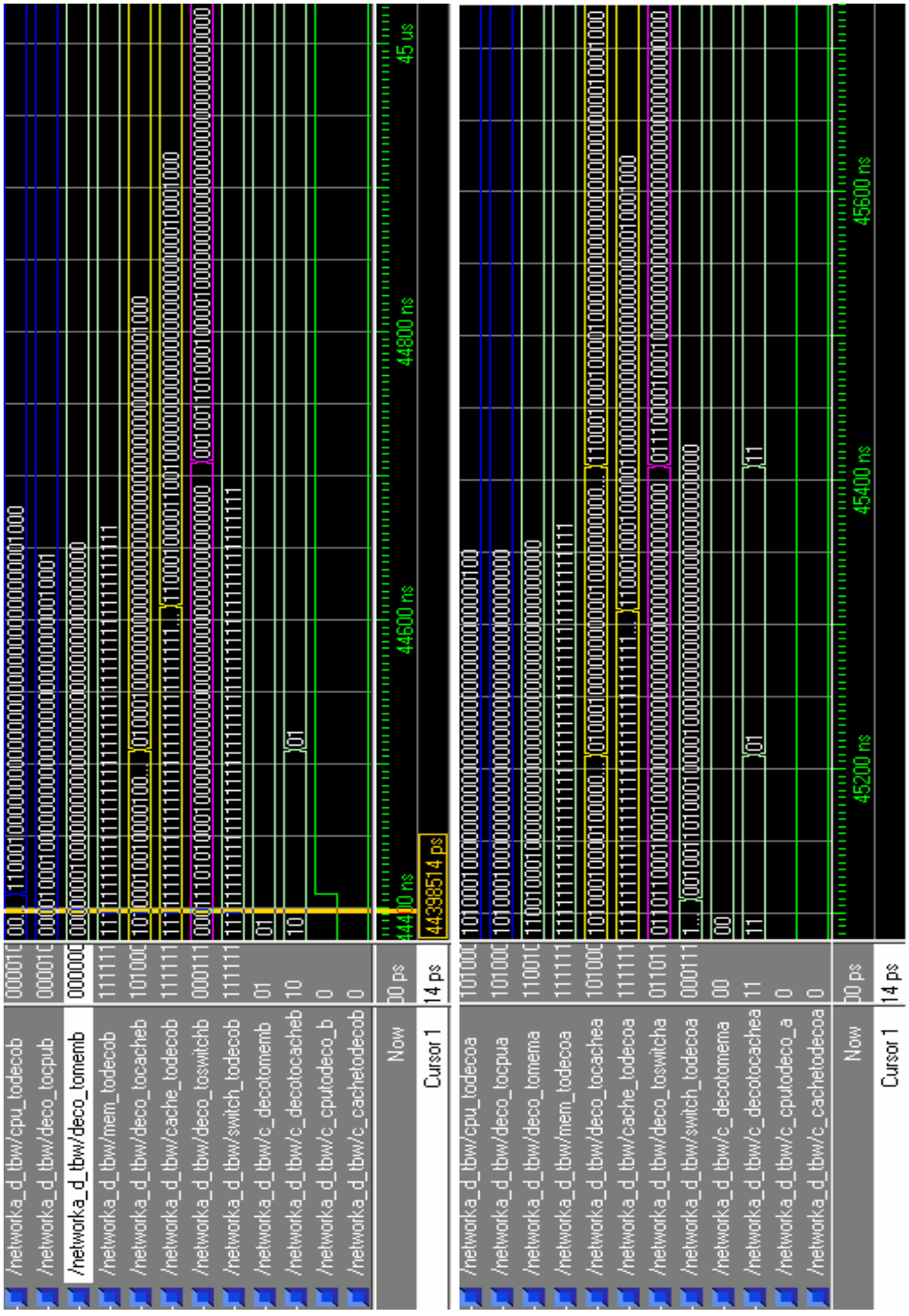


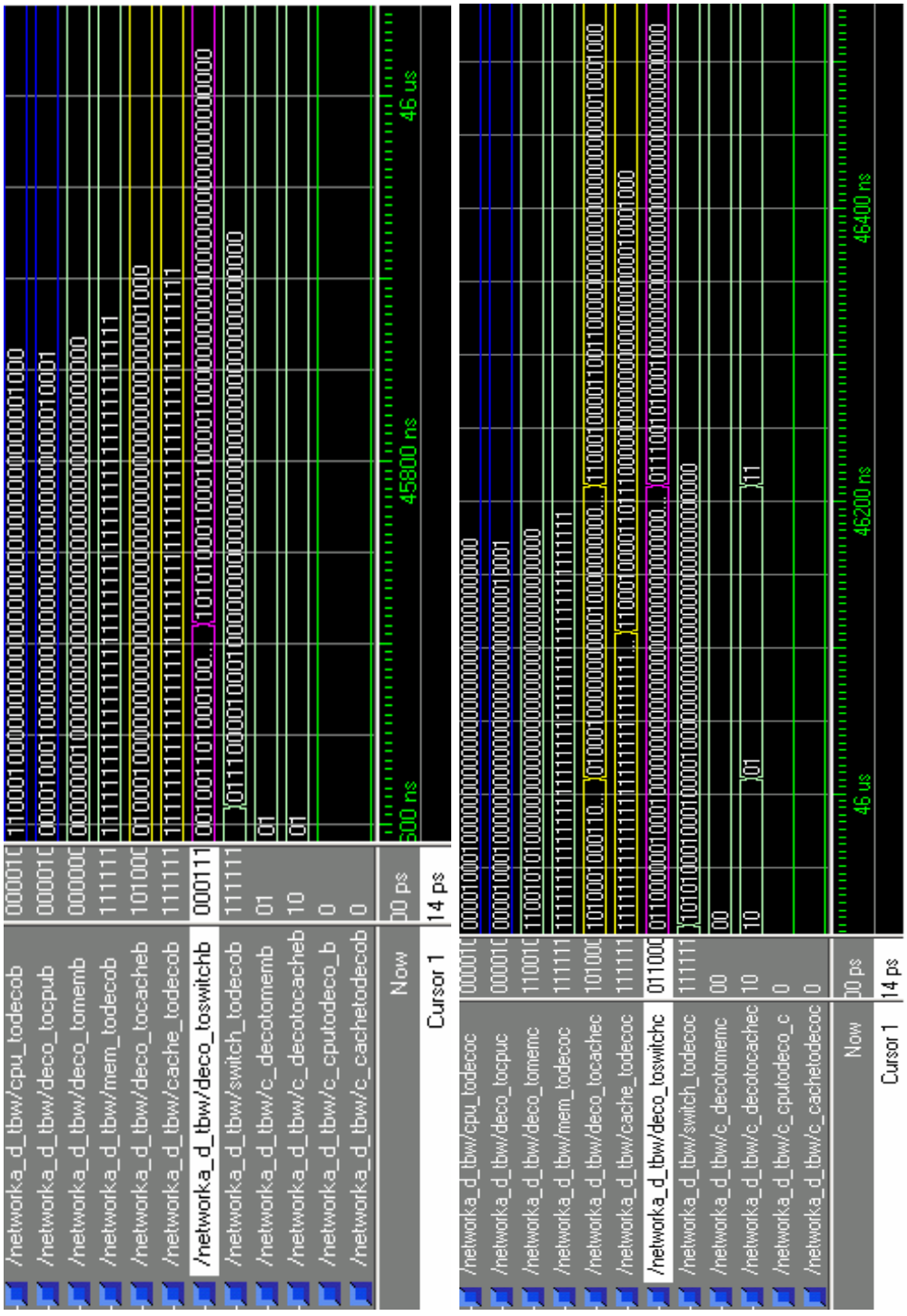
Figure 4.16: New Head