High-Level Modeling of Network-on-Chip

M.Sc. thesis

Computer Science and Engineering
Informatics and Mathematical Modelling (IMM)
Technical University of Denmark (DTU)

31st of July 2006

Supervisor: *Prof. Jens Sparsø*
Co-supervisor: *Ph.d. student Mikkel Bystrup Stensgaard*

Matthias Bo Stuart (s011693)

Abstract

This report describes the design, implementation and testing of a high-level model of an asynchronous network-on-chip called MANGO that has been developed at IMM, DTU. The requirements to the model are twofold: It should be timing accurate, which allows it to be used in place of MANGO, and it should have a high simulation speed. For these purposes, different approaches to modeling network-on-chip and asynchronous circuits have been investigated. Simulation results indicate a simulation speedup on a magnitude of a factor 1000 over the current implementation of MANGO, which is implemented as netlists of standard cells.

Acknowledgements

This Master of Science project has been carried out at Informatics and Mathematical Modelling at the Technical University of Denmark in the spring and summer of 2006. I would like to thank my fellow students Morten Sleth Rasmussen and Christian Place Pedersen for thorough discussion of different issues that popped up along the way. I would also like to thank Tobias Bjerregaard, who created the current implementation of MANGO, for invaluable discussions on the inner workings of MANGO. I am grateful to Shankar Mahadevan for a kick-start discussion on how to model MANGO. I would especially like to thank my supervisor Jens Sparsø and my co-supervisor Mikkel Bystrup Stensgaard for invaluable guidance and discussions.

Matthias Bo Stuart
Kgs. Lyngby
July, 2006

Contents

List of Figures

Chapter 1

Introduction

Decreasing transistor sizes have led to more transistors being able to be placed on a single chip. This has led to ever more complex chips, which can hold an entire system, so-called System-on-Chip (SoC). SoCs can be described as “heterogeneous architectures consisting of several programmable and dedicated processors, implemented on a single chip” [11]. These processors, memories, IO-controllers, etc - called Intellectual Property Cores (IP-Cores) - may be connected by a number of different means such as busses, point-to-point connections and network structures. Busses have the issue that wires do not scale very well, while at the same time more IP-cores are connected to them further increasing the capacitance on the bus. Less bandwidth is available to each IP-core at an increased cost in latency and power consumption.

Direct connections between IP-cores result in very inflexible designs, as the available connections in the system are fixed, once the chip enters production. If two IP-cores that have no direct connection needs to communicate, a path through other IP-cores must be taken, requiring these intermediate IP-cores to handle communication rather than their own computation, if such a path exists at all. Furthermore, single IP-cores become much harder to reuse between designs, due to the lack of a single standard interface to the IP-core. At the same time the scalability of direct connections is very poor, as the number of connections may grow quadratically with the number of IP-cores in the system.

1.1 Network-on-Chip

The Network-on-Chip (NoC) approach to communication systems has none of these issues. NoCs make use of segmented communication structures similar to computer networks. Network Adapters (NA) provide a standardised interface between the IP-core and the network, while the network is made up of network nodes connected by links. An example NoC is shown in figure 1.1.

The length of wires is kept fairly constant relative to transistor sizes due to the segmented structure of networks. The longest wires in the network are used exclu-

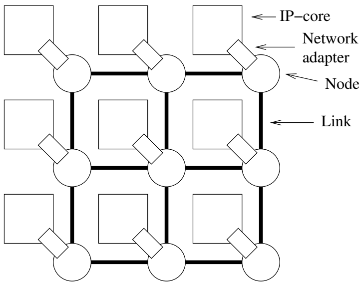


Figure 1.1: An example of a Network-on-Chip based system.

sively to connect neighbouring network nodes, which has the effect that these wires only have one driver, reducing the capacitance of the wires. Long wires may also be pipelined, further reducing wire length while increasing throughput at the same time. Using standardised interfaces to the network makes design-reuse effortless, as a plug-and-play style of system design becomes possible.

1.2 System Modeling

In early stages of the design process when exploring different network topologies, application mappings and other system-level considerations, there is little or no need for fully accurate and synthesisable descriptions of the IP-cores in the system, as these take a very long time to simulate. For purposes of system exploration, high-level models that produce a reasonably accurate estimate of performance and possibly power consumption and area should be used.

Similarly, a high-level model of the communication system should be employed at this stage, for fast simulation. The level of detail in such a high-level model of a NoC depends on the abstraction level at which it is to be used. While application programmers might not care at all about communication between processes having a non-zero latency, system designers need a detailed communication model in order to ensure the system functions properly.

Another use for such a model is for the network designer to explore the impact of different implementations of different components of the network. For example, different switching structures, arbitration schemes, link encodings and packeting schemes may be explored by use of a high-level model. New hardware structures may be examined without the need to accurately simulate the uninteresting environment to these structures, and packeting schemes may be tried out without restrictions on bit widths.

1.3 Asynchronous Circuits

Asynchronous - or clock-less or self-timed - circuits are a type of circuits which use local handshakes for controlling the data flow through the system, rather than a global controller as seen in synchronous circuits. Asynchronous circuits have an advantage in NoCs as they allow for Globally-Asynchronous Locally-Synchronous (GALS) designs, where the lack of a global clock reduces timing closure to a local problem for each IP-core. Furthermore, asynchronous circuits have zero dynamic power consumption when idle. As a NoC will not be utilised 100% most of the time, this can lead to an advantage in power consumption over synchronous circuits. This advantage over synchronous circuits may however be lessened due to increasing leakage currents in newer manufacturing technologies.

Asynchronous circuits are however not as straight-forward to create timing accurate models of compared to synchronous circuits. While clock-cycle accurate models of synchronous circuits may be developed, a similar notion does not exist for asynchronous circuits.

1.4 This Work

This thesis describes the development of a high-level model of a Network-on-Chip called MANGO which is developed at IMM DTU. This Network-on-Chip is asynchronous, requiring different modeling techniques than those commonly used for synchronous designs. The current implementation of this NoC is constructed by netlists of standard cells from a given library, making it very slow to simulate.

The purpose of the model is twofold: First, it should be usable for rapidly evaluating different system designs, characterised by different network topologies and application mappings. Second, a network designer should be able to accurately examine the impact of new implementations of physical components using the model. In order to fulfil these requirements, a fairly timing accurate model that is fast executing is required.

Chapter 2 will introduce the Network-on-Chip concept, chapter 3 will give an introduction to MANGO, chapter 4 will take a look at different approaches to modeling NoCs, chapter 5 will introduce asynchronous circuits and develop a method for modeling the circuits found in MANGO, chapter 6 will describe the design choices made for the model, chapter 7 will describe implementation details of the model, chapter 8 will present and discuss the results of simulations of both the model and the current implementation of MANGO while chapter 9 will discuss how the model may be applied to system modeling and design and where to proceed with the model. Chapter 10 will conclude the report.

Chapter 2

Network-on-Chip

This chapter gives a general introduction to Network-on-Chip. First, the main characteristica of a NoC are presented. Then the basic components that comprise a NoC are introduced, and finally different abstraction levels of NoCs are discussed.

2.1 Network-on-Chip Characteristica

Network-on-Chip may be seen as a subset of System-on-Chip (SoC) [7]. A SoC is an entire system implemented on a single chip, whereas a NoC is one approach to the communication structure in a SoC. A NoC is made up of network adapters, nodes and links as illustrated in figure 1.1.

2.1.1 Transaction Transport

One of the characteristica of a given NoC is how transactions are transported between IP-cores. An IP-core may communicate with any other IP-core in the system by means of the NoC. As long as the protocol of the standardised interface is complied with, the IP-cores need no knowledge of the NoC.

Often a transaction has too many bits to be transported through the NoC all at once. Therefore, transactions are split into smaller transmittable parts and then re-assembled at the destination before being presented to the IP-core. Two general approaches to transporting these transmittable parts - sometimes called flow control units (flits) - are store-and-forward and wormhole routing.

In a NoC using store-and-forward, a node must receive all the flits in a single transaction before the flits are sent on to the succeeding node. When using wormhole routing, the first flits of a transaction can be sent on from a node before the last flit has arrived. This allows a transaction to span many nodes, reducing buffering needs, but potentially increasing the impact of stalls.

2.1.2 Routing Schemes

NoC routing schemes range from basic static to highly adaptive schemes. For NoCs with grid or torus topologies, a simple deadlock free routing scheme is xy-routing. The strategy is to route along one axis of the NoC before routing along the other axis. A proof that no deadlocks occur using this strategy is presented in [8]. However, it makes the assumption that a flit that has reached its destination will be removed from the NoC. Some IP-cores - such as on-chip memories - may need to insert a response to a previous request before accepting a new one. If this is the case, the proof must be made at the system level rather than at the network level.

Adaptive schemes generally try to minimise congestion in the NoC by routing around areas with heavy load. Adaptive schemes generally have a higher cost in terms of routing logic and complexity of avoiding deadlocks, but may yield better performance under heavy load compared to static schemes.

Another issue in routing is where routing decisions are made. Either the route is pre-determined - called source routing - or it is determine on a hop-to-hop basis - called distributed routing. Source routing goes well with wormhole routing, as the entire route needs to be contained in a flit. If store-and-forward were to be used, this would require a large header in each flit. However, distributed routing goes well with store-and-forward, as only a few bits are required to determine the coordinates of the destination. In an adaptive NoC, distributed and source routing may be combined, allowing a node to alter the route in order to minimise or avoid congestion in the NoC.

2.1.3 Service Levels

A NoC may provide a wide range of service guarantees. The most basic guarantee is that a transmitted flit will eventually arrive at its destination. This is characteristic of a so-called best effort (BE) net. Multiple flits from the same transaction need not arrive in the same order they were sent, but a guarantee may be provided that arrivals are in-order. In any case, the network adapter must reassemble the flits into a transaction, but an overhead in flit size is introduced if flits may arrive out of the order in which they were sent, as some header bits are needed to keep count of the order of the flits.

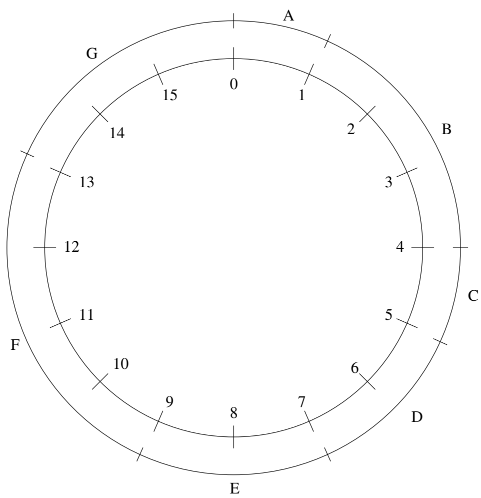
Another type of guarantee is a maximum latency or a minimum bandwidth or a combination of the two on a connection between two IP-cores. Such guarantees are commonly known as Guaranteed Service (GS). The guarantees can be either absolute or statistical. Whether statistical guarantees suffice depends on the application.

2.1.4 Virtual Circuits

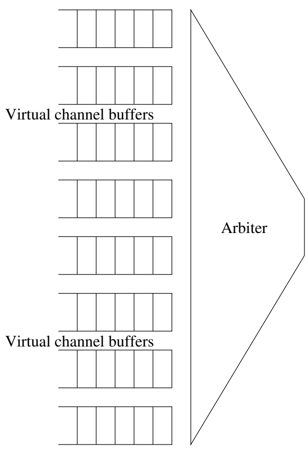
Standardised interfaces provide the illusion of a point-to-point connection between IP-cores. In a NoC, these connections are called virtual circuits and may also be used to guarantee that the NoC never enters a deadlock.

One possibility is to assign each connection a time interval in which transactions can be made. Figure 2.1(a) shows a system with 16 time slots and 7 connections, A through G. Such a system has a tight coupling between latency and bandwidth. Assume the current time is 1 and connection A needs to make a transaction. It then needs to wait 15 time slots before being able to make the transaction, as it has reserved the time interval $[0; 1]$. Connection F may at most have to wait 12 time slots, as it has reserved the time interval $[9; 13]$, but at the same time connection F has reserved 25% of the bandwidth in the system. Using this time slot allocation, it can be guaranteed that no contention will occur in the NoC, which results in very simple control circuits [7]. Furthermore, deadlocks are not an issue, as a flit will never have to wait for another in the NoC.

Another option is to make use of virtual channels. A channel is represented by a buffer, and a number of parallel channels exist on a link. Depending on the service guarantees of the NoC, fair access arbitration must be implemented on the link. Well-known arbitration schemes are static priorities and round-robin, but more elaborate schemes may be implemented. Figure 2.1(b) shows eight parallel virtual channels sharing a link by means of an arbiter. Deadlocks are prevented by only allowing one connection to use a given virtual channel buffer, making these private areas of a connection. If a flit only enters the shared areas of the NoC when it is known that it may enter a private area upon arrival, deadlocks do not occur.



(a)



(b)

Figure 2.1: 2.1(a): A time slot allocation with 16 time slots available for 7 connections, A through G. 2.1(b): An access control system using arbitration between eight virtual channels.

2.2 Basic Components

The following will give a brief description of each of the components shown in figure 1.1.

2.2.1 Node

The nodes and the links, constitute the basic framework of a NoC. Figure 2.2 shows a generic router model. Input and output buffers are connected through the switch, which is controlled by the arbitration and routing algorithms. The local ports connect to the network adapter, which in turns connects to an IP-core as shown in figure 1.1. A node needs not have both input and output buffers and if virtual channels are used, arbitration may be part of the output link controllers, as is the case in MANGO which is presented in chapter 3.

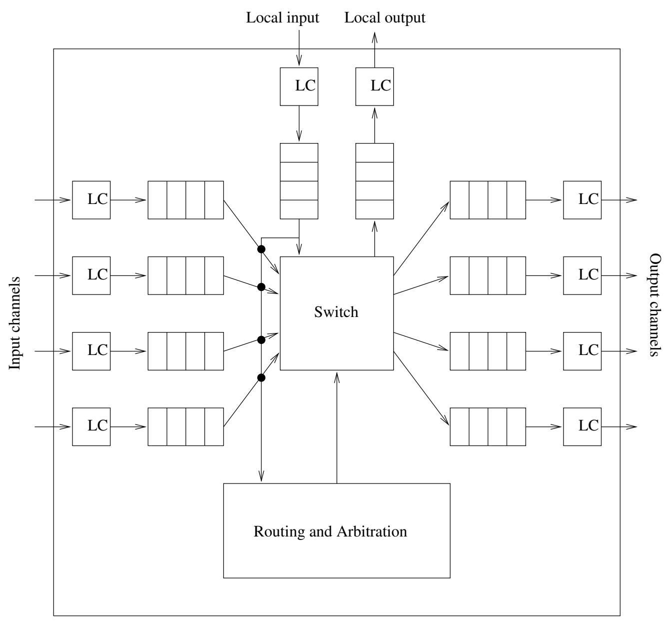


Figure 2.2: A generic model of a node, adapted from [7]. The boxes labelled LC are link controllers.

2.2.2 Link

Links are the wires that connect nodes. These should be kept at a reasonable length or possibly pipelined in order to provide a reasonable throughput. Due to the increased effect of variability on wire delays in deep sub-micron technologies, links may have to be heavily pipelined or make use of delay insensitive encoding [15] in order to not significantly reduce the production yield. This is a topic of ongoing research.

2.2.3 Network Adapter

The network adapter (NA) provides a conversion from the interface protocol used by the attached IP-core and the transmission format used in the NoC and back again. Multiple interface protocols may exist in a NoC, as long as it is possible to insert transactions from all protocols into same-sized flits for transmission. Each interface protocol requires a different NA.

2.2.4 Intellectual Property Core

The intellectual property cores (IP-cores) cover processing elements (CPU, DSP, ASIC), peripherals (on-chip memory, off-chip memory controller, off-chip bus interface), IO-controllers (UART, ethernet, VGA) and any other core one might put on a chip. Given a standard interface on the core as well as an NA with the same interface, it is possible to do a plug-and-play style of design using IP-cores from multiple different vendors, and reuse these cores between multiple designs.

2.3 Levels of Abstraction

NoCs may be considered at several different levels of abstraction. This section will give an introduction to levels of abstraction based on the designers working at different levels. In [7], levels of abstraction for NoC that resemble the OSI network model are presented.

Figure 2.3 illustrates the levels of abstraction based on the different design levels: Application, system and network designer.

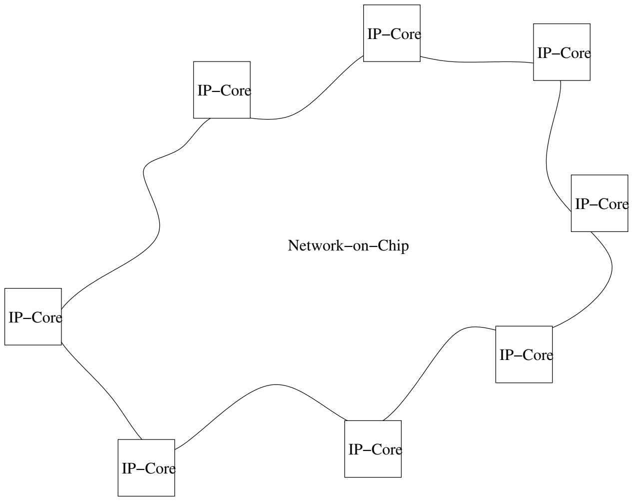
2.3.1 Application Level

At this level, the NoC is considered as a medium which provides connections between all cores. How these connections are made is irrelevant, and communication delays may be disregarded, essentially making the NoC a huge fully connected, delay-less crossbar. Communication may be handled by message passing or shared memory. Connections may be created explicitly by a function call such as `int myConn = openConnection(int dest)` and messages passed by `send(int myConn, struct myData)` and `struct myData = receive(int myConn)`. Another option is to have all possible connections open at all times, and simply identify the receiver by a unique identifier, such as a process ID. This level of abstraction is illustrated in figure 2.3(a).

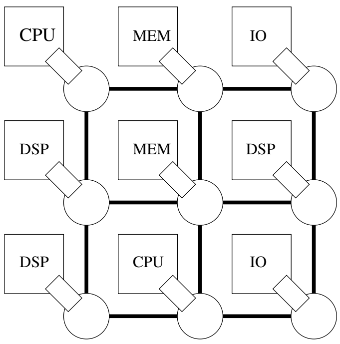
Models at this level of abstraction are independent of the actual NoC, and will not be considered further.

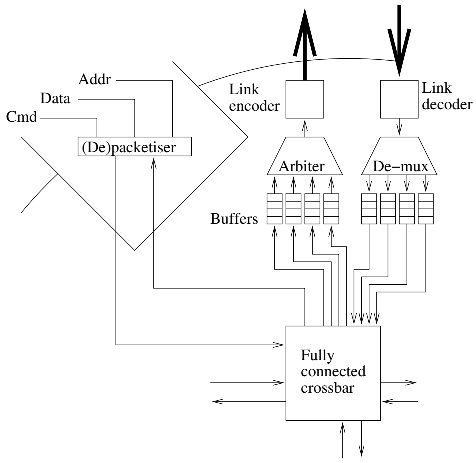
2.3.2 System Designer Level

Figure 2.3(b) presents a NoC as seen from the system designer. A mesh-topology with IP-cores attached to specific points in the NoC is shown. This mapping of IP-cores to access points must be able to satisfy the application's communication



(a)





(b) (c)

Figure 2.3: 2.3(a): At the application level, the NoC is simply considered as a medium which provides connections between all cores. 2.3(b): At the system level, issues such as network topology and application mapping are considered. 2.3(c): At the network level, circuit implementations are considered, while actual systems are not considered. The NoC is simply nodes, links and network adapters that provide a service and may be connected to form an actual NoC.

requirements using the services offered by the NoC. A reasonably accurate model of the NoC must be used in order to evaluate whether these requirements are met. Other objectives such as constraints on power consumption may also be a factor in deciding on a topology and mapping, if switching activity on the long wires in the links constitute a major contribution to power consumption, which may be the case in deep sub-micron technologies.

2.3.3 Network Designer Level

At this level, circuit implementations are considered, as shown in figure 2.3(c). Decisions on arbitration and routing schemes, principles for creating virtual circuits, which service guarantees are offered, how transactions are transmitted through the NoC and how data is encoded on the links are all made at this level. Any involvement with the system and application levels is to ensure that the communication requirements can be satisfied by the NoC. Possible restrictions on power consumption and area must also be taken into consideration during the design of a NoC.

Chapter 3

The MANGO Clockless Network-on-Chip

MANGO is an abbreviation of "Message passing, asynchronous Network-on-Chip providing guaranteed services over OCP interfaces," and is a NoC developed at IMM DTU by Tobias Bjerregaard as part of his ph.d. thesis [2].

Messages may be passed between two IP-cores connected to the network over a connection with service guarantees in the form of latency and/or bandwidth guarantees. Memory mapped access is also possible on a best effort (BE) network, which guarantees that messages eventually arrive and arrive in-order.

An introduction to asynchronous circuits and how they are modeled will be presented in chapter 5.

Guaranteed Services (GS) are provided as minimum bandwidth and/or maximum latency for a given connection. Guarantees are given as "data items pr second" for bandwidth and (nano)seconds for latency guarantees. As there is no global clock, latency guarantees can not be given in terms of "number of clock cycles." These guarantees are obviously affected by switching to a different technology.

Open Core Protocol (OCP) [14] is a collaboration between companies and universities for providing a standardised interface between IP-cores. Although only the OCP interface is supported currently, providing access points for different interfaces should be relatively simple. This standardised interface is used to provide point-to-point interfaces into a shared address space for IP-cores.

This chapter will present the node and link architectures and describe the current implementation of MANGO.

3.1 Node Architecture

The node architecture in MANGO is presented in [4] with in depth circuit descriptions in [6]. The main points are presented in the following.

3.1.1 Node Overview

The MANGO node architecture is shown in figure 3.1. A node has five input and five output ports: Four to links connecting to neighbouring nodes and one to a network adapter to which an IP-core can be connected. The nodes are output-buffered with a number of virtual channels (VC) on each interface.

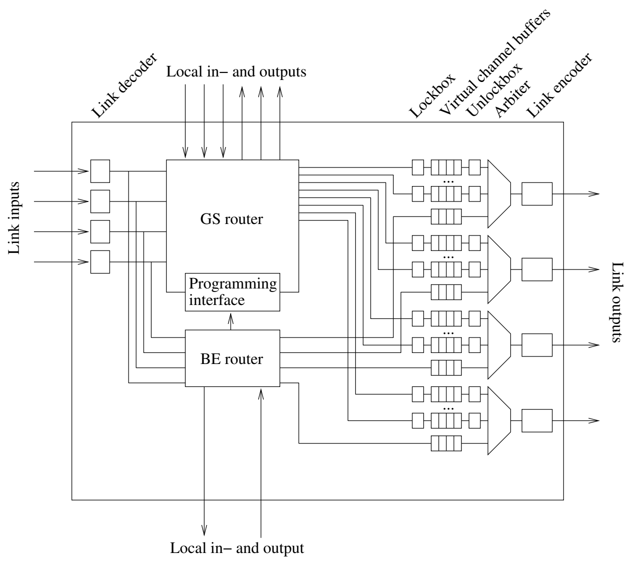


Figure 3.1: MANGO node architecture

VCs are provided by parallel buffers on the output ports of network nodes for GS VCs, and buffers on both in- and output ports for BE VCs. Access to the link is granted by an arbitration scheme called asynchronous latency guarantees (ALG) described in section 3.1.2. In order to prevent congestion in the shared regions of the network, access to these is only granted when it is guaranteed that the flit will be able to leave the shared regions by entering the VC buffer in the neighbouring node. The means for making these guarantees are described in section 3.1.3.

Routing information for BE is contained in a header flit, while for GS it is stored in routing tables in the nodes. These tables are programmed through BE transactions as explained in [1].

BE traffic uses source routed wormhole routing based on a memory map. The NA looks up the route based on the address being accessed and sends out a flit with this routing information followed by the packeted transaction. In each flit, a bit is reserved to indicate the final flit. Flits from two different BE transactions can not be interleaved on a link.

GS transactions make use of a connection ID, which labels one of the VCs available to the IP-core. The flit is routed through the network using the routing tables in the network nodes. Conceptually, these tables contain information in the form of "incoming flits on port p_i, VC v_i are routed to port p_o, VC v_o." It is important to have the routing information set up properly before using GS connections, as there is otherwise no way of telling where the flits end up.

In the present implementation, there are seven GS VCs and one BE VC on each link. On the interface between the network adapter and the node, there are three GS VCs and one BE VC. The GS router is fully connected and non-blocking. The BE router has four input and output buffers, which are connected through a single latch, as illustrated in figure 3.2. The VCs associated with the NA do not need buffers, as transmissions on these channels do not occupy shared areas of the NoC.

The present implementation of MANGO uses 4-phase bundled data signals internally in the router. This - and other concepts of asynchronous circuits - are presented in chapter 5.

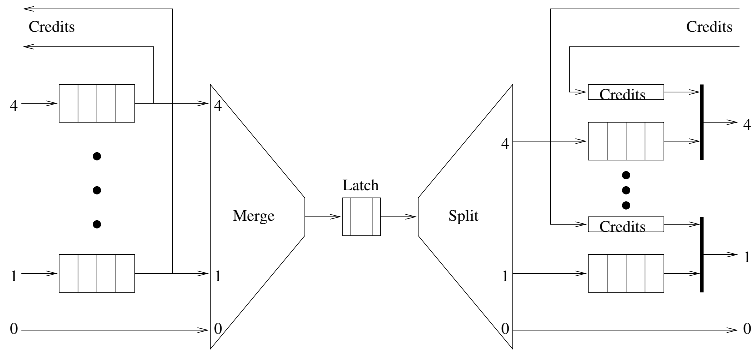


Figure 3.2: The MANGO BE router. Port 0 indicates the local port. Credits are used to ensure that transmitted flits will be accepted into the VC buffer in the next node in order to prevent stalling the link.

3.1.2 Arbitration Scheme

Access to links is granted based on a scheduling discipline called Asynchronous Latency Guarantees (ALG), which is described thoroughly in [5]. Each VC has a static priority in ALG, but with the restriction that no flit may wait for more than one flit on each higher priority level. With eight VCs trying to access a link, a flit on the lowest prioritised VC - the BE channel - may wait for at most seven flits to be transmitted before being given access.

ALG has the advantage over other scheduling disciplines that latency and bandwidth guarantees are decoupled. A low-latency low-bandwidth connection may reserve the highest priority channel and be given immediate access for its transmissions, as long as these transmissions are made rarely - ie. the low-bandwidth requirement. For a more detailed presentation and a proof of the correctness of the guarantees, refer to [5].

The implementation of ALG is illustrated in figure 3.3. It has two levels of latches in front of a merge component. The first level is the admission control, where flits on a given channel are stalled, if a lower-priority flit has already waited for another flit on the given channel. For example, a flit arrives on each of the first and second highest prioritised channels simultaneously. Both pass the admission control, and the flit on the first channel is transmitted. If another flit arrives on the first channel, it will be stalled in the admission control, until the flit on the second channel has been transmitted.

The second level is a static priority queue, which ensures that flits on the highest prioritised channel are transmitted before flits on lower prioritised channels, ie. this level along with the merge component contains some control structures beside the latches that enact these static priorities.

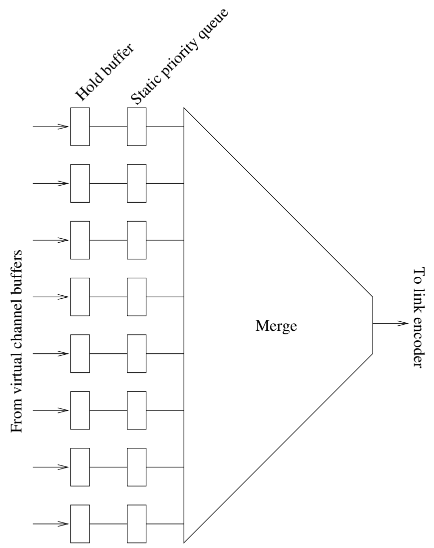


Figure 3.3: The ALG arbiter.

The merge component is constructed in such a way that all VCs are guaranteed a fair share of the bandwidth. The control part is shown in figure 3.4. The arbitration unit chooses an input that is allowed to propagate through it. If only one input is active, that input is selected. If both inputs are active, a choice is made, but the input that was not allowed to pass the arbiter at first is guaranteed to pass as the next one. For the asynchronous arbiters presented in [15], the choice is random. The asynchronous latches[1] provide pipelining of the merge component in order to improve throughput.

3.1.3 Preventing Blocking of Shared Areas

In order to make guarantees on latency and bandwidth, it is required that no flits stall on the link or in other shared areas, effectively blocking the NoC. These areas

[1] Asynchronous circuits are presented in chapter 5.

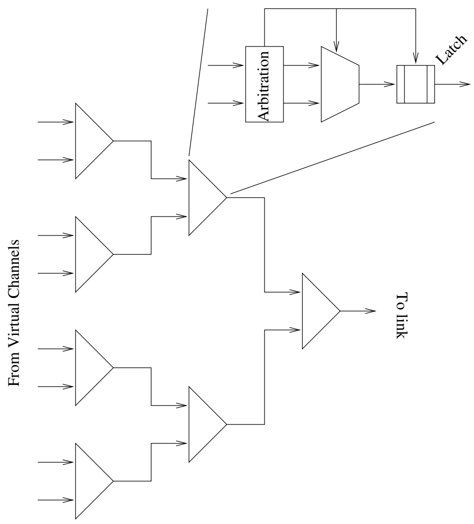


Figure 3.4: The control circuit in the merge in the ALG arbiter.

comprise the links and the routers. Two different means have been developed, and the current implementation of MANGO uses one for GS VCs and the other for BE VCs.

Lock- and Unlockboxes for GS

Only one flit may be in flight on a GS VC at any time. Figure 3.5 shows the lock- and unlockboxes used to guarantee that once a flit leaves the VC buffer, it will be admitted into the buffer in the neighbouring node. When a flit enters a lockbox, no further flits may enter. The flit is then granted access to the link by the ALG, transmitted over the link, and when it arrives at the VC buffer in the next node, it passes the unlockbox, which toggles the wire to the lockbox, unlocking it for a new flit to use.

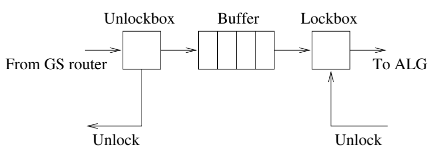


Figure 3.5: The MANGO GS VC buffer with lock- and unlockboxes. These boxes allow only one flit in flight from a VC at a time.

In the current implementation, the lock- and unlockboxes contain a latch each, while the buffer inbetween consists of a single latch. The wire connecting lock- and unlockboxes over a link only toggles once to indicate an unlock in order to reduce power consumption in this long wire.

Credit Based Transmission for BE

The scheme used for BE channels allows more flits in flight at a time by use of a credit based system. It is illustrated in figure 3.2. The output VC buffer has as many credits as the number of flits that may be stored in the input VC buffer. When a flit is sent, a credit is consumed. When this flit leaves the input buffer, the credit is restored.

3.2 Link Architecture

A property that the links in MANGO gain from being implemented as asynchronous circuits is that they may be pipelined as deeply as the designer may wish, without upsetting latency guarantees based on a transmission taking a certain number of clock-cycles. In [1] it is reported that the addition of two pipeline latches to the link only increases the forward latency[2] by 240 ps.

The current implementation of MANGO uses delay-insensitive coding on the links with a 2-phase protocol. This results in less switching activity and thereby less power consumption at the cost of double the number of wires. Another cost is in the conversion between the 4-phase bundled data protocol used in the router and the coding on the links.

[2] Forward latency and other elements of asynchronous circuits are introduced in chapter 5.

Chapter 4

Modeling Approaches

This chapter will describe general approaches to system modeling. The chapter will be concluded by a discussion and a decision of which approach to take for modeling MANGO.

4.1 Modeling System Communication

NoC models are either analytical or simulation based [7]. The following will concentrate on simulation based models, as analytical models do not allow the network designer to interchange parts of the model with parts of the actual network for evaluation of new or modified components.

Simulation based models can be divided into two classes: Structural and behavioural models. Structural models use models of the subsystems which are then tied together similarly to a structural RTL design. Likewise, a structural NoC model uses models of the major components which are then tied together. A behavioural model tries to capture the behaviour of the entire system at once in a single model or an abstract modeling framework.

In the following, behavioural and structural models will be described.

4.1.1 Behavioural Model

A general behavioural model is characterised by having the same behaviour as an RTL or similar description of what is being modeled. The behaviour may be timing accurate, but this is not a requirement. An example is a clocked micro processor, which can be modeled either as all instructions executing in a single cycle or at a cycle-accurate level, which includes pipeline stalls, branch delays and other specifics of processor design.

For a Network-on-Chip, the requirements to a behavioural model depends on the abstraction level at which the model is to be used. An application programmer may see the NoC as a multi-port black box, which allows communication between all ports. In this case, the model needs only act as a large crossbar switch: No conflicts between packets are considered, and communication is instantaneous or some constant delay. Such a model obviously executes very fast and is very simple to create, but it can be used for neither system architecture exploration nor network component development. This simple type of behavioural model is not considered further in this text.

A more detailed behavioural model - without the shortcomings of the simple one - makes use of one or multiple data structures which contain representations of the resources in the network. The granularity at which resources are represented must be fine enough to accurately model the behaviour of the network, but also coarse enough not to slow down the simulation speed unnecessarily. When a transaction is initiated, the resources used for the transaction are reserved for the required time intervals. If a resource is not available, arbitration identical to the one performed in the network must be done. The reservation and arbitration may be done all at once or one resource at a time. The disadvantage of all-at-once is that periods with heavy load on the network will lead to many changes in the reservations and arbitration results of existing transactions whenever a new transaction is initiated. However, during periods with light load, transactions may be considered only once by the model, as no conflicts will occur. The execution speed of the model varies according to the traffic patterns of the system, with heavy traffic leading to slower execution than a constant added time for each extra transaction, as all active transactions must be reevaluated for each new transaction. Reserving only one resource at a time yields a more constant execution speed, as a resource conflict only has local implications on the latency of the transactions involved. The delayed arrival at the succeeding resource is automatically achieved, as this resource is only reserved once the currently reserved resource has been released. Furthermore, no global rearbitration must be done in the event of a resource conflict as in the case of all-at-once reserving.

Using a behavioural model to examine the impact of new implementations of single network components - such as for example the switching structures in the node - is not easy, as they can not simply be “plugged into” the model. Rather, the behaviour - including timing - of the new implementation needs to be integrated into the model. The network designer thus has to determine how the new implementation behaves without using the model in order to insert this behaviour into the model. However, for trying out new arbitration or packeting schemes, a properly implemented behavioural model allows the network designer to experiment by making only small changes to the model.

An example of an abstract modeling framework that may be used to model the behaviour of a NoC is presented in the following.

ARTS

ARTS is a System-level MPSoC Simulation Framework developed at IMM DTU [10]. It can be used to model NoCs through the means of a behavioural model. The workings of ARTS is described in [11] and [12] and is summarised here to demonstrate what a behavioural model may look like.

ARTS is intended for system level simulation of multiprocessor System-on-Chip systems. Each process in the system is represented by a task with certain resource requirements. A processing element (PE) is represented by a real-time operating system (RTOS) model, which makes use of a scheduler, an allocator and a synchroniser. The scheduler models a real-time scheduling algorithm, while the allocator arbitrates resource access between tasks. The synchroniser is common for all PEs and is used to model the dependencies - including inter- and intra-process communication - between tasks. In this way, a task which needs to wait for data from another task is stalled until that other task has sent the data.

Communication in ARTS is modeled in a very similar manner. One task is used to model a transaction between one pair of PEs, requiring $O(n^2)$ communication tasks where n is the number of PEs. A communication task is then dependent on a processing task, representing some processing going on before data is transmitted. The receiving processing task is similarly dependent on the communication task, simulating that the task waits for the transmitted data. These dependencies are handled in the synchroniser.

The NoC model has its own allocator and scheduler, similarly to an RTOS model. The allocator has a resource database and arbitrates access to the resources in this database, effectively making it reflect the topology of the network. The scheduler "executes" the communication tasks, reflecting the protocol in the network [11].

The arbitration method in ARTS is first-come first-serve [12] and the routing is static, but provisions for implementing other arbitration and routing schemes exist [11].

ARTS provides a framework for fast investigation of entire SoCs, including both processing and communication. It is well suited for exploring the design space of different network topologies and application mappings. The network designer may also use ARTS to investigate arbitration, routing and packeting schemes, but examining the impact of new implementations of network components is not easily done for the same reasons as why it is not easily done in general behavioural models.

4.1.2 Structural Model

A structural model is comparable to hierarchical RTL models. Single components are modeled individually, and these components are then connected similarly to how RTL models are connected by signals or wires.

When an IP-core initiates a transaction, the first component model to receive the transaction performs the processing its comparative implementation does and delays the transaction for the duration of the processing. After this delay, the transaction is passed on to the next component which also processes and delays. This is done for all components on the communication path until the destination is reached. At this point the transaction is presented to the destination IP-core.

The main challenge of creating a structural model is having the components cooperate to accurately reflect the system being modeled. The component models are in fact small behavioural models, but they only capture the behaviour of the component and not that of the entire system.

The execution speed of a structural model is fairly constant with the traffic load. No global knowledge is present in the model, and therefore only local decisions are made. The magnitude is comparable to that of the behavioural model that reserves a single resource at a time. The behavioural model has some processing overhead in updating the data structures with resources, while the structural model relies on the simulation kernel for transferring data between components and managing delays, causing a processing overhead in the kernel rather than in the model.

System architecture exploration may be performed easily with a structural model. The network topology is created by connecting nodes and links, while standardised interfaces at the network adapters allows for easy movement of IP-cores between nodes, allowing exploration of the impact application mappings have on traffic patterns.

Trying out new packeting, routing and arbitration schemes is also quite easy with a structural model, as changes only need to be made in the affected components, while all other components are untouched. Inserting actual implementations of components into the model is also doable, but requires some conversion between the data representation used in the model and the physical data representation of an implementation. The complexity of this conversion depends on how closely the data representation in the model matches that of the implementation.

4.2 Conclusions

It has been argued that structural and behavioural models may be developed with roughly equal complexity and simulation speed. Both types of models adequately fulfil the requirement of being usable for exploring system architectures in a reasonable amount of time. Both may be used for examining the impact of changes in arbitration, routing and packeting schemes, but only the structural model allows the network designer to evaluate new implementations using the model.

The behavioural models - particularly the ARTS framework - allow a unified approach to modeling computation and communication while keeping the design of the two independent of each other. A structural model requires separate models of IP-cores to be obtained and attached to the network, causing a slight disadvantage in system exploration compared to behavioural models.

With regard to accurately modeling asynchronous circuits described in chapter 5, capturing the behaviour of these circuits which use distributed, local control circuits can be very difficult in a purely behavioural model. A structural model lends itself much better to modeling such control circuits, as the structure of the actual implementation is inherently reflected in the model.

Both types of models have advantages and disadvantages compared to each other, but as actual implementations of network components can only be inserted into a structural model and a purely behavioural model is unlikely to accurately capture the behaviour of asynchronous circuits, the model of MANGO is to be structural.

Chapter 5

Asynchronous Circuits

Asynchronous circuits [15] are circuits that make use of local handshakes for flow control rather than the global clock used in synchronous circuits. This chapter first gives an introduction to asynchronous circuits and then discusses how they may be modeled. The introduction to asynchronous circuits is only meant to cover the schemes used in the current implementation of MANGO. Topics not discussed include data validity schemes, push vs pull circuits and flow control structures beyond the basic C-element. A comprehensive guide to asynchronous circuits can be found in [15].

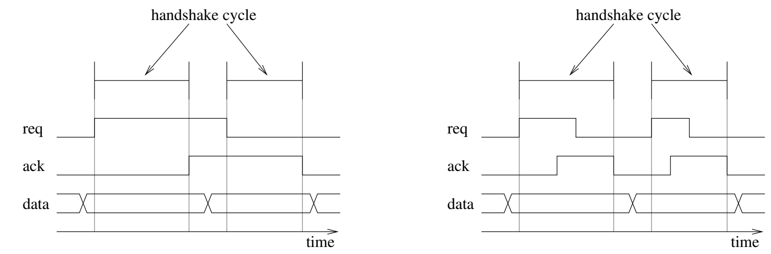
5.1 Introduction to Asynchronous Circuits

Two of the basic concepts of asynchronous circuits are handshake protocols and data encodings. Once these have been introduced, the basic building block of asynchronous circuits, the C-element, will be presented along with how to use it to create pipelines. Lastly, properties of these pipelines will be discussed.

5.1.1 Handshake Protocols

Because asynchronous circuits do not make use of a global clock as synchronous circuits do, one slow path does not restrict the speed of all other paths in the circuit. In other words, the concept from synchronous circuits of a critical path restricting the speed of the entire circuit does not exist in asynchronous circuits. Every path operates at its full speed potential, due to the local handshakes.

Two common handshake protocols are 2-phase and 4-phase handshakes. Both make use of request and acknowledge signals, with 2-phase requiring one transition of each signal for a complete handshake while 4-phase requires two transitions of each signal. This is illustrated in figure 5.1(b) for 4-phase handshakes and in figure 5.1(a) for 2-phase handshakes. In either protocol, handshakes may not overlap.



(a) (b)

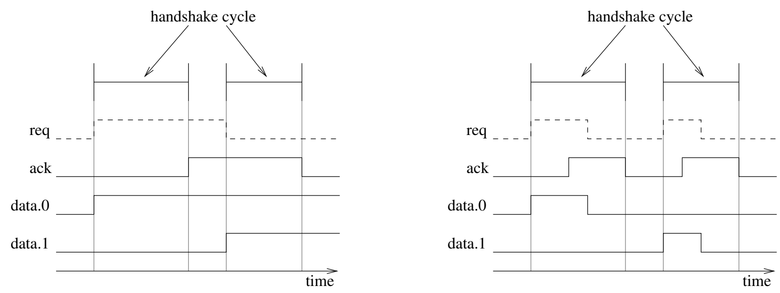
Figure 5.1: Handshake protocols used in asynchronous circuits. 5.1(a) shows the 2-phase protocol and 5.1(b) shows the 4-phase protocol.