writes to cache



(a) Node B sends Back ID Pointer Update to Node A

(b) Node A Sends Update Confirmation to Node B

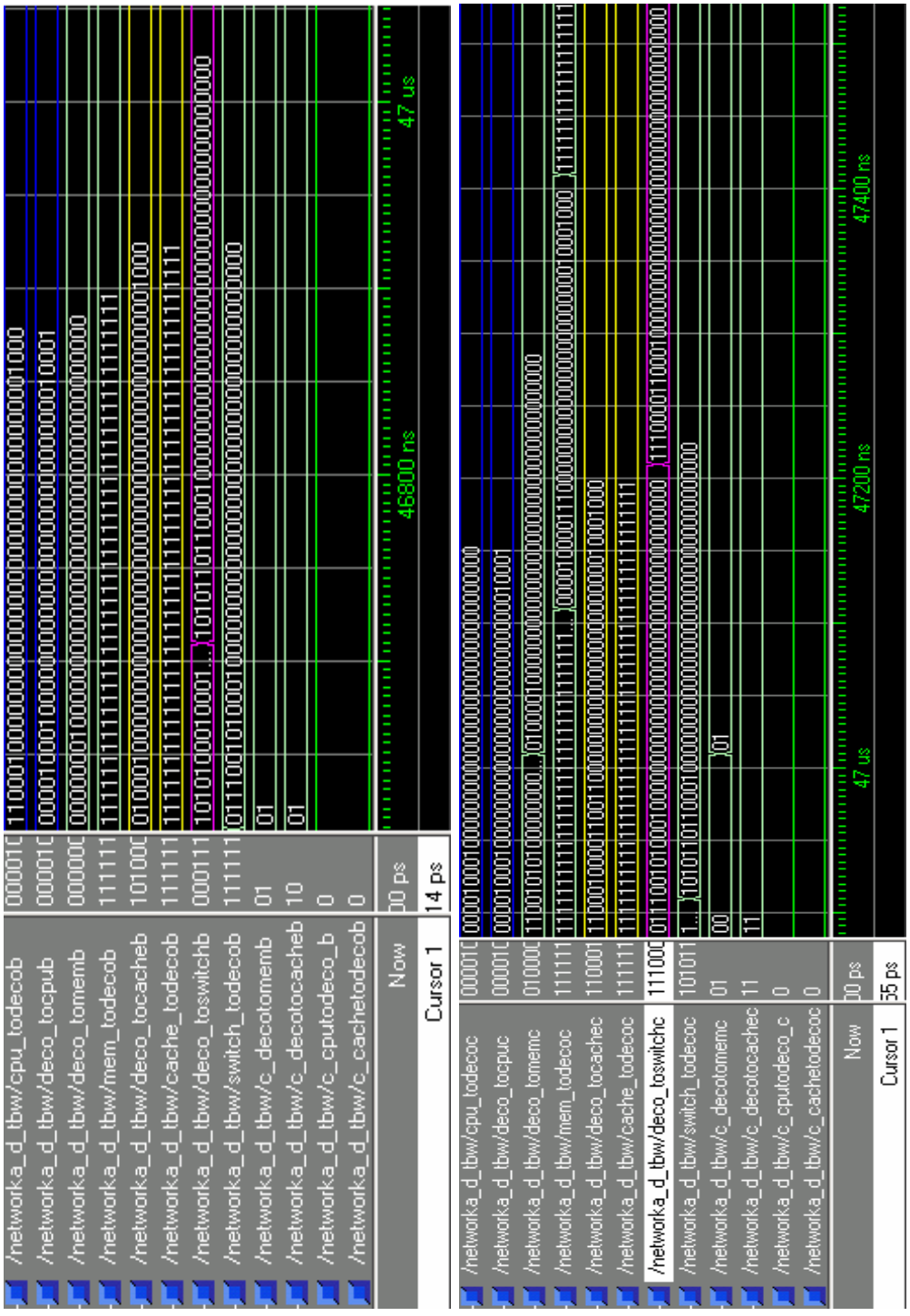
Figure 4.17: Node B: Backward Pointer Update