5.1.2 Encodings

Two encoding schemes used in asynchronous circuits are bundled data encoding and dual-rail, delay insensitive or one-hot encoding. Both of these schemes are used in the current implementation of MANGO.

In the bundled data encoding, data and handshakes are carried on separate signals. The request signal needs to be delayed by at least the maximum delay the data may experience. The acknowledge signal does not need to be delayed, as it does not indicate data validity as the request signal does. This encoding is illustrated in figure 5.1.

In dual-rail or one-hot encoded circuits, requests are embedded in the data. Two wires are used for each bit, one wire signifying ’0’ and the other ’1’. Figure 5.2 shows the handshake phases of the dual-rail encoding. In the 4-phase protocol, the data value is indicated by signal levels, while for 2-phase, the value is indicated by transitions.



(a) (b)

Figure 5.2: 5.2(a): 2-phase dual-rail handshakes. 5.2(b): 4-phase dual-rail handshakes. In both figures, a ’0’ is transmitted first and then a ’1’. The request signal is not physically present, but included to clearly indicate the phases of the handshakes.

5.1.3 Basic Building Blocks

This section will introduce the Muller C-element and show how to use it to create asynchronous pipelines. Concepts of these pipelines will then be introduced. These concepts are to be used in the discussion on modeling asynchronous circuits.

The Muller C-element

The basic element in asynchronous circuits is the Muller C-element shown in figure 5.3. The output only changes when both inputs have identical values. The feedback inverter shown in figure 5.3(b) is only necessary in technologies where leakage currents are a concern when the output is connected to neither source nor ground.

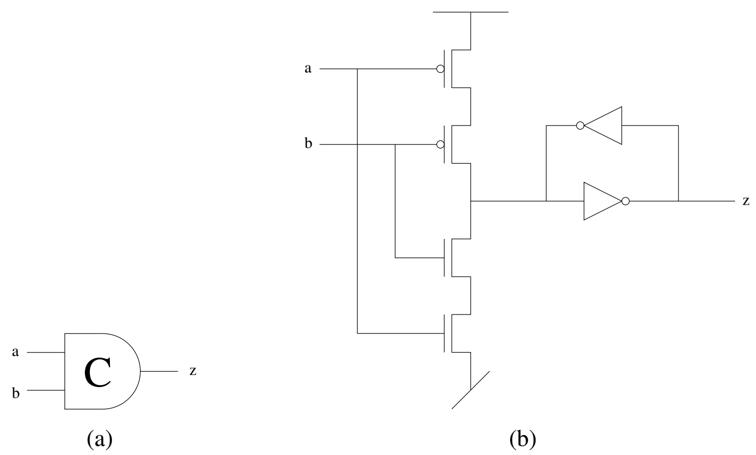


Figure 5.3: 5.3(a): The symbol denoting the basic building block of asynchronous circuits, the Muller C-element. 5.3(b): An implementation of the Muller C-element. The functionality is to only change the output value when both input values have changed.

Latches

The C-element can be used as a controller for a latch, as shown in figure 5.4(a). The symbol for an asynchronous latch is shown in figure 5.4(b). This description illustrates the functionality of an asynchronous latch in a 4-phase bundled data circuit.

Assuming as an initial state of the C-element that all in- and outputs are '0', the latch is transparent. Using the signal names of figure 5.4(a), let a be the input request signal, req_{in}, b the inverted acknowledge from the output, ack_{out} and z the output request, req_{out} and the acknowledge back to the input ack_{in}. When a request arrives on req_{in}, then ack_{in}, req_{out} and en are all asserted. When ack_{out} has been asserted, the data has been processed by the output and the latch no longer needs to hold the data. The output of the C-element is thus free to return to zero, which requires that

req_{in} is deasserted, which may happen at any time relative to ack_{out} being asserted - both earlier, simultaneously or later.

When using the asynchronous latch symbol in figure 5.4(b) in schematics, the handshake signals are rarely drawn separately. A line connecting two latches is simply taken to mean both data and handshake signals.

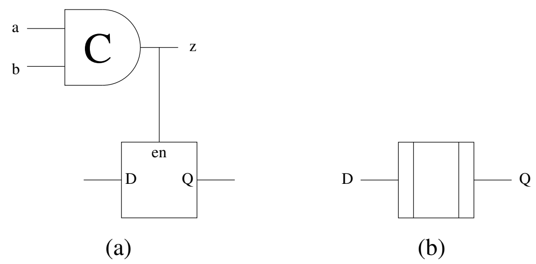


Figure 5.4: 5.4(a): A schematic of an asynchronous latch using a C-element as a controller. The latch holds data when the enable port is ’1’. 5.4(b): The symbol used for an asynchronous latch. The handshake signals are not explicitly drawn.

Pipelines

C-elements can be connected as shown in figure 5.5(a) to function as the controller of a pipeline in a 4-phase bundled data circuit. The output of each C-element goes to the enable-port on a latch as shown in figure 5.4(a). The corresponding schematic using the symbol for an asynchronous latch is shown in figure 5.5(b).

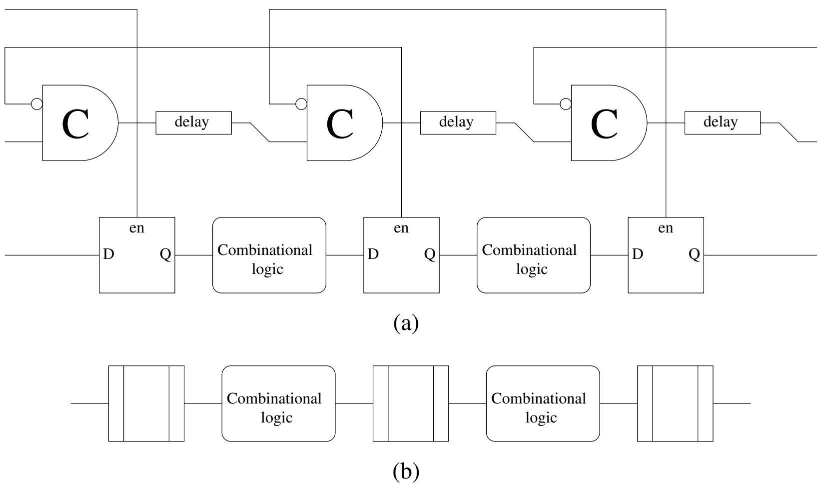


Figure 5.5: 5.5(a): An asynchronous pipeline with handshake signals exposed. 5.5(b): The same asynchronous pipeline using the schematic symbol for a latch in figure 5.4(b)

The setup and hold times of the latches must not be violated, which requires the request signals to be delayed by at least the slowest path through the combinational logic. This is accomplished by a delay element, which may be implemented in any manner, as long as the delay is "long enough" and no glitches occur on the output of the delay element.

5.1.4 Pipeline Concepts

Tokens And Bubbles

A common concept used in describing pipelines is tokens and bubbles [15]. These indicate the state and contents of an asynchronous latch, with a valid token indicating valid data and an empty token indicating the return-to-zero part of the handshake. Bubbles indicate that the latch is able to propagate a token of either type. Thus tokens flow forward while bubbles flow backward, feeding the flow of tokens. If all latches in the pipeline are filled with tokens, data will not be able to propagate until a bubble has been inserted at the end of the pipeline. For a steady flow of data, a balance between tokens and bubbles must thus be established. Figure 5.6 shows a snapshot of a pipeline described by latches containing tokens and bubbles. Valid and empty tokens are represented by the letters 'V' and 'E' respectively. Tokens are distinguished from bubbles by tokens having a circle around their descriptive letter.

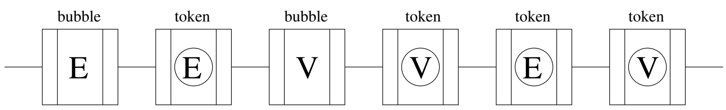


Figure 5.6: Part of an asynchronous pipeline with tokens and bubbles marked.

A consequence of having both valid and empty tokens is that only every other latch may hold valid data, but this is no different than the case of synchronous circuits, where two latches are used to make a flip-flop which stores a single data element. However, more elaborate latch controllers called semi-decoupled and fully-decoupled controllers exist that allow valid tokens in all latches when the pipeline is full [15].

When dealing with 2-phase protocols, no empty tokens are used, as there is no return-to-zero part of the handshake.

Forward And Reverse Latencies

A metric used for the timing of handshakes is forward and reverse latencies. The forward latency is the time it takes a request to arrive at the next pipeline latch, while the reverse latency is the time an acknowledge takes to arrive at the previous latch. The latencies of both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions of both request and acknowledge signals are used, even though the latencies of both transitions are normally identical.

The $0 \rightarrow 1$ transition is the latency of a valid token or bubble while the $1 \rightarrow 0$ transition is the latency of an empty token or bubble, provided that the other handshake signal is "in place" when the one considered arrives, eg. the value of the forward valid latency assumes that the acknowledge from the succeeding stage is '0' when the request arrives [15].

The symbols used to denote these latencies are $L_{f,V}$, $L_{f,E}$, $L_{r,V}$ and $L_{r,E}$ for forward valid and empty and reverse valid and empty latencies respectively.

5.2 Modeling Asynchronous Circuits

A typical timing accurate model of a synchronous circuit goes along the lines of "wait for the clock to tick and then do these computations." However, asynchronous circuits are without the global clock of synchronous circuits, requiring modelers to find some other means of generating timing accurate models.

5.2.1 Handshake Level Modeling

The modeling level comparable to the in-between-clocks level for synchronous circuits, is the handshake level. [3] describes a library for simulation at this level. As in RTL models of synchronous circuits, the handshake level modeling uses latches and combinational logic. The addition over RTL models is explicit delays for the handshake signals.

A fully timing accurate model at this level is quite easy to develop, as a latch can be fully characterised by the forward and reverse latencies of the pipeline stage. Even though these values assume that the previous part of the handshake is completed when the signal triggering the next part arrives, these latencies are a good measure for the delay between the arrival of any signal triggering a transition of the C-element and this transition arriving at the neighbouring C-elements in the pipeline.

While the number of signals is reduced considerably compared to a netlist of standard cells, at least four signal transitions happen for each latch: One for each transition on an output for two outputs each performing two transitions. Each of these transitions needs to be handled by the simulation engine, each causing a slightly slower simulation execution. However, replacing components in the model with those from the current implementation of MANGO is quite easy, as the model and MANGO components have nearly identical module declarations in Verilog terminology.

For simulation purposes, this modeling level is equivalent to considering tokens and bubbles. Although all the details of the handshake are not explicitly modeled by transitions of request and acknowledge when considering token and bubble flow, the number of events[1] to be handled by the simulation engine is of the same magnitude as when fully modeling the handshake.

[1]By an event is meant a condition that the simulation engine needs to react to. This is for example to trigger all processes that are sensitive to changes in a signal when said signal makes a transition.

5.2.2 Higher Level Modeling

An even higher level of abstraction in modeling asynchronous circuits requires that the details of handshakes are abstracted away. This section will explore how this may be possible.

Non-Stalling Pipelines

The main issue in correctly capturing the behaviour of asynchronous circuits in high level models is not the forward flow of data, but rather the reverse flow of bubbles needed to enable the data flow. A non-stalling pipeline is characterised by the destination always accepting data immediately when it arrives, as is the case of the shared areas of MANGO, ie. the link and switching structures in the nodes.

At initialisation, the pipeline is full of bubbles. When a valid token arrives at such a pipeline, that token will have passed through the pipeline after $\sum_i L^i_{f,V}$, where i denotes a pipeline stage - ie. the sum of the forward latencies of all pipeline stages. If the maximum rate at which tokens may be injected into the pipeline is no faster than the slowest pipeline stage is able to accept new tokens, a token will never be stalled inside the pipeline by another token, ie. the pipeline performance is constrained by the forward flow of tokens. This is true, as the empty token will never catch up with the valid token, as it will never wait longer in a stage than the difference in insertion times between the valid and the empty token.

In this case, the behaviour of a non-stalling pipeline is fully characterised by the forward latency of the entire pipeline and the maximum rate of injection of valid tokens. There is no need to consider empty tokens as valid tokens simply may enter the pipeline half as often as tokens in general. Also, it is not necessary to consider bubbles, as “enough” exist in the pipeline in order to avoid tokens waiting for each other as described above. An arbitrarily long pipeline may thus be reduced to a FIFO with a constant delay from input to output and a restriction on the frequency of insertion of new elements. This is illustrated in figure 5.7 which shows a pipeline with an injection rate of one token each two “time units” - for simplicity a “time unit” is a new state of the pipeline - and a latency of each pipeline stage of one time unit.

Now, consider the case where one of the pipeline stages is slow compared to the rate at which tokens are injected into the pipeline. As tokens will be injected faster than they can pass the slow pipeline stage, they will accumulate in the pipeline up to the bottleneck stage. In a “steady state”, the pipeline performance after this stage will thus be constrained by the forward flow of tokens as before, but up to the bottleneck, performance will be limited by the reverse flow of bubbles. In this steady state, there will be a constant forward latency through the pipeline as well as a constant injection rate of new tokens. However, until the steady state is reached, the forward latency and injection rate vary and are not easily modeled. This situation is illustrated in figure 5.8 where the injection rate is one token every two time units as before, but the middle pipeline stage has a latency of three time units. In this figure, the initial forward latency and injection rate are seven time units and one token every two time

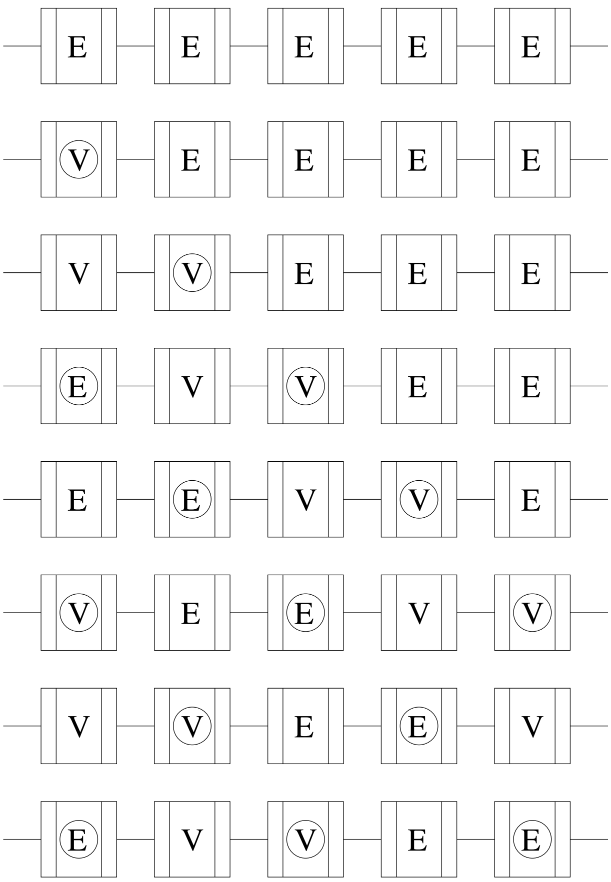


Figure 5.7: Execution of an asynchronous pipeline. Time progresses from top to bottom. New tokens are inserted every two time units while all pipeline stages have a latency of one time unit, making the latency of the entire pipeline five time units.

units respectively, while the steady state forward latency and injection rate are eleven time units and one token every four time units respectively.

Depending on the pattern of use of the pipeline in the modeled system, either the initial or the steady state parameters may provide useful models. In case tokens are rarely injected, the initial parameters may provide an accurate model, but in case tokens are injected at the maximum rate, the steady state parameters accurately model the behaviour of the pipeline except for the first few tokens, which have a longer forward latency than would really be experienced. If nothing is known about the pattern of use of the pipeline, a compromise might be used or a more detailed model, such as fully modeling the handshakes.

This type of model only considers what happens at the two ends of the pipeline. There is no information concerning the internals of the pipeline - not even the depth of it. Realistic injection rates and forward latencies ensure that the model pipeline holds no more tokens than an implementation would be able to. This makes this type of

Figure 5.8: Execution of an asynchronous pipeline with a bottleneck. All pipeline stages have a latency of one time unit, except for the middle pipeline stage which has a latency of three time units, which causes variations in forward latency and injection rate until a steady state is reached.

model very fast executing, as the number of events the simulation engine must handle is reduced even further compared to the handshake level model. However, replacing components in the model with actual implementations becomes more complicated, as a translation between handshake signals and the model is necessary.

5.3 Conclusions

The two approaches to modeling asynchronous circuits examined in this chapter have each their advantages and disadvantages with regard to the purposes of modeling MANGO. A handshake level model allows very easy replacement of model components with actual implementations of the same components, whereas a higher level model needs some translation between handshake signals and the model. However, the higher level model triggers very few simulation events, the number being independent of the length of the asynchronous pipelines. A handshake model triggers more events per token, and this number of events increases linearly with the depth of the pipeline.

A faster executing model will have lasting benefits with regard to exploring different network topologies and application mappings. The added effort of converting between a high level model and handshake signals when replacing model components with actual implementations is better justified than a general reduction in simulation speed when exploring system designs.

The model to be developed is based on the higher level modeling of asynchronous circuits.

Chapter 6

Modeling MANGO

This chapter will describe the design of a structural, high level model of the current implementation of MANGO. As previously mentioned, a high level model does not allow effortless replacement of model components with actual implementations. The focus of this section is on correct functionality and accurate timing of the model, while interchanging model components and actual implementations are defered to an example of such in chapter 7.

6.1 Functionality

A functionally accurate model of MANGO - or any other Network-on-Chip - is characterised by transporting transactions between IP-cores. How data is represented and whether transports are timed is irrelevant to a functional model. Specifically, the data coding - 4-phase bundled data or 2-phase dual rail - may or may not be reflected in the model, and has no influence in terms of functionality. However, replacing model components with actual implementations require a conversion between the model data representation and the coding used in the actual implementations. This is similar to the requirement of converting between handshake signals and the model, and both conversions may be done at the same point.

Most of the components in MANGO have a rather simple functionality. Models of the major components in MANGO will be described in this section.

6.1.1 Link

The links - including link encoders and decoders - are essentially FIFO buffers. The link acts as an asynchronous pipeline without combinational logic between pipeline stages. The only combinational logic is at the ends of the link, where flits are encoded and decoded for transmission. As this coding is of no consequence to a functional model, it may be omitted completely, but it may also be included if this is advantageous to the implementation of the model.

6.1.2 Node

As described in chapter 3, the node is comprised of VC buffers, BE and GS routers and ALG arbiters. Each of these will be dealt with individually in the following.

Virtual Channel Buffers

A different type of buffer is used for each of GS and BE channels. A common characteristic of these two buffers is that a mechanism is in place for either type that guarantees that when a flit is transmitted, it can be stored in the destination buffer - it may not stall in the shared areas of the NoC. This mechanism may be taken advantage of in a model, as the node knows that all incoming flits may proceed directly to their destination buffer when they arrive. This supplants backwards to the link that also knows that any flit it receives will be accepted by the node. There is thus no need for an indication backwards of whether a flit may be accepted or not in these parts of the model. Otherwise, both types of buffer acts like FIFOs. The following will deal with each VC type individually.

The GS buffers consist of three decoupled latches, meaning three flits may be stored in the buffer at a time. One latch is the lockbox and another latch is the unlockbox. A GS buffer may not transmit a flit before receiving an unlock signal from the destination buffer in the neighbouring node. This unlock is generated when a flit leaves the unlockbox, and for GS this is the mechanism mentioned above that prevents stalls in the shared areas of the NoC. A GS VC buffer thus needs unlock in- and outputs and data in- and outputs, and no signals for indicating whether a flit may be received or not.

The BE channels use both in- and output buffering in the node. Four flits may be stored in each buffer, and a credit based system is used to ensure no more flits are sent than may be received. An input BE buffer does not need to know how deep it is, as long as the output buffer in the neighbouring node knows how many credits are available. For the output buffers, an indication backwards is needed to the router in order to prevent too many flits being sent to them. Furthermore, a similar indication is needed between the arbiter and output buffer. This will be described below when the arbiter is described.

Routers

The routers in the nodes are controlled by the incoming flits and transport these flits to their destination VC buffer. The GS and BE routers are quite different as described in chapter 3.

Flits can not stall in the GS router which can simply pass an incoming flit on to its destination. The GS router is non-blocking, which means practically no effort is needed in modeling it, as flits make no interactions in the GS router. Furthermore, no two flits from different inputs may be routed to the same output, as channel reuse is not allowed in MANGO.

The current BE router is problematic in that all flits must pass through a single latch creating additional dependencies between VCs. This leads to some uncovered deadlock problems that require a more restrictive routing scheme than the xy-routing scheme described in section 2.1.2. Efforts to replace the current BE router are underway, but not completed. For this reason, no detailed model of the BE router will be created as part of this work. Furthermore, as the router must contain some means of preventing interleaving of flits from different BE input channels on one output channel, which requires interaction between the BE router and BE VC buffers, the BE VC buffers will not be fully implemented either.

ALG Arbiter

The functionality of the ALG is described in both [5] and section 3.1.2. One model of this arbitration scheme closely resembles the implementation. It uses two levels of eight latches, one for each channel. The first level is the admission control, while the second is the static priority queue (SPQ). When a flit moves from the admission control to the SPQ, a list of which other channels the current channel must wait for before another flit may enter the SPQ is updated.

In order to determine which flit in the SPQ to transmit first, a model of the control path of the merge shown in figure 3.4 should be made. This is simply a binary tree, where flits enter at the leaf corresponding to the VC they are being transmitted on and progress upwards until they reach the root of the tree. At this point, the flit has passed through the merge and may be passed on to the link.

The model of the ALG takes advantage of the fact that at most one flit is in flight at a time on each GS VC. There will thus at most be one flit in the arbiter on each channel, removing the need for an indication backward of readiness to accept another flit. Similarly, as flits can not stall on the link, there is no need for such an indication from the link back to the arbiter. However, such an indication is needed for BE VCs, as these may have more flits in flight at a time.

6.2 Timing

The service guarantees - and thereby the behaviour - of MANGO are based fully on the arbitration scheme employed [6]. Thus, an accurate model may be created by ensuring that flits arrive at the correct time at the arbiter and that flits are passed accurately through the arbiter. This section will add timing to the functional model designed above in order to fulfil these requirements as closely as possible.

One of the goals for this model is to be fast executing, which means that the number of simulation events must be minimised. As each delay or delayed signal assignment[1] triggers an event, the number of single delays should be minimised. This

[1] A delayed signal assignment is for example `Z <= A and B after 5 ns;` while a delay might be `wait 5 ns;` in VHDL.

means that optimisations are made across multiple components and components are only discussed individually when some special cases are present.

Assumptions

A number of assumptions are made concerning the timing behaviour of different parts of MANGO. These assumptions are presented and justified here.

The first assumption is that all paths through the GS router have symmetrical delays, which means that the entire area between the output of the arbiter and the VCs in the neighbouring node may be seen as an asynchronous pipeline. It is assumed that the pipeline is constrained by the forward flow of flits, ie. it may be accurately described by a single forward latency and a maximum rate of injection of new flits as described in section 5.2.2. This is supported by [5] that states that the forward latency in the shared areas is constant. The only variation in the time flits take moving between two VC buffers is caused by being stalled in the arbiter, waiting to be granted access to the shared areas.

The second assumption is that the forward latency through the arbiter is constant for all channels. Furthermore, it is assumed that a flit does not propagate beyond the SPQ before being granted access to the link. Thus, the constant forward latency is applied to the flit from the moment it is granted access to the link. While this is not entirely consistent with the implementation of the arbiter presented in [6], the actual deviation should be minimal. Furthermore, the guarantees provided by MANGO assume worst case latencies, which are the same for all VCs through the arbiter.

The third assumption is that the arbiter is ready to accept a new flit when it arrives, ie. the handshake is in its initial state - request and acknowledge are '0' - when the flit arrives. For GS connections this is realistic as the previous flit must first propagate through the shared areas and through the unlockbox in the VC buffer in the neighbouring node, and then the unlock signal must propagate back across the link. The input of the arbiter has all this time to complete a handshake. As completing a handshake only involves propagation through a single C-element, this is more than enough time, making the assumption very realistic.

The fourth assumption is very similar to the third one, in that it states that the unlockbox is ready to accept a new flit when it arrives. As there is a significant amount of time between flits similarly to above, this assumption is realistic.

Merging Delays

Using these assumptions, the general model may now be described. In order to minimise the number of simulation events, the delays through the shared areas are merged into a single pipeline model. Thus, the model of the routers in the nodes is delay-less, while the actual delay through these is contained in the model of the link. In order to further reduce the number of simulation events, the forward latency through both the VC buffers and the arbiters may also be merged into the delay on the link. In order to realise that this creates a realistic timing model, consider figure

6.1. This figure shows a model of the communication on a VC between two nodes. The single delay used on the link in the model covers the area from just before one arbiter to just before the next one. A number of different cases will now be examined.

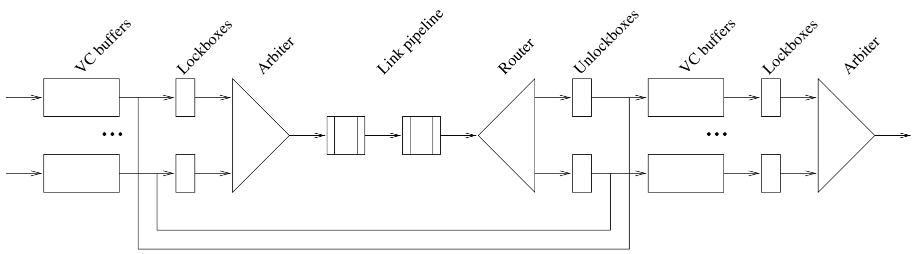


Figure 6.1: A model of the communication between two nodes. The entire delay is contained in the link, while all other parts of the model are delay-less.

In the first case, there are no flits already present in the parts considered in the figure. When a flit arrives at the VC buffer on the left, it is instantly moved to the arbiter, which grants it access immediately. It then enters the link, where it is delayed for the aforementioned amount of time. After this time has passed by, the flit arrives at the router and is instantly passed on to the destination VC buffer, where it is passed on to the arbiter. As the delay on the link encompasses everything between the first lockbox and the second arbiter, the flit arrives on time at the second arbiter. However, the unlock signal caused by passing the unlockbox is generated "too late" as the flit has already passed the buffer and the unlockbox when it is generated. This can be rectified by shortening the delay on the unlock propagation across the link by a similar amount of time, causing the unlock to arrive[2] on time, assuming the unlock propagation is longer than the time to be subtracted from it. This is a reasonable assumption, because the unlock signal both needs to pass through a router and across the link. The timing of a transmission of a single flit is accurately modeled in this case.

Transmission of multiple flits on a single VC requires a minor modification to the design above. If the second flit arrives after the lockbox has been unlocked, the situation is as above and everything is fine. However, if it arrives before the lockbox has been unlocked, the propagation from VC buffer to arbiter happens instantly, rather than taking the time this propagation does in the actual implementation. Thus, the flit arrives too early at the arbiter compared to the implementation. This may be rectified by delaying the unlock by the forward latency of the lockbox. Now, the unlock arrives later in the model than it does in the implementation, but still, the timing of flits is correct. In order to realise this, observe figure 6.2 which shows wavetrace-like illustrations of the sequence and timing of events around the lockbox. The dataA and dataZ signals represent the positions just before and just after the first lockbox in figure 6.1 respectively. The forward latency of the lockbox is arbitrarily assumed to

[2] By the arrival time of the unlock is used as a reference the time at which the lockbox is able to accept a new flit, and not the time at which the lockbox starts reacting to the unlock signal.

be two time steps in the figure. The actual value is of no consequence to the design. In all the figures, the lockbox starts out locked, and the three topmost signals indicate the situation in MANGO while the other signals indicate the situation in the model.

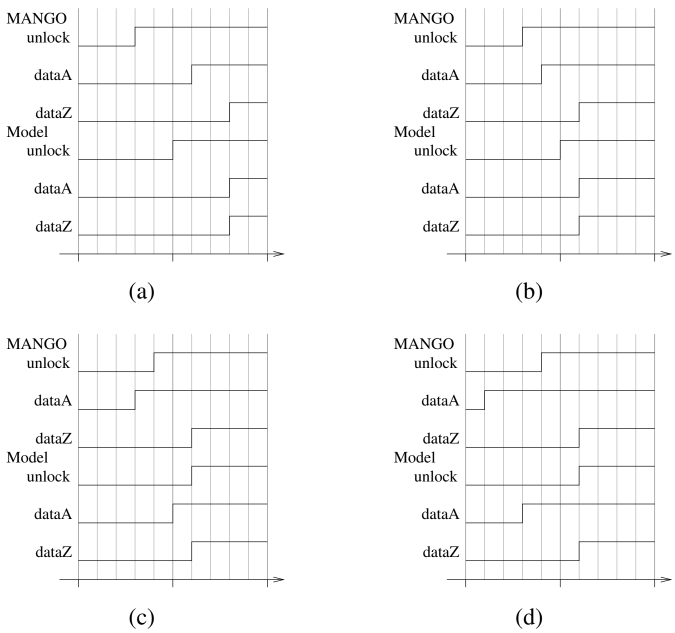


Figure 6.2: Wavetraces showing the timing around the leftmost unlockbox in figure 6.1.

In figure 6.2(a), the lockbox is first unlocked, and a flit arrives a "long" time afterwards. This is similar to the initial situation, as the system considered here is back in its initial state before the flit arrives. In MANGO, the flit needs two time steps to propagate through the lockbox, while in the model, the flit's arrival at the lockbox is delayed, but a delay-less lockbox ensures that the flit is produced at the output of the lockbox at the correct time. Also notice that the unlock signal in the model is delayed by the forward latency of the lockbox compared to MANGO as discussed above.

In figure 6.2(b), the flit arrives just after the lockbox has been unlocked. Again, the flit takes some time to propagate through the lockbox in MANGO, while its arrival is delayed in the model, such that it is output at the same time in both MANGO and the model. If the lockbox is unlocked and the flit arrives within a very short time of each other, it has no effect on this timing due to the definition of the time when the unlock arrives - which is the same as the time when the lockbox is unlocked - made above.

In figure 6.2(c), the flit arrives just before the lockbox is unlocked. In MANGO, the flit is at the output of the lockbox two time steps after it has been unlocked. In the model, the flit arrives at the input to the lockbox at the same time relative to the unlock as in MANGO. However, in the model the flit propagates through the lockbox in zero time when the unlock arrives, allowing it to reach the output at the same time as in MANGO.

In figure 6.2(d), the flit arrives a “long” time before the lockbox is unlocked. Again, the flit spents some time propagating through the lockbox in MANGO, while in the model, the flit is instantly propagated once the unlock arrives. Both MANGO and the model output the flit from the lockbox at the same time. In all four cases, the timing is seen to be accurate.

It has been shown that by merging all forward latencies from arbiter input to arbiter input into one single latency and by making the latency of an unlock that of the time it takes from the generation of an unlock until the receiving lockbox is ready to accept a new flit subtracted the forward latency of a VC buffer and a lockbox and added the forward latency of a lockbox back again, a model that has flits arrive at the correct time at the arbiter can be made. This model is unaffected by the time spent in the arbiter waiting for access to the link, as extra time spent here simply results in the lockbox being unlocked at a later point in time.

Network Adapter

When a flit is heading to the NA rather that a VC buffer, the forward latency is most likely different and somehow this difference in delay must be modeled. Whether it is done by allowing variable delays on the link or using a short delay for all flits followed by the remaining delay for those flits that require an additional delay is decided in the implementation of the model.

For data entering a node from the NA, this design can also be used, as this part of MANGO is functionally identical with a lockbox preventing too many flits being sent. The only difference is that flits from the NA have a much shorter delay to their destination VC buffer than flits being transmitted over a link. Thus, the exact same construct with a single delay may be used to generate the desired timing behaviour.

Arbiter

The requirements for an accurate timing model stated at the beginning of this section were that flits arrive at the correct time at the arbiter and that flits are passed accurately through the arbiter. The first requirement has been fulfilled by the timing model described above, while a functionally accurate arbiter was described in section 6.1.2. Under the assumptions made at the start of this section, the delay through the arbiter is constant. This constant delay has been merged into the single delay in the link, requiring the only timing in the arbiter to be a constant delay between granting flits access to the link - the rate of injection. Even though flits do not arrive at the merge in the arbiter at the correct absolute time, they do arrive at the correct time relative to each other due to the constant forward latency through the admission control and the static priority queue which is the same for all VCs.

One element of the arbiter which is impossible to model such that the behaviour is identical to what would be seen in a manufactured chip is the arbitration unit seen in figure 3.4. This arbitration unit makes a random choice if two flits arrive at the same time as described in section 3.1.2. How this choice is implemented in the model is irrelevant as long as no VC is favored.

Chapter 7

The Model

This chapter will present the implementation of the model created as part of this work. It follows mostly the design created in the previous chapter. As a model of the network adapter has not been created, the actual implementation of the NA is used instead. This also provides some insight into the issues associated with using actual implementations of components within the model.

First, a choice of modeling language is made, then the implementation of the model is described and lastly the inclusion of the actual implementation of the NA in the model is described.

7.1 Choice of Modeling Language

Under the requirement to be able to co-simulate the model with the real network, three modeling languages are available: VHDL, Verilog and SystemC. The current implementation of MANGO is written as netlists of standard cells in Verilog. The environment to be used for simulation is Mentor Graphics' Modelsim [13] as it supports co-simulation of these languages.

VHDL and Verilog are straight-forward to co-simulate in Modelsim. Component and entity declarations can be moved fairly freely between the two, as long as no user-defined types are used.

A model in Verilog can obviously be co-simulated with the current implementation of MANGO. However, Verilog does not allow the user to define types, requiring all modeling interfaces to be at the bit-level, ie. rather than passing an OCP request type, it is necessary to pass all the fields of a request as appropriately wide vectors.

A SystemC model can be co-simulated with the Verilog description of MANGO fairly easily. Wrappers must be used in order to convert to and from user-defined types, but the model itself may use any abstraction of for example OCP requests and flits. Furthermore, inheritance in C++ allows for easy replacement of component types in the model, which is not possible in either VHDL or Verilog.

SystemC is chosen as the modeling language, due to the ease with which abstractions can be made as well as the easy replacement of component types. The execution speed of a SystemC model should also be faster than that of other models in other languages, as SystemC is compiled to native machine code. The following will give a brief introduction to SystemC and general methods for getting fast execution times of SystemC models.

7.1.1 Introduction to SystemC

SystemC is a class library for C++ and a reference simulator is freely downloadable at http://www.systemc.org. The basic terminology of SystemC is as follows: A design unit is called a module, the contents of modules are defined in methods or threads and connections between modules are made on ports. This will be elaborated below.

Modules and Interfaces

High-level modeling in SystemC operates with interfaces and modules, which are both defined as classes. In order to avoid confusion with bus interfaces such as OCP, SystemC interfaces will be denoted by their class name, sc_interface. General sc_interface and module classes are provided by SystemC and user defined sc_interfaces and modules must inherit from these.

An sc_interface is an abstract definition of the functions a module implementing that sc_interface must provide. A module implements an sc_interface by inheriting from it. These concepts are illustrated in figure 7.1. In this figure, the user defined abstract class link_tx_if inherits from sc_interface, and the class called link inherits from both link_tx_if and sc_module. The link_tx_if class defines the functions that must be implemented by a link class that may be used for transmitting flits.

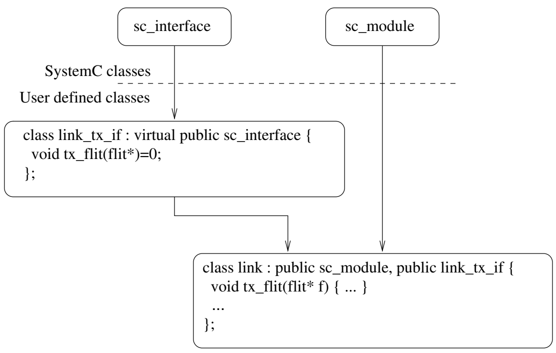


Figure 7.1: SystemC interfaces and modules. The link_tx_if sc_interface defines the functions the link module must implement.

Methods and Threads

SystemC has three means of defining the contents of a module. These are methods, threads and cthreads. Cthreads are special-case threads, which are only sensitive to a clock signal. Methods and threads will be described in the following.

Methods

A method is a state-less process. A method is defined as a function in the module such as the tx_flit function in figure 7.1, and is made sensitive to a list of events in the module constructor. These may be explicit sc_event objects or events on for example input ports on the module. The sensitivity list may be dynamically updated, if such is required. It is also possible to temporarily disable the sensitivity list and instruct the simulation engine to trigger the method again after a certain amount of time. Whenever an event in the sensitivity list is triggered, the method is executed, and due to the state-less nature of methods, it is not possible to wait for a certain amount of time or for an event to occur in the middle of execution. Thus, when the sensitivity list is dynamically updated or temporarily replaced by a fixed time before triggering the method again, execution continues until a return statement is encountered.

Methods are light-weight processes due to their lack of state. This means low memory requirements and fast execution, as they simply need to be called by the simulation kernel. If a method needs to have memory between different executions, it can be achieved by storing the required data in members of the module.

Threads

Threads are processes which are able to retain state. Threads are defined just like methods. A thread's execution is only started once, and if the end of execution is reached, the thread may not start again. Looping behaviour thus requires an explicit loop in the function code. Threads may also have a sensitivity list which may also be updated dynamically. Execution can be suspended for a specific amount of time or until an event in the sensitivity list or a specific event occurs. Threads are heavier than methods due to the requirement to store all data used by the thread as well as a pointer to the present point of execution.

Module Ports

SystemC provides four types of ports: In-, out- and inout-ports and simply ports, which will be called sc_ports in order to avoid confusion. The first three are comparable to the port types of identical names in other modeling languages, while an sc_port is to be used at higher level of abstraction than provided by other languages. The following will deal exclusively with these high-level sc_ports.

An sc_port is a templated class that may have any class type as template argument. Typically, an sc_interface will be given as template argument, as these define the functions a module of the given type must implement. Any such module can be bound to the sc_port during design elaboration. It is thus possible to change what type of module is actually used in a top-level module, which connects lower-level modules. For example, in a system consisting of a producer, a consumer and a FIFO, multiple implementations of the FIFO may exist such as a high-level model, an RTL model or even a netlist of standard cells. If all three implementations implement the same interface, they may be interchanged with absolutely no changes made to the producer or consumer. It is also possible for the different implementations to have different behaviour - e.g. blocking or non-blocking writes - but extreme care must be taken when using implementations with different behaviour, as the producer or consumer may rely on a specific behaviour.

7.1.2 Simulation Performance

A number of modeling techniques for high performance simulation are presented in chapter 8.5 of [9]. These include using transactions on sc_ports and not signals for transporting data between modules. When the functions defined in the sc_interface are called, both control and all the data in the transaction are transferred to the receiving module without involving the simulation engine. This also has the effect that modules are not pin-accurate - they do not have all in- and outputs defined explicitly as an RTL model would have. For example, a burst write may be made through a function burst_write(int addr, int* data, int length). When this function is called, control is transferred to it along with the address, the burst length and a pointer to the data being written.

This leads to another technique, which is to pass data by reference as often as possible in order to avoid spending time copying the data. One must of course take care in case the producer needs to access the data after having transmitted it, as any changes made to the data by the consumer also affect the data as seen by the producer. This is not an issue in MANGO, as the components do not change the flits after passing them on to the next component.

Another technique used in the small example of a burst write above is to use only high-level data types. Bit-vectors and logic-vectors[1] should be avoided. It is also possible to transfer a single object or struct with the entire request rather than the individual fields. The function would then be burst_write(req_t& req).

One more technique which may be taken advantage of in modeling MANGO is to have some modules that contain no processes. Rather, all the processing these modules do is made in the function called through the sc_interface. This may be used in the delay-less modules in the model. Rather than receiving the data, then inform the simulation engine that the data should be processed immediately and then have the simulation engine trigger the function that does the processing, this technique lets the function that receives the data do the necessary processing right away and possibly pass the data on to the next module. This will be elaborated when the implementation of the model is described below.

The last technique which will be used in the implementation of the model is to use methods rather than threads as often as possible. Due to their state-less nature, methods execute more quickly than threads which require a context switch each time they are triggered [9].

[1] A logic type variable may for example have values 0, 1, X and Z, whereas a bit type variable may only have values 0 and 1.

Some further techniques include using dynamic updates of the sensitivity list to avoid unnecessary triggering of methods and threads and make methods and threads that are triggered frequently do as little work as possible. However, these are not really relevant to a model of MANGO as the methods used are only sensitive to a single event and all methods are triggered equally often.

7.2 Implementation Details

This section will present the implementation of the model as it currently is. First, the representation and transport of flits in the system is discussed, then the implementations of the node, arbiter, VC buffers and link are presented and it is described how the actual implementation of the NAs is fitted into the model.

7.2.1 Data Representation and Transport

Transporting data is at the heart of every NoC. The key factor for an accurate model is that data arrives at the same time in both the model and in the actual NoC. How it gets there in the model is unimportant.

In MANGO, the OCP [14] interface is used to provide end-to-end communication between IP-cores. The OCP transactions are divided into multiple flits for transmission through the network with each flit containing a certain part of the transaction. This partitioning into flits must be reflected in the model, as the transaction can only initiate at the slave core when a certain number of flits have arrived. The specific number depends on the transaction type and on whether the transaction is made on a GS or a BE connection.

One possibility for transporting data through the model is to let each flit reflect the contents it would have in MANGO. This is a very straight forward approach that allows easy replacement of model components with the actual implementations that have a certain bit-width. The contents of the flit simply needs to be converted to a logic- or bit-vector of that width before the replaced component and back again afterwards.

It is possible in the model to disassociate a flit and it contents. A scheme for transporting data that does this is to keep the entire transaction in the first flit and let the remaining flits be "dummy" or empty flits. The receiving NA would then have to keep count of the number of flits received and only initiate the transactions when the correct number has arrived. Using this approach, it is less straight forward to replace model components with the actual implementation. The contents of the first flit would need to be passed around the replaced component, as all the data simply would not be able to be passed through the component.

A variation of this last scheme is to immediately move the transaction to the destination NA around the network and only transmit dummy flits through the network. This makes the NAs more complex, as they theoretically need to be able to store an arbitrary number of transactions while waiting for the flits to arrive. Realistically the number would be limited, but some complexity would still be added to the NA.