(a) Node B sends Forward ID Pointer Update to Node C

(b) Node C Sends Update Confirmation to Node B

Figure 4.18: Node B: Forward Pointer Update



(a) Node B sends Memory Update and Purge command

(b) Node C updates memory location and deletes list

Figure 4.19: Node B: Memory Update And Purge List

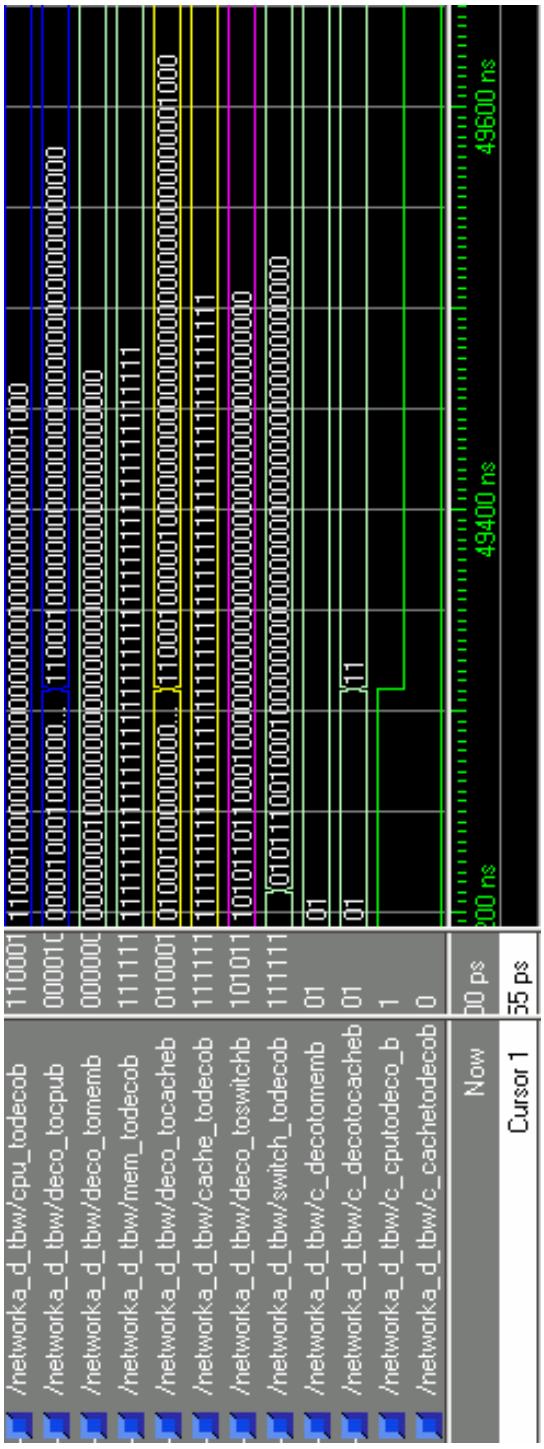


Figure 4.20: New Head writes to cache

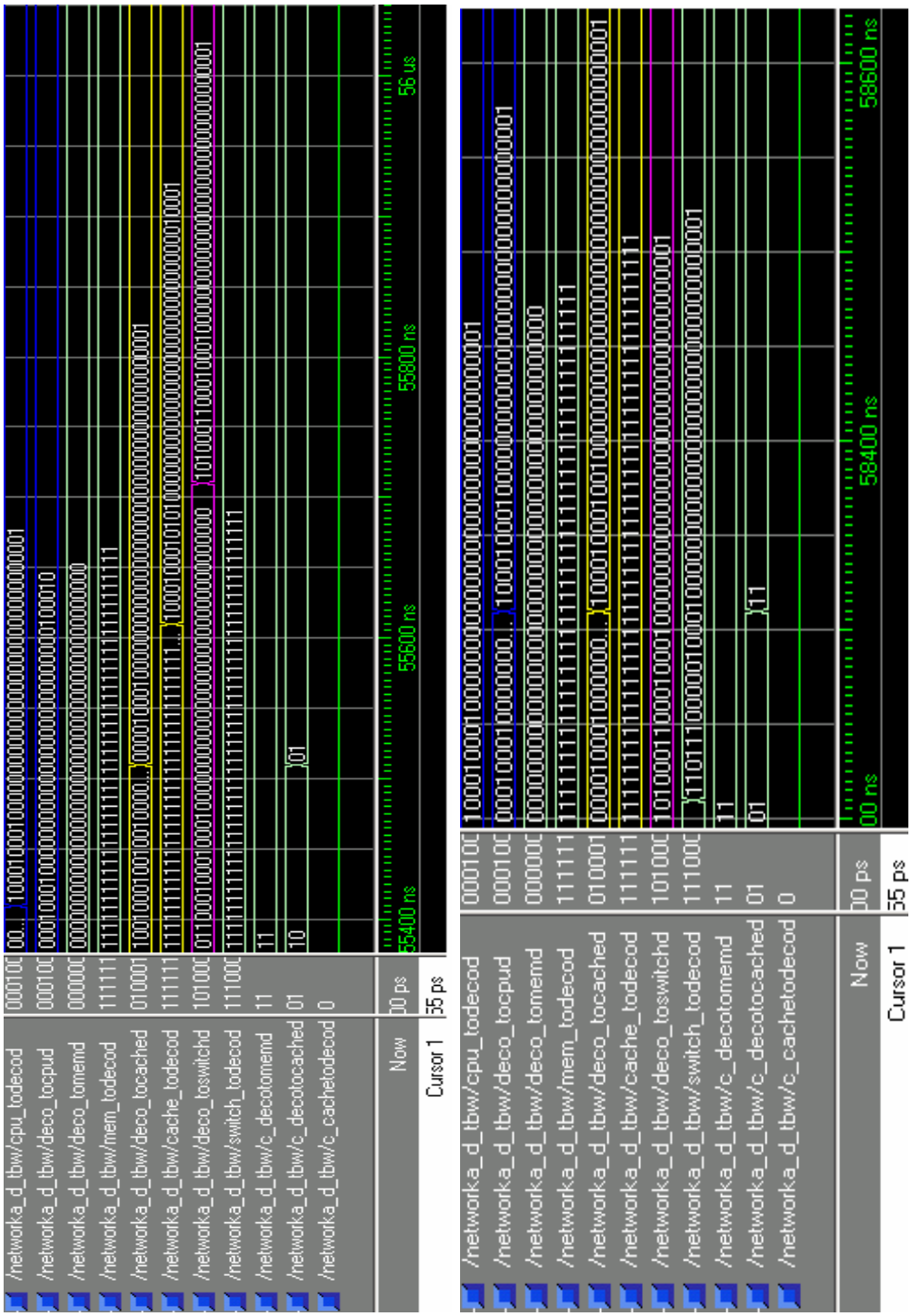
nodes. Once completed, the node needs to update the memory source of the address ($22 = node A) and purge the shared list, obtaining an exclusive copy of the data. Once exclusive ownership exists, the node can update its cache line with the write.

4.3.9 Node D: Write $11 with $00000001

The final transaction in the test-bench is performed by node D. As node D is the head list entry for address $11, it will only need to delete the other list entries. After the cache hit has taken place, node D's decoder simply sends a *purge rest of list* packet (transaction#8) to the next list entry, node C (indicated by the forward pointer field). This can be seen in Figure 4.21(a). The other list entries wipe their cached copies of address $11 and the T.L.E, node A, eventually sends confirmation that the list has been purged successfully, using transaction#7. Node D, now H.O.E.L, will receive this confirmation and update its cache line entry for address $11. This final step is illustrated in Figure 4.21(b).

4.4 Evaluation

This chapter discussed how the deign was synthesised and tested. From the example transactions shown in the previous sections, it can be seen that the SCI cache coherence protocol was implemented correctly. The scalability advantages of the protocol were not highlighted, as the design was restricted to 4 nodes for demonstration purposes. The design allows each CPU to be programmed with different transaction, and to execute them under the control of the Scheduler entity. In SCI, each node in the network is responsible for the update of the list. This synchronous design allows the clear observation of the individual responsibilities of each node. For students with a knowledge of VHDL, this design is of value and could be used as a teaching tool, allowing them to interactively learn about the SCI distributed directory cache coherence protocol.