Moving data like this also allows easy replacement of model components with actual implementations, as the dummy flits contain nothing and thus can be converted to any bit-width. However, if the purpose of replacing a component is to evaluate power consumption in that component, any correlation between the contents of succeeding flits that might impact switching activity would be lost.

As the NA has not been modeled as part of this work, the decision necessarily is to have each flit contain the same parts of the transaction as it does in MANGO. This would also be the decision made if the NA had been modeled, as it is the option with the lowest overhead and the easiest replacement of components. The data structures used for the flits would be an abstract flit class from which each unique type of flit would inherit. These types should match the individual flits in a transaction described in [1]. The implementation presented below is ready for this approach as the data type used in the sc_interface functions is pointers to an abstract flit type.

7.2.2 Components

This section will describe the implementations of the individual components. The source code for both these components and the test benches described in chapter 8 can be found in appendix A.

Link

There are two sc_interfaces to a link: One at either end. For a module sending flits on a link, the sc_interface consists of a single send function, which takes a flit as an argument. For a module receiving flits, the sc_interface consists of functions for unlocking GS and for crediting BE connections. A link in the model implements both these sc_interfaces. A link makes use of two sc_interfaces as well: One to the node at either end. These have the same functions as those of the link itself. This description matches a one-way link, and two links are used to create a bi-directional link between two nodes.

Transporting flits across the link is done in a C++ standard queue used as a FIFO buffer. As mentioned in section 5.2.2, the depth of the pipeline - here represented by the maximum number of flits on the link at a time - does not need to be known if the timing is accurate. A C++ standard queue is thus ideally suited to the purpose of the link. When the send function is called, the flit is pushed onto the queue, along with a time stamp indicating when the flit should be removed from the queue again. An event associated with the send functionality is notified with the delay on the link, triggering when the flit has arrived at the receiver. As it is only possible to have one pending notification of an event in SystemC, the event is notified on transmission of a new flit only if there are no other flits on the link. Otherwise, when a flit leaves the link, the event is notified with the remaining delay of the next flit to arrive at the receiver.

Transmission of unlock and credit signals is handled in much the same way as transmitting flits. The link model imposes no restriction on the frequency with which these signals can be transmitted. In MANGO a separate wire is used for unlock signals for each channel, which has the effect that no restrictions on the timing of unlocks between separate channels exist. The only restriction is on the frequency with which a specific channel may be unlocked, but as the lockbox/unlockbox mechanism prevents a new flit from being transmitted before the channel is unlocked, this restriction is automatically enforced.

In order to properly identify to the nodes from where an unlock signal arrives, the link has a direction value. The value is the direction relative to the receiving node. For example, a link transmitting flits from north to south and unlocks the other way would have a direction value of north.

The link has two methods that are sensitive to the events triggered when flits or unlock signals arrive at their destination. The functionality of initiating a transfer is implemented in the functions defined in the sc_interface and thus need no methods as described in section 7.1.1.

Node

The node is a combination of structural and behavioural modeling. The delay-less routers are not implemented as separate components, as they can be implemented simply as indexing into an array of VC buffers.

The implementation of the node does not contain a BE router or BE VC buffers, as mentioned previously. However, as all programming flits are sent on the BE channel, it needs to be present somehow. Currently, that is done by simply having eight GS VC channels and using statically programmed routing tables.

The model of the node is also different from MANGO in that the routing information is appended to the flit at different points in the two. In MANGO, the routing bits are appended to the flit as it leaves the node, such that the routing tables in one node actually contain the values used in the neighbouring nodes. This is advantageous, as the flits may be routed directly to their destination VC buffer when they enter a node rather than have to wait for a table look-up. However, in the model, the node looks up the destination based on the VC number the flit was transmitted from. As statically programmed routing tables are used, this is not an issue, but if dynamic programming is implemented, this must also be changed as the information is currently stored in two different places in MANGO and in the model. The programming flits to the nodes would thus be sent to the wrong destination in the model.

The node implements a significant number of sc_interfaces: Two for the links to use, one for the arbiter, one for the VCs internally and two for the NA. It makes use of the two link sc_interfaces described above and one sc_interface to the NA. The sc_interfaces implemented by the node will be presented here.

The sc_interfaces used by the links has the same functions as the link sc_interfaces used by the node: Sending flits and unlock signals. The link identifies itself by a direction as mentioned above.

The sc_interface used by the arbiter has two functions: One for moving a flit to the link and one to indicate that the arbiter is ready to receive another flit on a given

VC. As this indication is only needed for BE VCs, it has no effect in this implementation. Similarly to the link, the arbiter must also identify itself by a direction.

The VCs have an sc_interface to the node which is used to transmit unlock signals. The VCs identify themselves by a direction and a number, which corresponds to their priority in the ALG. These are used to look up the destination of the unlock signal, just like it is done in MANGO.

The sc_interfaces used by the NA have one function each: Sending a flit and sending an unlock signal. However, the sc_interface which provides the unlock function ought not be there, as the unlockbox is actually positioned in the node - not in the NA. Sending flits from the NA functions similarly to sending flits from a link: The destination is looked up in a table and the flit is delivered to the appropriate VC buffer. This delivery ought to be delayed by the forward latency through the router, unlockbox, VC buffer and two lockboxes - one when entering the node and one when entering the arbiter - but this delay is not currently implemented.

The sc_interface used by the node to access the NA defines two functions: One for sending flits and one for unlocking VCs. Similarly to the unlock function provided by the node to the NA, this functionality is in the node in MANGO and should be moved there also in the model. The function could then be removed. Sending flits functions the same as in all other sc_interfaces.

Arbiter - ALG

The arbiter implements one sc_interface for the forward flow of flits. As no high-level reverse flow exists through the arbiter, this is the only sc_interface needed to the arbiter. The arbiter makes use of an sc_interface to the node which has functions for sending flits and for indicating to the BE VC buffers that the arbiter is ready to accept a new flit.

The arbiter is implemented by two functions: One for graduating flits from the admission control to the SPQ and one for transmitting the appropriate flit from the SPQ on the link. When a flit is admitted to the SPQ, a boolean array is updated to indicate which lower priority VCs have flits in the SPQ. This array is used to guarantee that a flit on a given VC does not stall more than one flit on each lower priority VC. When a flit from a given VC leaves the arbiter, this boolean array is updated to indicate that higher priority VCs must no longer wait on the given VC.

The function that transmits flits does not accurately model the binary tree control structure shown in figure 3.4. Rather, when it is time to transmit a flit, the highest priority VC with a valid flit is selected. The binary tree control structure should be included in the model to assure that flits are transmitted in the correct order.

When a flit is transmitted, a boolean variable is set to indicate that the arbiter is busy, and an sc_event is set to trigger after the minimum time between flits can be admitted to the link: The maximum injection rate of flits. A method sensitive to this sc_event is then triggered when it is time to transmit a new flit. This function has three steps: First, if there is a flit in the SPQ, the highest priority flit is transmitted. Second, flits are graduated from the admission control to the SPQ. Lastly, if no flit was transmitted in the first step, the current highest priority flit in the SPQ is transmitted.

When a new flit arrives at the arbiter, it is placed in the admission control. If the arbiter is not busy, the method described above is executed, otherwise the flit is left in the admission control and will be handled by the method at some future point in time. As mentioned, this does not accurately model the actual implementation of the control structure in figure 3.4 and should be changed.

Virtual Channel Buffers

The VC buffers present two sc_interfaces to the node: One for sending flits to the buffer and one for sending unlock signals and indications of the arbiter being ready to accept a new flit. Even though the unlock mechanism is different for GS and BE VCs, the same function signature may be used for the unlock function of both types. Even though the GS VC buffers do not need the indication from the arbiter, they may still implement a function that simply has no effect. This allows the VCs to implement the same sc_interfaces and thus be interchangeable transparently to the surrounding components. Of course, when the BE router has been implemented, care must be taken to route flits on BE and GS VCs through the correct router. However, as previously mentioned the BE parts of MANGO are not modeled in this work.

Two sc_interfaces are used by VC buffers: One to send flits to the arbiter and another to send unlocks to the node which passes them on to the appropriate link.

The GS VC buffer model is delay-less as discussed in section 6.1.2. Therefore, it contains no methods or threads and all work is done in the functions defined in the sc_interfaces. The buffer can hold three flits at a time: One in the unlockbox, one in the lockbox and one in a latch inbetween. When a flit is sent to the buffer, it is immediately moved as far as possible towards the arbiter. If the flit enters or passes the lockbox, a boolean variable is set to indicate the locked state. If the lockbox is locked when the flit arrives, it is moved to the latch if it is empty or the unlockbox if the latch is not empty. In case the unlockbox is passed, the unlock function in the node sc_interface is called. When a VC buffer is unlocked, flits in the latch or unlockbox are moved forward if any are present. The flit in the latch is sent to the arbiter while the flit in the unlockbox is moved to the latch. The unlock function is then called. If no flits are present in the VC buffer when the lockbox is unlocked, the boolean variable that indicates the state of the lockbox is set to indicate that it is not locked.

Overall

The effect of the node and VC buffers being without methods is that all processing of flits in the nodes happens when the methods in the link and arbiter are triggered. When a method on the link that indicates the flit has arrived at its destination is triggered, the send function in the node sc_interface is called which calls the send function in the VC buffer sc_interface. If the lockbox is not locked, the send function in the arbiter sc_interface is called, which calls the send function in the link sc_interface. When control is returned to the VC buffer, it calls the unlock function in the node sc_interface as the flit has left the unlockbox. The node then calls the unlock function in the appropriate link, completing all the processing required in the node for that flit. The call stack would be as shown in figure 7.2(a). If the lockbox is locked at the time of the function call, the unlock function is called if the flit passes the unlockbox and the call stack would be as shown in figure 7.2(b). If the flit is not able to pass the unlockbox, the node::unlock_vc function would not be called, and that part of the figure should be omitted.

The situation when an unlock arrives is similar and will not be described in detail. The point is that the simulation engine is involved in very little of what happens inside the node. The only time a function in the node is invoked by the simulation engine is when a flit has been stalled in the arbiter. When a new flit can be sent on the link, the simulation engine calls the method in the arbiter which in turn calls the link::send function before returning control to the simulation engine.

7.3 Network Adapter

The NAs in MANGO have not been modeled as part of this work. Therefore, the actual implementations need to be connected to the model in order to simulate a system. This also presents an opportunity to discuss the issues involved when using actual implementations alongside the model. First, a brief introduction to the NAs will be given, then two approaches to interfacing between the model and the NAs will be discussed and finally the details of this interfacing will be described.

7.3.1 Introduction

Two types of NAs exist: An initiator adapter which connects an OCP master IP-core to the network and a target adapter which connects an OCP slave IP-core. Both NAs have the same interface to the node, which consists of four 39 bit wide input ports and four 39 bit wide output ports along with handshake signals on all eight ports. These ports are treated similarly to virtual channels, three of them being able to access GS connections and the last being assigned to BE traffic.

Each NA contains tables with routing information which need to be programmed before they can be used. The tables in the initiator are programmed by the attached master IP-core, while the tables in the target are programmed by a master IP-core which transmits the programming information on BE transactions.

7.3.2 Interfacing Approaches

The NAs have two interfaces each: The one described above to the node and an OCP-interface to the attached IP-core. It is only the interface to the node that needs to be converted.

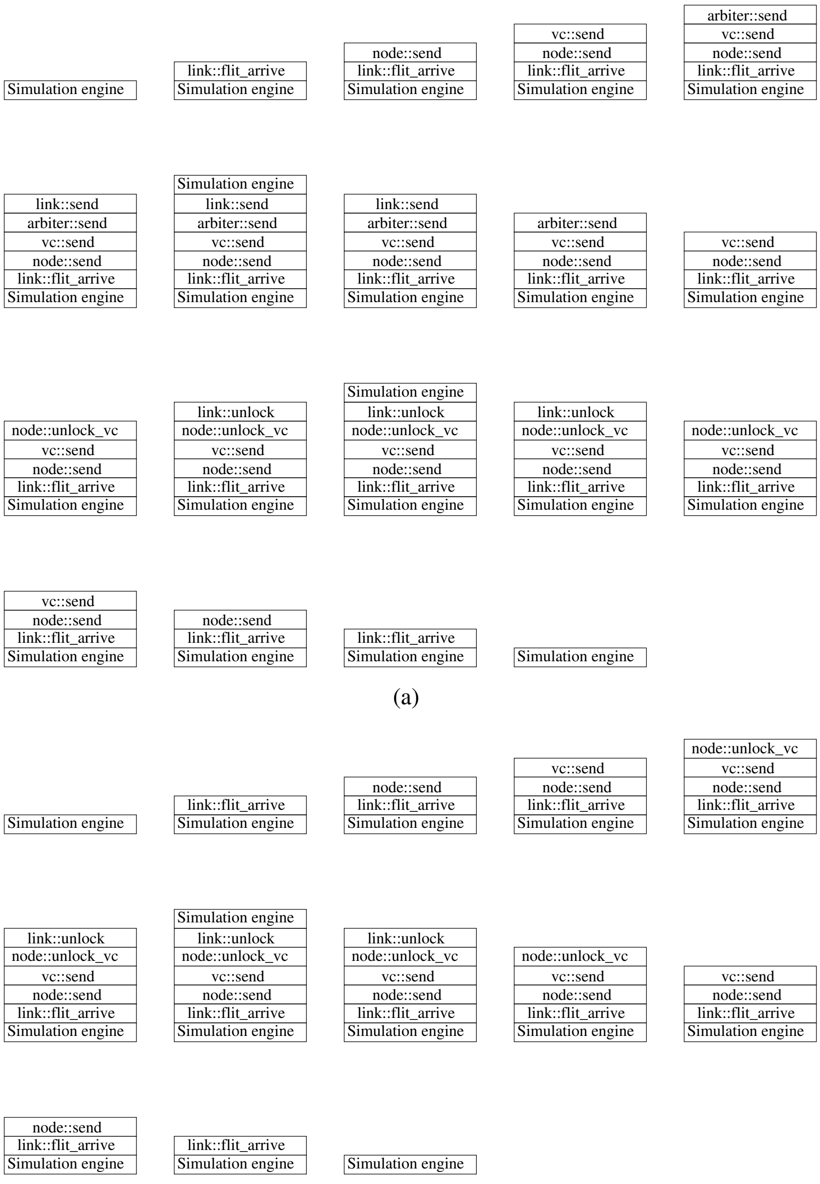


Figure 7.2: The call stack when flits arrive at a GS VC. The calls to the simulation engine from link::unlock and link::send are used to delay the unlock signal and the flit respectively. 7.2(a): The flit passes straight through the VC. 7.2(b): The flit is not able to pass through the lockbox in the VC, but it passes the unlockbox, resulting in the unlock functions being called.

This can be done in two ways. One is to create a SystemC module that encapsulates the NA and interfaces to both the OCP IP-core and the node. The other is to create a SystemC module that sits inbetween the node and the NA. Generally, when a model of a component is replaced with an actual implementation, conversion between a bit-level interface and the model needs to be done at both inputs and outputs of the component. In such a case, an encapsulation of the component in a SystemC module that handles both conversions will be the most practical approach. Suppose for the sake of the example that the link including encoder and decoder is to be replaced by the actual implementation. The module would the implement both sc_interfaces for the link and would have sc_ports with each of the sc_interfaces to the node used by the link. When the send function in the link sc_interface is called, the flit is converted to a bit or logic vector which is applied to the encoder input. A short time later, the request input to the encoder can be asserted and deasserted once the acknowledge has been asserted. The minimum delay between sending new flits implemented in the arbiter will make sure that flits do not arrive faster than the encoder can accept them. The data proceeds down the link pipeline, and when the decoder asserts its output request the logic vector can be converted back into a flit and the send function in the node can be called. The handshake on the output of the decoder of course needs to be completed. A similar string of events will take place for unlocks transmitted across the link.

In the case of the NA however, it is only the interface to the node that needs to be converted. Thus, there is no need to encapsulate the NA, as the OCP interface signals are not processed in any way. Thus, it would be advantageous to simply insert a module between the NA and the node which handles the conversion from logic vectors and handshake signals to flits and function calls and back again. This module needs to make realistically timed handshakes with the NA on one side and to observe the assumptions and behaviour of the model on the other side.

7.3.3 Implementation Details

One issue that must be dealt with in the implementation of this conversion module is the positioning of the lock- and unlockboxes as mentioned in section 7.2.2. In the current implementation of MANGO they are placed immediately inside the node, while the model has them in the NA. Thus, when using the model of the node and the actual implementation of the NA, the lock- and unlockboxes are nowhere to be found. The conversion module therefore needs to implement their functionality.

This section will first describe how transmissions from the NA and then transmissions to the NA are handled. Achieving the behaviour of lockboxes is covered in the part about transmissions from the NA, while unlockboxes are covered in the part about transmissions to the NA.

Transmissions from the NA

When a transaction is made from the NA on one of the VCs, one of the corresponding four - one for each VC - TxReq signal is asserted. This change in signal value triggers a function which inserts the 39 bits on the data port in a templated generic data flit. This flit may hold any value of the template type, which in this case is a logic vector representation of a flit. The acknowledge signal - TxAck - is only asserted when the unlock function is called by the node. This emulates the behaviour of the lockbox, as the NA will not be able to transmit another flit before it may be accepted by the destination VC buffer in the node. Once the NA deasserts the TxReq signal, the TxAck signal is also deasserted after a fixed delay of 0.4ns which has been measured as the actual delay in simulations of MANGO.

One issue in this implementation of the conversion module is that the transmission of flits from the NA to the VC buffers in the node is delay-less. Thus, flits arrive too early at the arbiter. A delay should be inserted in order to ensure that the flits arrive at the correct time. Similarly, the unlocks are transmitted from the VC buffers to the conversion module with no delay, but a delay of 5.1ns has been inserted between the unlock arrives and the TxAck signal is asserted. This is the delay that has been measured between rising edges on TxReq and TxAck signals in simulations of MANGO when the destination VC buffer is empty. This delay includes both the forward latency from the interface between NA and node to the VC buffer and the latency of the unlock signal back to the lockbox. It thus has the effect that the interval between flits being admitted into the node is the same in the model and in MANGO as long as the VC buffer does not become filled. The value of this delay should be corrected at the same time the forward latency of flits is inserted into the model.

Transmissions to the NA

When a flit arrives in the conversion module, the logic vector is retrieved and written to the RxData port. After a delay of 1.5ns, the RxReq signals is asserted. This delay has been measured in simulations of MANGO. When the NA asserts RxAck, the conversion module deasserts RxReq after 0.7ns for the three GS channels and 3.6ns for the BE channel. These values have also been measured in simulations. When the NA deasserts the RxAck signal, the unlock function in the node is called.

This part of the conversion module also has some timing issues that need correcting. These are however not directly part of the conversion module, but are placed in the link. The issue is that the delay when transmitting a flit or unlock across a link is constant. In reality, the delay is different if the flit is destined for a VC or for the NA. These delays should be corrected as well, but a discussion of how to implement this is defered to chapter 9.

Chapter 8

Verification and Results

This chapter will present the verification of functionality and timing of the model along with the difference in performance between the model and MANGO. First the test system will be introduced.

8.1 Test System

This section will first introduce the topology of the test system and then the method of testing applied.

8.1.1 Topology

The test system used is the same as the one presented in [1], which will be described here as well. The system consists of three nodes, one initiator NA, one target NA and a pair of OCP cores. The structure of the system is shown in figure 8.1. Two GS connections are used in the system: One from the initiator to the target and one from the target to the initiator, which is used for responses. These connections are set up by the master OCP core when MANGO is used and statically programmed into the model.

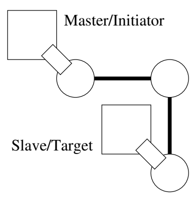


Figure 8.1: The system used in testing the model and for determining the difference in simulation performance between MANGO and the model.

In both MANGO and the model, the target NA needs to be programmed. This happens through BE transactions from the OCP master. However, the BE part of MANGO is not modeled, but this issue may be overcome by making a statically programmed GS connection in place of the BE VCs. Thus, any BE flits transmitted by the initiator NA will be routed to the target NA, allowing it to be properly programmed and used. One issue though is that the BE router in MANGO rotates the first flit of a BE transaction - the flit that contains the route - a certain number of bits in each node. This rotation is implemented by the conversion module before the flit is transmitted through the network in the model. This way, the flit that is presented to the target adapter has the correct contents.

In MANGO, the ports on the nodes that are not connected to something in the system have traffic generators attached. These generators can be used to include background traffic in simulations, but as time constraints did not allow a conversion between the 2-phase dual-rail generators and the model, the results presented here have only the system traffic included.

The same OCP cores and NAs are used when simulating both the model and MANGO. The only variation is thus the actual network, which is also the object of interest. The testing environment should thus be as neutral as possible towards the test.

8.1.2 Method of Testing

The testing of the model can be divided into three major parts: Functionality, timing and simulation performance. The method for testing each will be presented individually along with the testing results in the following sections in this chapter. This section will present the environment used in the tests.

The two OCP cores used in the test system are implemented in a single SystemC module. The master OCP core reads the test vectors from input files in the module's constructor. This means that the files are read during design elaboration and that the actual simulation will not be affected by this disc activity.