(a) Node D purges list

(b) Node D is given write permission

Figure 4.21: Node D: Write $11 with $1

Chapter 5

Conclusions

This final chapter will draw conclusions based on the results outlined in the previous chapter. Future work, which would add to the goals achieved in this project, will also be suggested.

5.1 Conclusions

This project successfully implemented a VHDL simulation model of the SCI directory-based cache coherence protocols. The design simulates 4 nodes interconnected in a DSM architecture of unidirectional point-to-point links. Each CPU can be programmed with different transactions, performing reads and writes to the global address space, while the SCI cache coherence protocols maintains coherency. The results from the previous chapter are conclusive. The selected transactions showed the SCI Cache Coherence Protocol functioning correctly. The design was fully synthesised using the XST tool, thus making the project a success.

5.2 Future Work

Additional functionality could be added to this project in a number of different ways. The exact SCI node model could be designed and implemented in VHDL. The entities designed in this project could be included in the node application layer of the model, providing a layer of abstraction. The SCI split-transaction feature could

also be implemented, using the request and response sub-actions outlined in Chapter 2. Further work could be done in designing different SCI topologies to interconnect the nodes in the network. One final suggestion would be to interconnect a network of FPGA's, using the SCI link controller chip. Each FPGA could implement the functionality of a node in the network, applying the protocol designed in this project to maintain cache coherency.

Appendix A

Pseudo Random Number Generator

[1] Random number generators can be extremely useful in a variety of computer programs. The code however, may be difficult to debug because results are not always easy to repeat. Most programmers use "pseudo-random" number generators instead instead of truly random values, since these algorithms produce a sequence of numbers that appear to be random, but the pattern can be repeated if started with the same initialization conditions. There are many interesting algorithms for pseudo-random numbers.

Pseudo-random number generators usually have several potential difficulties including:

1. The algorithm may become cyclic, repeating the same sequence that that was given before.

2. The sequence can decay eventually into a pattern that does not appear random at all.

3. The numbers are not uniformly distributed.

[1] Information and Figures for Appendix A come from *http://www.tjhsst.edu/ dhyatt/arch/random.html*

A.1 A Pseudo-Random Number Generator Using the XOR Operation and Bit Shifting

The following algorithm uses the "exclusive OR" (XOR) operation to take a bit pattern of some seed number, and generate a pseudo-random number that can be used as the next seed. This bit manipulation process modeled after the Tausworth Algorithm, is known to have a a relatively good distribution and a long cycle before repeating. The example illustrated here uses an eight bit entity to demonstrate how the algorithm works, but ideally would be based on a datatype that contains a very long string of binary digits.

The general concept is that the bits from the high order position in the original seed are shifted into the low order positions. The XOR operation between those two values "flips" some of the bits. Then the low order bits are shifted into the high order position to do the same thing. Once both shifts and flips are done, the resulting number becomes the new seed.

Notice that the sum of bits used in the two shifts adds up to be the total length of the data type, in this case 8 bits. If a 32-bit integer were used as the data type, the sum of the left and right shift operations should equal 32. Another important detail is that shifts are "open" or "unsigned", where zeros fill from the left and the right.