The files contain one OCP transaction on each line. If the OCP master is to be idle for a clock cycle, an idle OCP command - which has no effect - is inserted on a line. The simulation has three phases: A programming phase where the NAs - and nodes when simulating MANGO - are programmed, a wait phase where the OCP cores are idle and wait for the programming to complete and finally the interesting phase where the network is actually used which will be called the data phase. Two files are used to ease simulating either the model or MANGO: One which contains the transactions used for programming the NAs and nodes, and one which contains data test vectors. The programming file has a number of idle commands at the end to wait for the programming of the network to complete.

During the data phase, the time when a transaction is initiated by the master and received by the slave are recorded. Similarly, the time when a slave responds to a read request is recorded along with the time when the master receives the response. These are the end-to-end latencies, which should be identical between MANGO and a fully timing accurate model and are the interesting latencies for a system which utilises MANGO.

The test vectors used in the data phase are randomly generated previous to the simulation. A bug in the NA requires the upper eight bits of the address field to be different from zero - even on GS transactions - as the transaction is otherwise taken to be a programming transaction to the NA. The lower fifteen bits of both address and data are randomly generated, as the largest random number that can be generated with the C++-compiler that was used is $2^{15} - 1$. The upper seventeen bits are fixed at '0' for both data and address except for a few '1's thrown in the upper eight bits of the address to avoid the bug in the NA. The program that generates the test vectors inserts an idle operation with 80% chance at each iteration. It generates 10000 writes and 5000 reads on the GS connection between master and slave. These are ordered such that all the writes are executed first.

8.2 Functionality

The functionality test can be divided into two parts. One is correct end-to-end transport of transactions and the other is correct ordering of flits through a single arbiter.

For the end-to-end transport of transactions, simulations of the test system show that all transactions are completed. This means that the model is able to correctly transport flits generated by the NAs to their destination using the statically programmed routes.

It has not been tested if the arbiter outputs flits in correct order - the same order the actual implementation of the arbiter outputs. However, it is almost certain that the ordering will be incorrect, as the binary tree control structure of the actual arbiter has not been modeled. A test should be developed specifically for this purpose, and would also necessarily include some measure of timing, as flits would otherwise simply be passed through the model arbiter instantly. This timing would only have to include the required delay between outputing flits, as this would cause interactions between flits stalled in the arbiter. The actual delay through the arbiter is irrelevant, as it has been assumed constant in section 6.2 regardless of how long the flit has been stalled in the arbiter. If tests show that this assumption is faulty and affects the simulation results, this difference in forward latency needs to be included in the model. However, it will never be possible to accurately model the arbitration unit in figure 3.4, which behaves randomly if two flits arrive at the same time.

8.3 Timing

In order to evaluate the timing accuracy of the model, the end-to-end latencies of transactions have been measured in simulations of both MANGO and the model. These should ideally be identical, but the values used for the forward latency of flits across links and the latency of unlock signals in the model have not been measured in simulations of MANGO but rather estimated at 10.5ns and 3ns respectively. The minimum time between the arbiter admitting flits onto the link has been measured in simulations of MANGO as 2.6ns.

The table in figure 8.2 summarise the mean and variance of the end-to-end latencies in both MANGO and the model. The write requests consist of two flits, while the read requests and responses consist of a single flit.

	Mean		Variance	
	MANGO	Model	MANGO	Model
Read request	28580	33601	$6.62 \cdot 10^5$	$7.44 \cdot 10^5$
Read response	32019	36446	$6.62 \cdot 10^5$	$1.52 \cdot 10^6$
Write request	37426	91340	$3.23 \cdot 10^6$	$3.27 \cdot 10^8$

Figure 8.2: The mean and variance of transaction latencies in the model and in MANGO. All times are in ps.

The mean and variance for single flit transactions are fairly close between the model and MANGO. Both values are somewhat higher for the model, but some of the latencies used in the model are estimated and not measured in MANGO. If accurate measurements of these latencies are made, the model should achieve approximately the same latencies as MANGO produces. However, there is a large difference in the end-to-end latency of write requests between MANGO and the model. This difference is much larger than the one observed for the single flit transactions. Similarly the variance on the latencies of the write requests in the model is much larger than the one for MANGO.

A possible explanation for these large differences may be found by taking a closer look at the latencies of the read requests. In the simulations, the read requests are executed after all the write requests have been transmitted. When a read request is in progress, the master must wait for the response before making a new request. Thus, only the flit associated with the read request is present in the network. By the time a new request is made, all VCs should be unlocked and succeeding flits should not have any influence on each other at all. It is observed that the latency of all flits beside the first one distributes roughly evenly over three discrete values: 27.6, 28.6 and 29.6ns for MANGO and 32.6, 33.6 and 34.6ns for the model. The very first read request however has a latency of 26.6ns in MANGO and 43.6ns in the model. This request follows a write request, and may thus experience influences from a flit immediately in front. In the model, this influence is seen to be very large - at least an extra delay of 9ns. While the difference for write transactions is much larger than this between MANGO and the model, it must be remembered that too large values for the unlock delays can have significant consequences when transactions are made at maximum speed. In this case, the VC buffers in the model may fill up faster than they do in MANGO. The flits involved in such a burst of traffic would thus experience a much larger transmission latency in the model than they would in MANGO. Whether this is actually the cause of this large difference in latency needs a more detailed analysis of simulation traces to decide.

Another contributing factor could be in the conversion module to the NA, where the TxAck signal is only asserted 5.1ns after the unlock function has been called from the node. If this delay is too large, the NA may itself become unable to accept new transactions, stalling the transaction already at the OCP interface. Similarly, VC buffers filled to capacity would eventually cause the TxAck signal to be asserted after a very long time, as the new flit would stall in the unlockbox. Only when a preceding flit has left the unlockbox in the succeeding buffer and the unlock signal has propagated backwards, the new flit will be able to unlock the NA. All in all, it would be better to resolve the currently known issues in the implementation mentioned throughout this and the previous chapter before digging deeper into the cause of the observed difference in latency for write requests. Resolving these issues includes measuring the delays in MANGO that are currently only estimated in the model.

8.4 Simulation Performance

Measuring the execution speed of a simulation - or any other computer program - is not as straight forward as taking a stopwatch and measuring the time from the execution is started until it terminates. Furthermore, the actual implementations of the NAs are used when simulating the model, which will have a large impact on the execution speed.

ModelSim includes a performance profiler, which pauses the simulation every x milliseconds and samples which part of the design is being executed. When the simulation run is completed, the distribution of samples between components can be reported. It is not the absolute number of samples taken in a specific component that is interesting, but rather the relative distribution between components. Furthermore, the distribution of samples does not necessarily indicate a difference in simulation performance between components. For example, if a system has two components and roughly 50% of the samples are taken in each component, it can be deduced that roughly half of the time during the simulation is spent in each component. However, it might be that one component is activated only once while the other is activated thousands of time, indicating a huge difference in simulation performance between the two components, but this is not reported by the performance profiler. In the test system used here though, all components are activated equally often.

Apparently, the performance profiler does not report in which part of SystemC code samples are taken, but all samples in SystemC code are commonly reported under an entry simply called NoContext. It is thus not possible to observe whether the samples are taken in the model or in the SystemC OCP cores, but as will be seen from the results, this is not really relevant.

The table in figure 8.3 shows the number of samples taken in the NAs, the network and SystemC code and the percentage of the total number of samples taken in user code during the simulation. When simulating MANGO, 16325 samples were taken, 7266 or 45% of these in user code. For the model, the numbers were 3631 samples in total and 545 or 15% in user code. The simulation environment was identical between the two simulations.

	Number of samples		% of samples	
	MANGO	Model	MANGO	Model
Initiator NA	312	192	4.3	35.2
Target NA	510	347	7.0	63.7
Network	6440	–	88.6	–
SystemC	4	6	0.1	1.1
Total	7266	545	100	100

Figure 8.3: The distribution of samples between simulations of MANGO and the model. No samples are reported in the network in the model, as it is not reported which part of SystemC code samples are taken in.

As can be seen, the number of samples is significantly decreased in the model compared to MANGO. The drop in the relative number of samples in user code is not readily explained. It may be caused by the faster executing model which prompts the threads in the OCP cores to be triggered more often, measured in wall clock time. As triggering a thread involves a context change, it is quite expensive and may be the cause of many of the samples taken outside user code. Also, the number of simulation events in the NAs is approximately constant between simulations - the same flits and OCP transactions pass through - causing the amount of time spent by the simulation engine to handle these events to increase relative to time spent in user code. However, this is difficult to state for a fact without more detailed knowledge of the implementation of the performance profiler than is given in the ModelSim user manual.

Another notable difference between MANGO and the model is that fewer samples are taken in the NAs in the model than in MANGO, despite the fact that the same number of flits pass through in both. One possible explanation is that in the model, the entire data input to the NA from the network is set up at the same time, whereas in MANGO individual bits may arrive at slightly different times due to different delays through the standard cells. The model thus produces a single simulation event when setting up data, whereas MANGO may produce up to 32 events. Another possible explanation may be found in the smaller code base of the model possibly producing fewer cache misses during simulation. However, determining this as a plausible cause also requires more detailed knowledge of how the performance profiler works.

Despite the apparent shortcomings of the performance profiler, it can be seen that the majority of samples taken in the model are taken in the NAs. Compared to MANGO, the NAs percentage of samples has increased dramatically from 11.3% to 98.9% combined. At the same time, the percentage of samples taken in the network has decreased from 88.6% to at most 1.1%. This is a very dramatic increase in performance in this part of the system, and introducing a high-level model of the NAs should yield a significant increase in overall simulation performance. While the number of samples taken in SystemC code is very small, the expected speedup of a purely high-level SystemC model appears to be on a magnitude of a factor 1000 compared to simulating the netlists of standard cells. This factor is calculated as the ratio of the number of samples in SystemC code in the model to the number of samples in the network in MANGO, $\frac{6}{6440}$. However, in the test vectors, only the lower 15 bits are different from '0', which halves the potential switching activity in MANGO, thereby reducing the possible number of simulation events considerably. The speedup may thus be greater, but the number of samples in SystemC code is very small for concluding a specific speedup, but a factor with a magnitude around 1000 is not unrealistic based on these measurements.

Chapter 9

Discussion

This chapter will discuss how to resolve the known issues in the current implementation of the model, how the model may be applied to system level modeling and simulation and finally how the model may be expanded and improved upon in the future.

9.1 Resolving Known Issues

The known issues in the current implementation of the model include that flits should be delayed by a different amount of time if they are destined for the NA or a VC buffer. How to resolve this issue will be discussed here.

Two possible solutions have been presented previously: Allow variable delays on the link or apply the shortest delay on the link and have a separate delay component in the node for the flits that need additional delays.

The cost of adding a second delay component in terms of simulation time depends on which destination requires the additional delay. If it is flits heading to the NA that require an additional delay, only one delay - and thus one simulation event - will be added to each flit. If however it is the flits heading to VC buffers that require the additional delay, one simulation event is added to each node each flit passes through, increasing the number of simulation events by 50%[1]. The percentage increase when it is flits to the NA that require an extra delay depends on the length of the routes in the system. If the system is small - for example a 3-by-3 grid network - the longest path passes through four nodes, generating ten events including the delays for the forward latency and the unlock signal on the boundary between the initiator NA and the node. An additional event is thus a 10% increase, while for the shortest path - a single hop in the network - four events are generated, resulting in a 25% increase. For a large system with a 10-by-10 grid, the maximum increase is still 25%, but the minimum is 2.5%. Assuming a simulation time that is proportional to the number

[1] Currently, two simulation events are generated per hop: One for the forward latency of the flit and one for the unlock signal.

of events generated, this can be a significant increase in simulation time for large simulations.

Allowing variable delays on the link incurs no cost in added simulation events. However, depending on the relation between the minimum time between allowing flits onto the link, the forward latency to a VC buffer and the forward latency to the NA, some extra processing overhead may be required when inserting flits in the queue on the link. Let t_{flit} denote the minimum time between flits, t_{VC} the delay applied to the flit when it is headed to a VC buffer and t_{NA} the delay when the NA is the destination. If $|t_{VC} - t_{NA}| > t_{flit}$, the flit with the shorter delay may arrive first at its destination even when transmitted after the flit with the longer delay. This does not mean that one flit overtakes another in the link, but that the delay through the router and other parts of the node is much smaller for one flit than for the other. It is thus necessary to order the flits by arrival time in the model of the link when a new flit is transmitted. This incurs some additional processing overhead for every hop a flit makes, but as the size of the queue is generally not very large - the number of flits is less than the depth of the asynchronous pipeline - this processing overhead should be minimal. If $|t_{VC} - t_{NA}| \leq t_{flit}$ however, no flit arrives before its predecessor and no ordering of the flits on the link is necessary.

Which solution is preferable depends on a number of variables, but overall, allowing variable delays on the link seems to have the smallest cost in terms of simulation time.

9.2 Application of Model

This section will take a look at how this model may be applied to various tasks in both the development of MANGO and in modeling systems that use a MANGO based network-on-chip for communication. It will also be discussed how the NAs can be modeled such that bus-interfaces may be abstracted away.

9.2.1 Exploring Concepts of Network-on-Chip in MANGO

A high-level model as the one developed in this work may be used to explore various concepts of network-on-chip in MANGO without producing netlists of standard cells and spending time simulating these. It may also be used to investigate alternative implementations of MANGO.

Currently, programming flits to the router tables in the nodes and NAs are transmitted on the BE part of MANGO and no acknowledgement that the programming has completed is sent. This has the effect that the time it takes for a GS connection to be set up is unknown and the system has to wait “long enough” before utilising the GS connections. Work is presently under way to improve upon this situation, but a high-level model could have been used to investigate the impact of different improvements. For example, the impact of adding a BE VC reserved for programming flits could be examined compared to the current approach of sharing the BE VC for both programming and data flits.

On another level, the model might be used to examine the impact of varying VC buffer depth on average case latencies in a system. While hard guarantees are made for the worst case latencies, and the best case latencies can be calculated, the average case latencies depend on other traffic in the network and can thus only be determined through simulation. Larger VC buffers may improve the average case latencies as more flits are allowed to propagate further along their routes, eventually resulting in earlier arrival at their destination. However, if all links along the route are fully utilised, most flits will still have their worst case latency, but for moderate traffic loads, deeper buffers may have a significant impact on average case latencies. The model can help in determining how large this impact is.

Another application of the model in the development of MANGO could be to examine the impact of different routing schemes. For BE traffic, adaptive routing might be employed in order to reduce congestion in areas with high traffic load. Several different adaptive routing schemes may be tried out in a relatively short amount of time with much less effort than would be required to construct netlists of standard cells.

9.2.2 System Modeling

It has been stated previously that a network-on-chip is an approach to system level communication in system-on-chip. Models at varying level of detail are employed at different phases of the system development process. Early on in this process, there is no need for fully detailed models of any of the components. Purely behavioural models are used for processing elements, and it may not even be decided at this point if the tasks executed are to be implemented in software or hardware or a combination of the two. This corresponds to the application level of abstraction of section 2.3, where even the high-level model of MANGO developed as part of this work may be too detailed.

However, at the system designer level, CPUs may be represented by instruction set simulators and dedicated hardware elements by cycle accurate behavioural descriptions. Similarly, a timing accurate model of the communication system should be used in order to generate realistic estimates of the system performance. The model also needs to be high-level in order to simulate many system configurations in a short amount of time such that the best configuration may be selected for implementation. These configurations involve assignment of GS VCs to connections as well as different network topologies and mappings of IP-cores to the network. For a system with 16 IP-cores, 3 different grid-based topologies exists with 16! possible application mappings each. If other topologies such as trees, toruses and heterogeneous topologies are also taken into account, the number of possible configurations increases rapidly, and simulating even a handful becomes a very large task. A fast executing model allows more configurations to be simulated than would be feasible with a detailed model, such as the netlists of standard cells presently available for MANGO.

Including the number of possible assignments of VC buffers to GS connections increases the number of configurations even further, and changing the route a GS connection takes through the network may impact performance greatly, if some highly congested links can be avoided. A high-level model may help in determining the traffic patterns in the network, which allows the system designer to change the routes taken by GS connections, the application mapping or the topology in order to minimise congestion. Iterations on this design step can be made quickly with a high-level model.

9.2.3 Abstracting Bus-Interfaces Away

One issue in system modeling is how to connect the models of individual components. When dealing with high-level models, it does not make much sense to have pin-accurate interfaces between components, as the models do not have this level of detail otherwise. In fact, when it has not yet been decided if individual components are to be implemented in hardware or software, it makes no sense to speak of a specific bus-interface. What should rather be done is to move to transaction level modeling (TLM) [9], where sc_interfaces are used to provide the same functionality a bus-interface would provide.

In a model of MANGO, this could be realised by letting the initiator NA implement an sc_interface with functions such as `void write(int address, int data)` and `int read(int address)`. The model would then need a flit capable of transporting an integer through the network. However, requiring the users of the model to use only integers when communicating between processes would be very restricting. Thus, the model would have to be able to transport every type the user might use, including user-defined classes. For this purpose, a templated generic data flit could be used, such as the one used in the conversion module for the network adapters described in section 7.3.3. This data flit may hold any type that may be defined in C++. However, a user-defined class may very well exceed the 32 bits available in a single flit, thereby upsetting the timing in the system as too few flits are transmitted for such a class. A simple solution would be to require the user to inform the NA how many flits the class should be partitioned into, allowing the correct number of flits to be transmitted. Adding this functionality to a model of the NAs would make the model very versatile when simulating systems.

9.3 Future Work

This section will discuss how the current model can be extended and improved upon in order to be even more useful in system design and exploration of network design choices.

9.3.1 Parametrising the Model

While the current implementation of the model reflects the current implementation of MANGO, it might be extended to allow experimentation with the setup of the network. For example, a system designer might want to experiment with nodes with more than four neighbours, or a different composition of virtual channels than seven GS and one BE channel on each link. It might also be that a system has a highly congested link and the system designer would like to have two physical links between the affected nodes, the impact of which could be examined in a parametrised model. Another part of the model that might benefit from parametrisation is the depth of VC buffers which can be used to improve average case latencies as described above.

These and other design choices can be tried out in a parametrised model before setting out to implement them in the actual network. The timing in such a model needs to be estimated, as measurements can not be made in an actual implementation. Even though the timing will not be completely accurate, the model should still be able to present a realistic indication of how an implementation of MANGO with the given parameters would behave.

9.3.2 Estimating Power Consumption

Currently, the model is very focused on behaving correctly with regard to timing. Another performance metric it would be very useful to have the model report is the estimated power consumption. In systems with low-power requirements, minimising power consumption can be equally or more important than optimising speed.

An estimate of the power consumption can be made by keeping a copy of the last flit to pass through a component and calculating the switching activity caused by the next flit to pass through. The switching activity can then be used to provide an estimate of how much power is consumed.

With the addition of power estimation, the high-level model may be used to investigate the impact of various power reducing techniques such as encoding transactions on the link in order to minimise switching activity. Topologies, application mappings and GS connection routes may also influence power consumption, as the number of hops on GS connections vary with these. These influences may also be examined with a high-level model.

9.3.3 Handshake Level Model

With the large increase in simulation speed observed by going to a high-level model, it could be interesting to see how a handshake level model performs. It is easier to achieve a timing accurate model using handshake level modeling, due to the finer granularity of the model. Where the link component in the high-level model covers all forward latencies, they would be distributed between smaller components in a handshake level model. Power consumption would also be estimated more accurately, due to the finer granularity.

In a model at the handshake level, it would also be easier to replace model components with actual implementations than in a high-level model, as there is no need for a conversion between handshakes and the model. However, investigating high-level concepts such as adaptive routing schemes or system design elements such as the number of links into and out of a node would be more difficult in a handshake level model, as components would have to be added and connected by signals to other components. In the current high-level model, adding a port to a node can be done simply by increasing the size of the arrays that represent the sc_ports and the routing tables.

The two different models each have advantages and disadvantages, and the better choice might actually be not to have one or the other, but both. Early system-level exploration would then use the high-level model to investigate topologies, application mappings and GS connection routings, while the handshake level model could be applied to provide a more fine-grained picture of timing and power consumption. Network developers would use the handshake level model to investigate the impact of new implementations of components.

Chapter 10

Conclusions

In this work, the design, implementation and testing of a high-level model of MANGO has been presented. The purpose of the model was to obtain a high simulation speed while being accurate in terms of timing. A model that meets both requirements has been designed, and implemented, but some issues are present in the implementation that should be corrected.

A small test system has been setup in order to compare simulations of the actual implementation of MANGO and the model. The two behave comparably when isolated flits are transmitted through the network, as happens in single flit transactions. When multiple flits are present on a GS connection however, they interact differently in the model than in MANGO. This may be caused by the fact that the values used for delays in the model are only estimated and not based on measurements in simulations of the current implementation of MANGO.

The test system demonstrates using actual implementations of components in conjunction with the model. The actual implementations of the network adapters are used, and a conversion module has been implemented that converts between handshakes and the model. This module shows the general approach to and feasibility of replacing components from the model with those from the actual implementation.