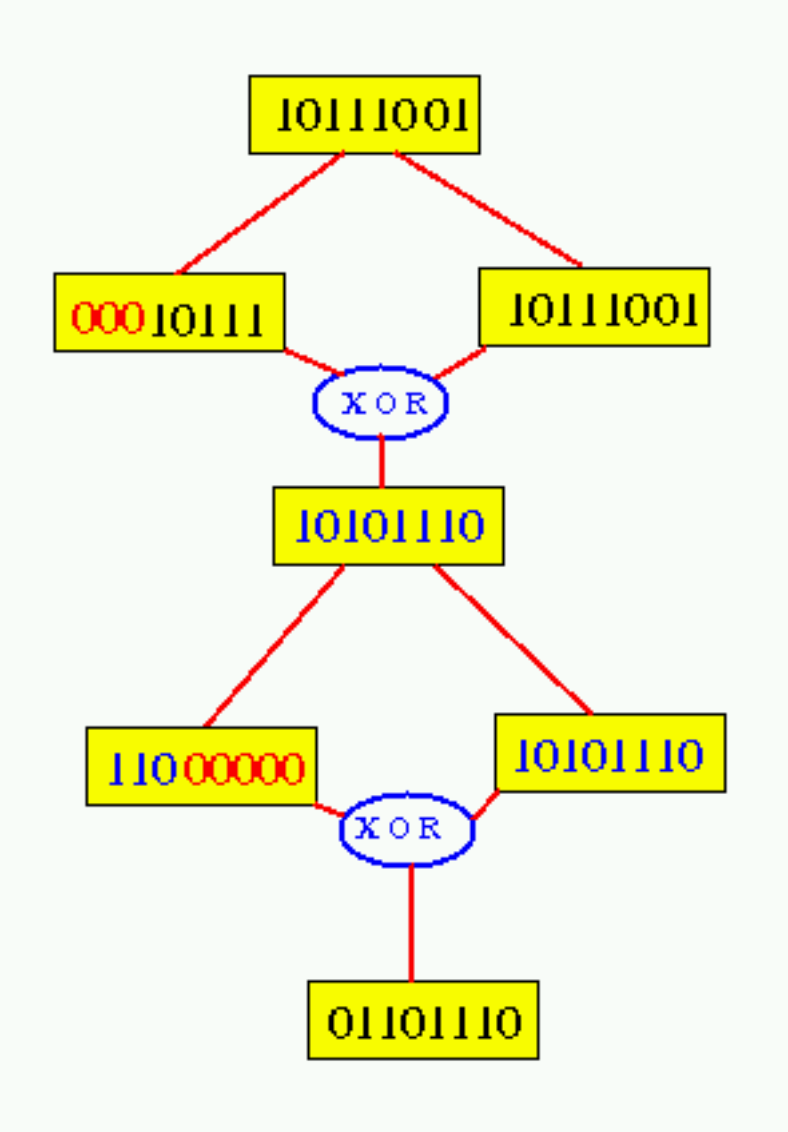


Figure A.1: Pseudo Random Number Generator

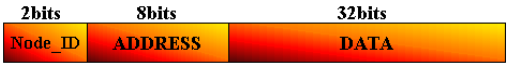
Appendix B

Packet Structures

(a) Decoder to Cache Packet Format

(b) Cache to Decoder packet format

Figure B.1: Decoder-Cache packet structure



(a) Decoder to Memory Packet Format

(b) Memory to Decoder packet format

Figure B.2: Decoder-Memory packet structure

Figure B.3: Interconnect Packets

Figure B.4: Decoder-CPU packet structure

Bibliography

[1] Martin Schulz. Shared Memory Programming on NUMA-based Clusters using a General and Open Hybrid Hardware/Software Approach. Research Report Series, Vol.24, 2001.

[2] IEEE Computer Society. IEEE Std 896-1994: information technology - microprocessor systems - Futurebus+ - Logical Protocol specification. The Institute of Electrical and Electronics Engineers, Inc., 345 47th Street, New York, NY 10017, USA, April 1994.

[3] M. Galles and E. Williams. Performance optimization, implementations, and verification of the SGI Challenge multiprocessor. Technical report, January 1994.

[4] James Archibald and Jean-Lou Baer. Cache Coherence Protocols: Evaluation using a Multiprocessor Simulation model. ACM Transactions on Computer Systems, 4(4):273-298, November 1986

[5] Mark Heinrich. The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. Ph.D. Dissertation Computer Systems Laboratory Stanford University October 1998

[6] BBN Laboratories, Butterfly Parallel Processor. Tech. Rep. 6148, Cambridge, MA, 1986.

[7] G.F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In Proceedings of the 1985 Conference on Parallel Processing, pages 764-771, 1985.