The performance profiler available in ModelSim has been used to measure the execution time of various parts of the test system in simulations of both MANGO and the model. A drop in the simulation time was observed when replacing MANGO with the model. However, the network adapters used in both simulations are the ones from MANGO, which are implemented as netlists of standard cells, which means the drop in total simulation time is relatively modest. However, in the simulation of the model, the majority of the execution time spent in user code was spent in the NAs. The speedup observed in the network between the simulation of MANGO and the model indicates that the magnitude of the achievable speedup by using a high-level model is roughly a factor 1000.

Bibliography

[1] T. Bjerregaard. Programming and using connections in the mango network-on-chip. To be submitted.

[2] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005.

[3] T. Bjerregaard, S. Mahadevan, and J. Sparsø. A channel library for asynchronous circuit design supporting mixed-mode modelling. In Odysseas Koufopavlou Enrico Macii, Vassilis Paliouras, editor, *PATMOS 2004 (14th Intl. Workshop on Power and Timing Modeling, Optimization and Simulation)*, Lecture Notes in Computer Science, LNCS3254, pages 301–310. Springer, 2004.

[4] T. Bjerregaard and J. Sparsø. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, pages 1226–1231. IEEE Computer Society, mar 2005.

[5] T. Bjerregaard and J. Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)*, pages 34–43. IEEE Computer Society, mar 2005.

[6] T. Bjerregaard and J. Sparsø. Implementation of guaranteed services in the MANGO clockless network-on-chip. *IEE Proceedings: Computing and Digital Techniques*, 2006. Accepted for publication.

[7] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, TBA. Accepted.

[8] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture, A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[9] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[10] http://www.imm.dtu.dk/arts.

[11] J. Madsen, S. Mahadevan, K. Virk, and M. J. Gonzalez. Network-on-chip modeling for system-level multiprocessor simulation. In *The 24th IEEE International Real-Time Systems Symposium*, pages 265–274. IEEE Computer Society, dec 2003.

[12] S. Mahadevan, M. Storgaard, J. Madsen, and K. M. Virk. ARTS: A system-level framework for modeling mpsoc components and analysis of their causality. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, sep 2005.

[13] http://www.model.com.

[14] http://www.ocpip.org.

[15] J. Sparsø. Asynchronous circuit design - a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pages 1–152. Kluwer Academic Publishers, Boston / Dordrecht / London, dec 2001.

Appendix A

Source Code

This appendix lists the source code of the source files produced in this work. A short description to the file will be given where necessary.

A.1 Top Level Files

A.1.1 interfaces.h

```
/*
Component interfaces. All components must inherit from these.
*/

#ifndef _INTERFACES_H
#define _INTERFACES_H

#include <systemc>
#include "types.h"

/* Arbiter */
class arbiter_if : virtual public sc_core::sc_interface {

public:
  virtual void arbitrate(flit*, int)=0;

};

/* Virtual Channel */
class vc_transmitter_if : virtual public sc_core::sc_interface {

public:
  virtual void send(flit*)=0;

};

class vc_receiver_if : virtual public sc_core::sc_interface {

public:
  virtual void unlock()=0;
  virtual void arb_ready() {}

};

/* Link */
class link_transmitter_if : virtual public sc_core::sc_interface {

public:
  virtual void send(flit*)=0;

};

class link_receiver_if : virtual public sc_core::sc_interface {

public:
  virtual void unlock_gs(int)=0;
  virtual void credit_be(int)=0;

};

/* Node */
class node_transmitter_if : virtual public sc_core::sc_interface {

public:
  /* Unlocks from links */
```

```
  virtual void unlock_gs(const int, const direction)=0;
  virtual void credit_be(const int, const direction)=0;

};

class node_receiver_if : virtual public sc_core::sc_interface {

public:
  virtual void send(flit*)=0;

};

class node_arbiter_if : virtual public sc_core::sc_interface {

public:
  virtual void arb_send(flit*, const direction)=0;
  virtual void arb_ready(const int, const direction)=0;

};

class node_internal_if : virtual public sc_core::sc_interface {

public:
  /* Unlocks from VCs */
  virtual void unlock(const int, const direction)=0;
  virtual void credit(const int, const direction)=0;

};

class node_na_receiver_if : virtual public sc_core::sc_interface {

public:
  virtual void na_send(flit*)=0;

};

class node_na_transmitter_if : virtual public sc_core::sc_interface
    {

public:
  virtual void na_unlock_gs(const int)=0;

};

/* Network Adapter */
class na_if : virtual public sc_core::sc_interface {

public:
  virtual void unlock(const int)=0;
  virtual void send(flit*)=0;

};

#endif
```

A.1.2 types.h

```
/*
  Types for use in NoC model.

  Contains:
    Direction
    Flit

*/

#ifndef _TYPES_H
#define _TYPES_H

#include <vector>

/* The non-local directions must be numbered from 0 to n-1 */
typedef int direction;
const direction dir_north = 0;
const direction dir_east = 1;
const direction dir_south = 2;
const direction dir_west = 3;
const direction dir_local = 4;
const direction dir_invalid = -1;

/* Enum to identify the type of flit */
enum flittype { ft_invalid, ft_data };

/* Non-instantiable flit class. All flits inherit from this */
class flit {

public:
  const flittype get_type() const {
    return _type;
```

```
  }

  const bool is_last() const {
    return _last;
  }

  const direction get_direction() const {
    return _dir;
  }

  const int get_vc() const {
    return _vc;
  }

  void set_direction(direction d) {
    _dir = d;
  }

  void set_vc(int vc) {
    _vc = vc;
  }

protected:
  flit(const flittype ft, const bool last) : _type(ft), _last(last)
      {

  }

private:
  const flittype _type;
  const bool _last;
  int _vc;
  direction _dir;

};

/* Templated data flit */
template<typename T>
class flit_data : public flit {

public:
  T get_data() {
    return _data;
  }

  flit_data(const T data, const bool last) : flit(ft_data, last),
      _data(data) {

  }

private:
  T _data;

};

#endif
```

A.2 Components

A.2.1 arbiter.h

```
/*
  Header for the model of the ALG arbiter
*/

#ifndef _ARBITER_H
#define _ARBITER_H

#include <systemc>
#include "../interfaces.h"
#include "../types.h"

/* Number of inputs */
const unsigned int N = 8;

class arbiter : public sc_core::sc_module, public arbiter_if {

public:
  /* Ports */
  sc_core::sc_port<node_arbiter_if> nde;

  /* Inherited (virtual) functions */
  void arbitrate(flit*, int);

  SC_HAS_PROCESS(arbiter);

  arbiter(sc_core::sc_module_name n, const sc_core::sc_time& lct,
      const direction dir);
```

```cpp

private:
  void do_arbitrate();
  bool transmit();

  const sc_core::sc_time _link_cycle_time;

  /* Admission control and static priority queue */
  flit* _hold_q[N];
  flit* _spq[N];
  bool _hold_q_valid[N];
  bool _spq_valid[N];

  /* Busy indication, wait signals for admission control and time
      out event */
  bool _busy;
  bool _wait[N][N];
  sc_core::sc_event _time_out;

  const direction _dir;

};

#endif
```

A.2.2 arbiter.cpp

```cpp
/*
  Arbiter implementing ALG
*/

#include "arbiter.h"

void arbiter::arbitrate(flit* f, int vc) {
  _hold_q[vc] = f;
  _hold_q_valid[vc] = true;
  if (!_busy) {
    do_arbitrate();
  }
}

void arbiter::do_arbitrate() {
  bool vc_wait = true;
  bool graduated = false;

  /* Attempt transmit */
  bool tx = transmit();

  /* Graduate from hold queue to spq */
  for (int i = 0; i < N; ++i) {
    if (_hold_q_valid[i]) {
      if (i < 7) {
        vc_wait = true;
      } else {
        vc_wait = false;
      }
      for (int j = i+1; j < N; ++j) {
        vc_wait = vc_wait & _wait[i][j];
      }
      if (!vc_wait) {
        _spq[i] = _hold_q[i];
        _spq_valid[i] = true;
        _hold_q_valid[i] = false;
        /* Arbiter ready for new input */
        nde->arb_ready(i, _dir);
      }
      graduated = true;
    }
  }

  /* Transmit highest priority in spq, if first transmit failed */
  if (!tx && graduated) {
    transmit();
  }
}

bool arbiter::transmit() {
  for (int i = 0; i < N; ++i) {
    if (_spq_valid[i]) {
      _spq_valid[i] = false;
      nde->arb_send(_spq[i], _dir);
      _busy = true;
      _time_out.notify(_link_cycle_time);

      /* VC i must now wait for valid lower priorities in spq */
      for (int j = i+1; j < N; ++j) {
        _wait[i][j] = _spq_valid[j];
      }

```

```
      /* VCs 0..i-1 must no longer wait for VC i */
      for (int j = 0; j < i; ++j) {
        _wait[j][i] = false;
      }

      /* A flit was transmitted */
      return true;
    }
  }
  /* No flit transmitted */
  _busy = false;
  return false;
}

arbiter::arbiter(sc_core::sc_module_name n, const sc_core::sc_time&
    lct, const direction dir) : sc_core::sc_module(n),
    _link_cycle_time(sc_core::sc_time(5, sc_core::SC_NS)), _dir(
    dir) {
  _busy = false;
  for (int i = 0; i < N; ++i) {
    _hold_q_valid[i] = false;
    _spq_valid[i] = false;
    for (int j = 0; j < N; ++j) {
      _wait[i][j] = false;
    }
  }
  SC_METHOD(do_arbitrate);
  sensitive << _time_out;
  dont_initialize();
}
```

A.2.3 link.h

```
/*
  A link, implemented as a std::queue with constant delays
*/

#ifndef _LINK_H
#define _LINK_H

#include <systemc>
#include <queue>
#include "../types.h"
#include "../interfaces.h"

class mango_link : public link_transmitter_if, public
    link_receiver_if, public sc_core::sc_module {

public:
  /* Ports */
  sc_core::sc_port<node_transmitter_if> transmitter;
  sc_core::sc_port<node_receiver_if> receiver;

  /* Inherited (virtual) functions */
  void unlock_gs(int i);
  void credit_be(int i);
  void send(flit* f);

  SC_HAS_PROCESS(mango_link);

  /* Constructor */
  mango_link(sc_core::sc_module_name n, const direction dir, const
      sc_core::sc_time& send_delay, const sc_core::sc_time&
      unlock_gs_delay, const sc_core::sc_time& credit_be_delay);

private:
  /* Time-out functions */
  void do_unlock_gs();
  void do_credit_be();
  void do_send();

  /* Direction of link */
  const direction _dir;

  /* Delays */
  const sc_core::sc_time& _send_delay;
  const sc_core::sc_time& _unlock_gs_delay;
  const sc_core::sc_time& _credit_be_delay;

  /* Queues */
  std::queue<std::pair<flit*, sc_core::sc_time> > _send_fifo;
  std::queue<std::pair<int, sc_core::sc_time> > _unlock_gs_fifo;
  std::queue<std::pair<int, sc_core::sc_time> > _credit_be_fifo;

  /* Time-out events */
  sc_core::sc_event _e_send;
  sc_core::sc_event _e_unlock_gs;
  sc_core::sc_event _e_credit_be;

```

```
};

#endif
```

A.2.4 link.cpp

```
/*
  A link with constant delays
*/

#include "link.h"

void mango_link::unlock_gs(int i) {
  _unlock_gs_fifo.push(std::make_pair<int, sc_core::sc_time>(i,
        simcontext()->time_stamp() + _unlock_gs_delay));
  if (_unlock_gs_fifo.size() == 1) {
    _e_unlock_gs.notify(_unlock_gs_delay);
  }
}

void mango_link::credit_be(int i) {
  _credit_be_fifo.push(std::make_pair<int, sc_core::sc_time>(i,
        simcontext()->time_stamp() + _credit_be_delay));
  if (_credit_be_fifo.size() == 1) {
    _e_credit_be.notify(_credit_be_delay);
  }
}

void mango_link::send(flit* f) {
  _send_fifo.push(std::make_pair<flit*, sc_core::sc_time>(f,
        simcontext()->time_stamp() + _send_delay));
  if (_send_fifo.size() == 1) {
    _e_send.notify(_send_delay);
  }
}

/* Activated when unlock arrives */
void mango_link::do_unlock_gs() {
  transmitter->unlock_gs(_unlock_gs_fifo.front().first, _dir);
  _unlock_gs_fifo.pop();
  if (_unlock_gs_fifo.size() > 0) {
    _e_unlock_gs.notify(_unlock_gs_fifo.front().second - simcontext
          ()->time_stamp());
  }
}

/* Activated when credit arrives */
void mango_link::do_credit_be() {
  transmitter->credit_be(_credit_be_fifo.front().first, _dir);
  _credit_be_fifo.pop();
  if (_credit_be_fifo.size() > 0) {
    _e_credit_be.notify(_credit_be_fifo.front().second - simcontext
          ()->time_stamp());
  }
}

/* Activated when flit arrives */
void mango_link::do_send() {
  receiver->send(_send_fifo.front().first);
  _send_fifo.pop();
  if (_send_fifo.size() > 0) {
    _e_send.notify(_send_fifo.front().second - simcontext()->
          time_stamp());
  }
}

mango_link::mango_link(sc_core::sc_module_name n, const direction
      dir, const sc_core::sc_time& send_delay, const sc_core::
      sc_time& unlock_gs_delay, const sc_core::sc_time&
      credit_be_delay) : sc_core::sc_module(n), _dir(dir),
      _send_delay(send_delay), _unlock_gs_delay(unlock_gs_delay),
      _credit_be_delay(credit_be_delay) {
  SC_METHOD(do_send);
  dont_initialize();
  sensitive << _e_send;
  SC_METHOD(do_unlock_gs);
  dont_initialize();
  sensitive << _e_unlock_gs;
  SC_METHOD(do_credit_be);
  dont_initialize();
  sensitive << _e_credit_be;
}
```

A.2.5 vc.h

```
/*
  Virtual channels
*/
```

```

#ifndef _VC_H
#define _VC_H

#include <systemc>
#include <queue>

#include "../interfaces.h"
#include "../types.h"

/* General VC, not to be instantiated */
class vc : public sc_core::sc_module, public vc_transmitter_if,
    public vc_receiver_if {

public:
  vc(sc_core::sc_module_name);

};

/* Guaranteed Service VC */
class gs_vc : public vc {

public:
  sc_core::sc_port<arbiter_if> arb;
  sc_core::sc_port<node_internal_if> nde;

  /* Inherited functions */
  void send(flit*);
  void unlock();

  /* Constructor */
  gs_vc(sc_core::sc_module_name, const int id, const direction dir)
      ;

private:
  /* ID and direction */
  const int _id;
  const direction _dir;

  /* Latches and indications of validity */
  flit* _lock_box;
  bool _lock_box_valid;

  flit* _unlock_box;
  bool _unlock_box_valid;

  flit* _buffer;
  bool _buffer_valid;

};

#endif
```

A.2.6 vc.cpp

```
#include "vc.h"

/*
  General abstract/virtual VC bufffer
*/

vc::vc(sc_core::sc_module_name n) : sc_core::sc_module(n) {

}

/*
  Guarateed Service VC Buffer
*/

void gs_vc::send(flit* f) {
  if (!_buffer_valid) {
    if (!_lock_box_valid) {
      /* Progress directly to arbiter */
      _lock_box = f;
      _lock_box_valid = true;
      arb->arbitrate(_lock_box, _id);
    } else {
      /* Progress directly to buffer */
      _buffer = f;
      _buffer_valid = true;
    }
    /* Flit left unlockbox */
    nde->unlock(_id, _dir);
  } else {
    /* Wait in unlockbox */
    _unlock_box = f;
    _unlock_box_valid = true;
  }
}
```

```

void gs_vc::unlock() {
  if (_buffer_valid) {
    /* Propagate flit from buffer */
    _lock_box = _buffer;
    _buffer_valid = false;
    arb->arbitrate(_lock_box, _id);

    /* Possible flit in unlock_box may now propagate */
    if (_unlock_box_valid) {
      _buffer = _unlock_box;
      _buffer_valid = true;
      _unlock_box_valid = false;
      nde->unlock(_id, _dir);
    }
  } else {
    if (_unlock_box_valid) {
      /* Propagate flit from unlockbox */
      _lock_box = _unlock_box;
      _unlock_box_valid = false;
      arb->arbitrate(_lock_box, _id);
      nde->unlock(_id, _dir);
    } else {
      _lock_box_valid = false;
    }
  }
}

gs_vc::gs_vc(sc_core::sc_module_name n, const int id, const
     direction dir) : vc(n), _id(id), _dir(dir) {
  _lock_box_valid = false;
  _buffer_valid = false;
  _unlock_box_valid = false;
}
```

A.2.7 node.h

```
#ifndef _NODE_H
#define _NODE_H

#include <systemc>
#include "../types.h"
#include "../interfaces.h"
#include "arbiter.h"
#include "vc.h"

class node : public sc_core::sc_module, public node_transmitter_if,
      public node_receiver_if, public node_internal_if, public
     node_arbiter_if, public node_na_transmitter_if, public
     node_na_receiver_if {

public:
  sc_core::sc_port<link_transmitter_if> lnk_tx[4];
  sc_core::sc_port<link_receiver_if> lnk_rx[4];
  sc_core::sc_port<na_if> net_adap;

  /* Send (receive) */
  void send(flit*);

  /* Arbiter */
  void arb_send(flit*, const direction);
  void arb_ready(const int, const direction);

  /* External unlocks */
  void unlock_gs(const int, const direction);
  void credit_be(const int, const direction);

  /* Internal unlocks from VCs */
  void unlock(const int, const direction);
  void credit(const int, const direction);

  /* Network Adapter */
  void na_send(flit*);
  void na_unlock_gs(const int);

  /* Initialise tables */
  void set_routing_table(direction*, int*);
  void set_steer_table(direction*, int*);

  node(sc_core::sc_module_name, const sc_core::sc_time&);
  ~node();

private:
  /* Arbiter and virtual channels */
  arbiter* _arb[4];
  vc* _vcs[4][8];

  /* Routing tables */
  direction _routing_dir[5][8];
```

```
  int _routing_vc[5][8];

  /* Steer table */
  direction _steer_dir[5][8];
  int _steer_vc[5][8];

};

#endif
```

A.2.8 node.cpp

```
#include "node.h"

void node::send(flit* f) {
  direction df = f->get_direction();
  int vf = f->get_vc();
  /* Look up and set new destination */
  direction d = _routing_dir[(f->get_direction()+2)%4][f->get_vc()
      ];
  int v = _routing_vc[(f->get_direction()+2)%4][f->get_vc()];
  f->set_direction(d);
  f->set_vc(v);
  /* Send flit to its destination */
  if (d == dir_local) {
    net_adap->send(f);
  } else {
    if (d > dir_local) {
    } else {
      _vcs[f->get_direction()][f->get_vc()]->send(f);
    }
  }
}

void node::arb_ready(const int i, const direction d) {
  _vcs[d][i]->arb_ready();
}

void node::arb_send(flit* f, const direction d) {
  lnk_tx[d]->send(f);
}

/* Internal, from VC */
void node::credit(const int i, const direction d) {

}

/* External, from link */
void node::credit_be(const int i, const direction d) {
  _vcs[d][i]->unlock();
}

/* Internal, from VC */
void node::unlock(const int i, const direction d) {
  if (_steer_dir[d][i] == dir_local) {
    net_adap->unlock(_steer_vc[d][i]);
  } else {
    if (_steer_dir[d][i] > dir_local || _steer_vc[d][i] > 7) {
    } else {
      lnk_rx[_steer_dir[d][i]]->unlock_gs(_steer_vc[d][i]);
    }
  }
}

/* External, from link */
void node::unlock_gs(const int i, const direction d) {
  /* (d+2)%4 calculates the reverse direction */
  _vcs[(d+2)%4][i]->unlock();
}

/* Network Adapter */
void node::na_send(flit* f) {
  /* Lookup and set destination */
  direction d = _routing_dir[dir_local][f->get_vc()];
  int v = _routing_vc[dir_local][f->get_vc()];
  f->set_direction(d);
  f->set_vc(v);
  if (d > dir_local) {
  } else {
    _vcs[f->get_direction()][f->get_vc()]->send(f);
  }
}

void node::na_unlock_gs(const int i) {
  if (_steer_dir[dir_local][i] > dir_local || _steer_vc[dir_local][
      i] > 7) {
  } else {
    lnk_rx[_steer_dir[dir_local][i]]->unlock_gs(_steer_vc[dir_local
        ][i]);
```

```
  }
}

/* Initialisation */
void node::set_routing_table(direction* dir, int* vc) {
  for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 8; ++j, ++dir, ++vc) {
      _routing_dir[i][j] = *dir;
      _routing_vc[i][j] = *vc;
    }
  }
}

void node::set_steer_table(direction* dir, int* vc) {
  for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 8; ++j, ++dir, ++vc) {
      _steer_dir[i][j] = *dir;
      _steer_vc[i][j] = *vc;
    }
  }
}

node::node(sc_core::sc_module_name n, const sc_core::sc_time& lct)
    : sc_core::sc_module(n) {
  for (int i = 0; i < 4; ++i) {
    _arb[i] = new arbiter(gen_unique_name("ARB", true), lct, i);
    _arb[i]->nde(*this);
    for (int j = 0; j < 8; ++j) {
      _vcs[i][j] = new gs_vc(gen_unique_name("VC", true), j, i);
      _routing_dir[i][j] = (i+2)%4;
      _routing_vc[i][j] = j;
      _steer_dir[i][j] = (i+2)%4;
      _steer_vc[i][j] = j;
      ((gs_vc*) _vcs[i][j])->nde(*this);
      ((gs_vc*) _vcs[i][j])->arb(*(_arb[i]));
    }
  }
}

node::~node() {
  for (int i = 0; i < 4; ++i) {
    delete _arb[i];
    for (int j = 0; j < 8; ++j) {
      delete _vcs[i][j];
    }
  }
}
```