[8] R.J. Swan et al. The implementation of the Cm* multi-microprocessor. In Proceedings AFIPS NCC, 645-654, 1977.

[9] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In Proceedings of the 11th International Symposium on Computer Architecture, pages 348-354, 1984.

[10] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. IEEE Transactions on Computers, C-27(12):1112-1118, December 1978.

[11] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In AFIPS Conference Proceedings, National Computer Conference, pages 749-753, June 1976.

[12] Charles P. Thacker, Lawrence C. Stewart and Edwin H. Satterthwaite JR., IEEE Transactions on Computers Vol. 37, No. 8, August 1988.

[13] R. Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. thesis. Stanford University Technical Report CSL-TR-92-550, October 1992.

[14] J.Kuskin, D.Ofelt, M.Heinrich, J.Heinlein, R.Simoni, K.Gharachorloo, J.Chapin, D.Nakahira, J.Baxter, MHorowitz, A.Gupta, M.Rosenblum, and J.Hennessy. The Stanford FLASH multiprocessor. In Proceedings the 21st Annual International Symposium on Computer Architecture, pp.302-313, April 1994.

[15] C.K.Tang. Cache Design in the Tightly Coupled Multiprocessor System. In AFIPS Conference Proceedings, National Computer Conference, NY, pages 749-753, June 1976.

[16] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. IEEE Computer, June 1990.

[17] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In Proceedings of the 1990 International Conference on Parallel Processing, pages I.312-I.321, August 1990.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In Proceedings of the 24th International Symposium on Computer Architecture, pages 241-251, June 1997.

[19] IEEE Computer Society. IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface. The Institute of Electrical Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.

[20] Oliver Pugh, Jeff Cowhig, Jonathan Crowe and Eoin Blacklock. Scalable Coherent Interface (SCI). 4BA2 Technology Survey 2005.
http://ntrg.cs.tcd.ie/undergrad/4ba2.05/group12/index.html.

[21] Dolphin Interconnect Solutions
http://www.dolphinics.com

[22] IEEE Computer Society. IEEE Std 1496-1993: IEEE standard for a Chip and Module Interconnect Bus: SBus. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, September 1993.

[23] Welcome to PCI SIG
http://www.pcisig.com/

[24] Fujitsu-Siemens
http://www.fujitsu-siemens.com/hpc/products/hpcline/

[25] Philips Medical Systems
http://www.medical.philips.com/main/products/ultrasound/general/iu22/features/-architecture.html

[26] NLX Corporation

*https://enotes.nlxcorp.com/corporate/training/weblib22.nsf/src/webdoc*home/?open*

[27] Camber LTD

http://www.camber.com/sandp.asp?n=modsim/

[28] Hermann Hellwagner and Alexander Reinefeld, editors. SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters. Lecture Notes in Computer Science Vol.1734. Springer Verlag, October 1999.

[29] VHDL Tutorial. Jan Van der Spiegel. University of Pennsylvania. Department of Electrical Engineering. *http://www.seas.upenn.edu/∼ee201/vhdl/vhdl_primer.html#_Toc526061340*

[30] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313, April 1994.

[31] An Evaluation of Directory Schemes for Cache Coherence. Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. Computer Systems Laboratory Stanford University, CA 94305

[32] An Innovative Implementation for Directory-based Cache Coherence in Shared Memory Multiprocessors. Weisong Shi Weiwu Hu and Ming Zhu. Center of High Performance Computing, Institute of Computing Technology Chinese Academy of Sciences, P.O.Box 2704-35, Beijing 100080.

[33] Ashwini K. Nanda, Hong Jiang. Analysis of directory based cache coherence schemes with multistage networks. In Proceedings of the 1992 ACM annual conference on Communications. Kansas City, Missouri, 1992, pages 485-492

[34] Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry Computer Systems Laboratory Stanford University, CA 94305

[35] Badrish Chandramouli, Sita Iyer. A performance study of snoopy and directory based cache-coherence protocols. Department of Computer Science Duke University