A.3 Test Files

A.3.1 mango_thesis_model.h

This is the top-level SystemC file, which contains the network and the conversion modules to the network adapters.

```
/*
  Top-level design file for model network of test system
*/

#ifndef _MANGO_THESIS_MODEL_H
#define _MANGO_THESIS_MODEL_H

#include <systemc>
#include "na_conv.h"
#include "link_sink.h"
#include "../components/node.h"
#include "../components/link.h"

class mango_thesis_model : public sc_core::sc_module {

public:
  /* Initiator */
  sc_core::sc_out<sc_dt::sc_lv<4> > RxReq_1;
  sc_core::sc_in<sc_dt::sc_lv<4> > RxAck_1;
  sc_core::sc_out<sc_dt::sc_lv<156> > RxData_1;
  sc_core::sc_out<sc_dt::sc_lv<4> > TxAck_1;
  sc_core::sc_in<sc_dt::sc_lv<4> > TxReq_1;
  sc_core::sc_in<sc_dt::sc_lv<156> > TxData_1;

  /* Target */
  sc_core::sc_out<sc_dt::sc_lv<4> > RxReq_2;
  sc_core::sc_in<sc_dt::sc_lv<4> > RxAck_2;
  sc_core::sc_out<sc_dt::sc_lv<156> > RxData_2;
  sc_core::sc_out<sc_dt::sc_lv<4> > TxAck_2;
  sc_core::sc_in<sc_dt::sc_lv<4> > TxReq_2;
  sc_core::sc_in<sc_dt::sc_lv<156> > TxData_2;

```

```
  sc_core::sc_in<sc_dt::sc_logic> reset;

 SC_HAS_PROCESS(mango_thesis_model);

  mango_thesis_model(sc_core::sc_module_name n) :
    sc_core::sc_module(n),
    na1("NA1"), na2("NA2"), lct(2.6, sc_core::SC_NS), sd(10.5,
         sc_core::SC_NS), ud(3., sc_core::SC_NS),
    ls1n("LS1N"), ls1w("LS1W"), ls1s("LS1S"), ls2n("LS2N"), ls2e("
         LS2E"), ls3w("LS3W"), ls3s("LS3S"), ls3e("LS3E") {

    /* Nodes */
    nd1 = new node("ND1", lct);
    nd2 = new node("ND2", lct);
    nd3 = new node("ND3", lct);

    /* Links */
    lk12 = new mango_link("LK12", dir_west, sd, ud, ud);
    lk21 = new mango_link("LK21", dir_east, sd, ud, ud);
    lk23 = new mango_link("LK23", dir_north, sd, ud, ud);
    lk32 = new mango_link("LK32", dir_south, sd, ud, ud);

    /* Dead end links */
    ls1n.nde_tx(*nd1); ls1n.nde_rx(*nd1);
    ls1w.nde_tx(*nd1); ls1w.nde_rx(*nd1);
    ls1s.nde_tx(*nd1); ls1s.nde_rx(*nd1);
    ls2n.nde_tx(*nd2); ls2n.nde_rx(*nd2);
    ls2e.nde_tx(*nd2); ls2e.nde_rx(*nd2);
    ls3w.nde_tx(*nd3); ls3w.nde_rx(*nd3);
    ls3s.nde_tx(*nd3); ls3s.nde_rx(*nd3);
    ls3e.nde_tx(*nd3); ls3e.nde_rx(*nd3);

    /* Connect initiator */
    na1.RxAck(RxAck_1);
    na1.RxReq(RxReq_1);
    na1.RxData(RxData_1);
    na1.TxAck(TxAck_1);
    na1.TxReq(TxReq_1);
    na1.TxData(TxData_1);
    na1.nde_tx(*nd1);
    na1.nde_rx(*nd1);

    /* Connect target */
    na2.RxAck(RxAck_2);
    na2.RxReq(RxReq_2);
    na2.RxData(RxData_2);
    na2.TxAck(TxAck_2);
    na2.TxReq(TxReq_2);
    na2.TxData(TxData_2);
    na2.nde_tx(*nd3);
    na2.nde_rx(*nd3);

    /* Connect nodes */
    nd1->lnk_tx[dir_north](ls1n);
    nd1->lnk_rx[dir_north](ls1n);
    nd1->lnk_tx[dir_west](ls1w);
    nd1->lnk_rx[dir_west](ls1w);
    nd1->lnk_tx[dir_south](ls1s);
    nd1->lnk_rx[dir_south](ls1s);
    nd1->lnk_tx[dir_east](*lk12);
    nd1->lnk_rx[dir_east](*lk21);
    nd1->net_adap(na1);

    nd2->lnk_tx[dir_north](ls2n);
    nd2->lnk_rx[dir_north](ls2n);
    nd2->lnk_tx[dir_east](ls2e);
    nd2->lnk_rx[dir_east](ls2e);
    nd2->lnk_tx[dir_west](*lk21);
    nd2->lnk_rx[dir_west](*lk12);
    nd2->lnk_tx[dir_south](*lk23);
    nd2->lnk_rx[dir_south](*lk32);
    nd2->net_adap(na1);

    nd3->lnk_tx[dir_north](*lk32);
    nd3->lnk_rx[dir_north](*lk23);
    nd3->lnk_tx[dir_east](ls3e);
    nd3->lnk_rx[dir_east](ls3e);
    nd3->lnk_tx[dir_south](ls3s);
    nd3->lnk_rx[dir_south](ls3s);
    nd3->lnk_tx[dir_west](ls3w);
    nd3->lnk_rx[dir_west](ls3w);
    nd3->net_adap(na2);

    /* Connect links */
    lk12->transmitter(*nd1);
    lk12->receiver(*nd2);

    lk21->transmitter(*nd2);
    lk21->receiver(*nd1);

```

```
lk23->transmitter(*nd2);
lk23->receiver(*nd3);

lk32->transmitter(*nd3);
lk32->receiver(*nd2);

/* Setup initiator NA routing tables */
direction na1dir[4];
na1dir[0] = dir_local;
na1dir[1] = dir_local;
na1dir[2] = dir_local;

int na1vc[4];
na1vc[0]=0;
na1vc[1]=1;
na1vc[2]=2;

na1.set_dir(na1dir);
na1.set_vc(na1vc);

/* Setup target NA routing tables */
direction na2dir[4];
na2dir[0] = dir_local;
na2dir[3] = dir_local;

int na2vc[4];
na2vc[0] = 0;
na2vc[3] = 3;

na2.set_dir(na2dir);
na2.set_vc(na2vc);

/* Setup node1 routing tables */
direction nd1dir[5][8];
nd1dir[dir_local][0] = dir_east;
nd1dir[dir_local][1] = dir_east;
nd1dir[dir_local][2] = dir_east;
nd1dir[dir_east][0] = dir_local;
nd1dir[dir_east][7] = dir_local;

int nd1vc[5][8];
nd1vc[dir_local][0] = 7;
nd1vc[dir_local][1] = 0;
nd1vc[dir_local][2] = 3;
nd1vc[dir_east][0] = 1;
nd1vc[dir_east][7] = 0;

/* Setup node1 steer tables */
direction nd1sdir[5][8];
nd1sdir[dir_local][0] = dir_east;
nd1sdir[dir_local][1] = dir_east;
nd1sdir[dir_local][2] = dir_east;
nd1sdir[dir_local][3] = dir_east;
nd1sdir[dir_east][0] = dir_local;
nd1sdir[dir_east][7] = dir_local;

int nd1svc[5][8];
nd1svc[dir_local][0] = 7;
nd1svc[dir_local][1] = 0;
nd1svc[dir_local][2] = 1;
nd1svc[dir_local][3] = 1;
nd1svc[dir_east][0] = 1;
nd1svc[dir_east][7] = 0;

nd1->set_routing_table(&(nd1dir[0][0]), &(nd1vc[0][0]));
nd1->set_steer_table(&(nd1sdir[0][0]), &(nd1svc[0][0]));

/* Setup node2 routing tables */
direction nd2dir[5][8];
nd2dir[dir_west][7] = dir_south;
nd2dir[dir_west][3] = dir_south;
nd2dir[dir_west][0] = dir_south;
nd2dir[dir_south][7] = dir_west;
nd2dir[dir_south][2] = dir_west;

int nd2vc[5][8];
nd2vc[dir_west][7] = 7;
nd2vc[dir_west][3] = 5;
nd2vc[dir_west][0] = 0;
nd2vc[dir_south][7] = 7;
nd2vc[dir_south][2] = 0;

/* Setup node2 steer tables */
direction nd2sdir[5][8];
nd2sdir[dir_south][7] = dir_west;
nd2sdir[dir_south][5] = dir_west;
nd2sdir[dir_south][0] = dir_west;
nd2sdir[dir_west][7] = dir_south;
nd2sdir[dir_west][0] = dir_south;

```

```
    int nd2svc[5][8];
    nd2svc[dir_south][7] = 7;
    nd2svc[dir_south][5] = 3;
    nd2svc[dir_south][0] = 0;
    nd2svc[dir_west][7] = 7;
    nd2svc[dir_west][0] = 2;

    nd2->set_routing_table(&(nd2dir[0][0]), &(nd2vc[0][0]));
    nd2->set_steer_table(&(nd2sdir[0][0]), &(nd2svc[0][0]));

    /* Setup node3 routing tables */
    direction nd3dir[5][8];
    nd3dir[dir_north][7] = dir_local;
    nd3dir[dir_north][5] = dir_local;
    nd3dir[dir_north][0] = dir_local;
    nd3dir[dir_local][3] = dir_north;
    nd3dir[dir_local][0] = dir_north;

    int nd3vc[5][8];
    nd3vc[dir_north][7] = 0;
    nd3vc[dir_north][5] = 2;
    nd3vc[dir_north][0] = 1;
    nd3vc[dir_local][0] = 7;
    nd3vc[dir_local][3] = 2;

    /* Setup node3 steer tables */
    direction nd3sdir[5][8];
    nd3sdir[dir_local][0] = dir_north;
    nd3sdir[dir_local][2] = dir_north;
    nd3sdir[dir_local][1] = dir_north;
    nd3sdir[dir_local][3] = dir_north;
    nd3sdir[dir_north][7] = dir_local;
    nd3sdir[dir_north][2] = dir_local;

    int nd3svc[5][8];
    nd3svc[dir_local][0] = 7;
    nd3svc[dir_local][2] = 5;
    nd3svc[dir_local][1] = 0;
    nd3svc[dir_local][3] = 6;
    nd3svc[dir_north][7] = 0;
    nd3svc[dir_north][2] = 3;

    nd3->set_routing_table(&(nd3dir[0][0]), &(nd3vc[0][0]));
    nd3->set_steer_table(&(nd3sdir[0][0]), &(nd3svc[0][0]));

  }

  ~mango_thesis_model() {
    delete nd1, nd2, nd3, lk12, lk21, lk23, lk32;
  }

private:
  node* nd1;
  node* nd2;
  node* nd3;
  mango_link* lk12;
  mango_link* lk21;
  mango_link* lk23;
  mango_link* lk32;
  na_conv na1, na2;
  link_sink ls1n, ls1w, ls1s, ls2n, ls2e, ls3w, ls3s, ls3e;
  sc_core::sc_time lct, sd, ud;

};

#endif
```

A.3.2 mango_thesis_model.cpp

```
#include "mango_thesis_model.h"

#include <systemc.h>

SC_MODULE_EXPORT(mango_thesis_model);
```

A.3.3 na_conv.h

This file contains the conversion module to the network adapters.

```
/*
  Conversion module for network adapters to model
*/

#ifndef _NA_CONV_H
#define _NA_CONV_H

```

```
#include <systemc>
#include "../interfaces.h"
#include "../types.h"

class na_conv : public sc_core::sc_module, public na_if {

public:
  /* Ports */
  sc_core::sc_out<sc_dt::sc_lv<4> > RxReq;
  sc_core::sc_in<sc_dt::sc_lv<4> > RxAck;
  sc_core::sc_out<sc_dt::sc_lv<156> > RxData;
  sc_core::sc_out<sc_dt::sc_lv<4> > TxAck;
  sc_core::sc_in<sc_dt::sc_lv<4> > TxReq;
  sc_core::sc_in<sc_dt::sc_lv<156> > TxData;

  /* Signals */
  sc_core::sc_signal<sc_dt::sc_logic> s_RxReq[4], s_RxAck[4],
       s_TxAck[4], s_TxReq[4];
  sc_core::sc_signal<sc_dt::sc_lv<39> > s_RxData[4], s_TxData[4];

  sc_core::sc_port<node_na_transmitter_if> nde_tx;
  sc_core::sc_port<node_na_receiver_if> nde_rx;

  SC_HAS_PROCESS(na_conv);

  /* On TxReq or RxAck */
  void do_input() {
    for (int i = 0; i < 4; ++i) {
      s_TxData[i] = TxData.read().range(39*(i+1)-1, 39*i);
      s_RxAck[i] = RxAck.read().get_bit(i);
      s_TxReq[i] = TxReq.read().get_bit(i);
    }
  }

  /* On TxAck, RxData */
  void do_output() {
    sc_dt::sc_lv<156> v_RxData;
    sc_dt::sc_lv<4> v_TxAck;
    for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 39; ++j) {
        v_RxData.set_bit(39*i+j, s_RxData[i].read().get_bit(j));
      }
      v_TxAck.set_bit(i, s_TxAck[i].read().value());
    }
    RxData.write(v_RxData);
    TxAck.write(v_TxAck);
    _e.notify(1.5, sc_core::SC_NS);
  }

  /* On RxReq */
  void do_delayed_output() {
    sc_dt::sc_lv<4> v_RxReq;
    for (int i = 0; i < 4; ++i) {
      v_RxReq.set_bit(i, s_RxReq[i].read().value());
    }
    RxReq.write(v_RxReq);
  }

  /* On TxReq0, BE */
  void do_TxReq_0() {
    if (s_TxReq[0].posedge()) {
      if (!first_BE_flit) {
        /* Not first BE flit */
        TxReqQ[0] = (flit*) (new flit_data<sc_dt::sc_lv<39> >(
             TxData.read().range(38, 0), false));
        TxReqQ[0]->set_direction(_dir[0]);
        TxReqQ[0]->set_vc(_vc[0]);
        nde_rx->na_send(TxReqQ[0]);
      } else {
        /* First BE flit */
        sc_dt::sc_lv<38> s_temp = TxData.read().range(37, 0);
        s_temp.lrotate(10);
        sc_dt::sc_lv<39> s_temp2;
        s_temp2.set_bit(38, TxData.read().get_bit(38));
        for (int i = 0; i < 38; ++i) {
          s_temp2.set_bit(i, s_temp.get_bit(i));
        }
        s_temp.set_bit(38, TxData.read().get_bit(38));
        TxReqQ[0] = (flit*) (new flit_data<sc_dt::sc_lv<39> >(
             s_temp, false));
        TxReqQ[0]->set_direction(_dir[0]);
        TxReqQ[0]->set_vc(_vc[0]);
        nde_rx->na_send(TxReqQ[0]);
      }
      /* Update first_BE_flit */
      if (TxData.read().get_bit(38) == sc_dt::sc_logic(1)) {
        first_BE_flit = true;
      } else {
        first_BE_flit = false;
      }
```

```
    } else if (s_TxReq[0].negedge()) {
      /* Delay for TxAck deassert */
      next_trigger(0.4, sc_core::SC_NS);
    } else {
      /* Deassert TxAck */
      s_TxAck[0].write(s_TxReq[0].read());
    }
  }

  /* On TxReq1, GS 1 */
  void do_TxReq_1() {
    if (s_TxReq[1].posedge()) {
      /* Request */
      TxReqQ[1] = (flit*) (new flit_data<sc_dt::sc_lv<39> >(TxData.
           read().range(77, 39), false));
      TxReqQ[1]->set_direction(_dir[1]);
      TxReqQ[1]->set_vc(_vc[1]);
      nde_rx->na_send(TxReqQ[1]);
    } else if (s_TxReq[1].negedge()) {
      /* TxReq deasserted, delay for TxAck */
      next_trigger(0.4, sc_core::SC_NS);
    } else {
      /* Deasser TxAck */
      s_TxAck[1].write(s_TxReq[1].read());
    }
  }

  /* On TxReq2, GS 2 */
  void do_TxReq_2() {
    if (s_TxReq[2].posedge()) {
      TxReqQ[2] = (flit*) (new flit_data<sc_dt::sc_lv<39> >(TxData.
           read().range(116, 78), false));
      TxReqQ[2]->set_direction(_dir[2]);
      TxReqQ[2]->set_vc(_vc[2]);
      nde_rx->na_send(TxReqQ[2]);
    } else if (s_TxReq[2].negedge()) {
      next_trigger(0.4, sc_core::SC_NS);
    } else {
      s_TxAck[2].write(s_TxReq[2].read());
    }
  }

  /* On TxReq3, GS 3 */
  void do_TxReq_3() {
    if (s_TxReq[3].posedge()) {
      TxReqQ[3] = (flit*) (new flit_data<sc_dt::sc_lv<39> >(TxData.
           read().range(155, 117), false));
      TxReqQ[3]->set_direction(_dir[3]);
      TxReqQ[3]->set_vc(_vc[3]);
      nde_rx->na_send(TxReqQ[3]);
    } else if (s_TxReq[3].negedge()) {
      next_trigger(0.4, sc_core::SC_NS);
    } else {
      s_TxAck[3].write(s_TxReq[3].read());
    }
  }

  /* On RxAck0, BE */
  void do_RxAck_0() {
    if (s_RxAck[0].posedge()) {
      next_trigger(3.6, sc_core::SC_NS);
    } else if (s_RxAck[0].negedge()) {
      nde_tx->na_unlock_gs(0);
    } else {
      s_RxReq[0].write(sc_dt::sc_logic(0));
    }
  }

  /* On RxAck1, GS 1 */
  void do_RxAck_1() {
    if (s_RxAck[1].posedge()) {
      next_trigger(0.7, sc_core::SC_NS);
    } else if (s_RxAck[1].negedge()) {
      nde_tx->na_unlock_gs(1);
    } else {
      s_RxReq[1].write(sc_dt::sc_logic(0));
    }
  }

  /* On RxAck1, GS 2 */
  void do_RxAck_2() {
    if (s_RxAck[2].posedge()) {
      next_trigger(0.7, sc_core::SC_NS);
    } else if (s_RxAck[2].negedge()) {
      nde_tx->na_unlock_gs(2);
    } else {
      s_RxReq[2].write(sc_dt::sc_logic(0));
    }
  }
```

```
/* On RxAck1, GS 3 */
void do_RxAck_3() {
  if (s_RxAck[3].posedge()) {
    next_trigger(0.7, sc_core::SC_NS);
  } else if (s_RxAck[3].negedge()) {
    nde_tx->na_unlock_gs(3);
  } else {
    s_RxReq[3].write(sc_dt::sc_logic(0));
  }
}

/* Flit arrives from node */
void send(flit* f) {
  flit_data<sc_dt::sc_lv<39> >* fd = (flit_data<sc_dt::sc_lv<39>
        >*) f;
  s_RxData[fd->get_vc()].write(fd->get_data());
  s_RxReq[fd->get_vc()].write(sc_dt::sc_logic(1));
  delete f;
}

/* Init tables */
void set_dir(direction* d) {
  for (int i = 0; i < 4; ++i) {
    _dir[i] = *(d+i);
  }
}

void set_vc(int* d) {
  for (int i = 0; i < 4; ++i) {
    _vc[i] = *(d+i);
  }
}

/* Unlock channel i */
void unlock(const int i) {
  _e2[i].notify(5.1, sc_core::SC_NS);
}

void do_unlock0() {
  s_TxAck[0].write(sc_dt::sc_logic(1));
}

void do_unlock1() {
  s_TxAck[1].write(sc_dt::sc_logic(1));
}

void do_unlock2() {
  s_TxAck[2].write(sc_dt::sc_logic(1));
}

void do_unlock3() {
  s_TxAck[3].write(sc_dt::sc_logic(1));
}

na_conv(sc_core::sc_module_name) {
  SC_METHOD(do_input);
  sensitive << RxAck << TxReq;
  dont_initialize();
  SC_METHOD(do_output);
  for (int i = 0; i < 4; ++i) {
    sensitive << s_RxReq[i] << s_TxAck[i] << s_RxData[i];
  }
  dont_initialize();
  SC_METHOD(do_TxReq_0);
  sensitive << s_TxReq[0];
  dont_initialize();
  SC_METHOD(do_TxReq_1);
  sensitive << s_TxReq[1];
  dont_initialize();
  SC_METHOD(do_TxReq_2);
  sensitive << s_TxReq[2];
  dont_initialize();
  SC_METHOD(do_TxReq_3);
  sensitive << s_TxReq[3];
  dont_initialize();
  SC_METHOD(do_RxAck_0);
  sensitive << s_RxAck[0];
  dont_initialize();
  SC_METHOD(do_RxAck_1);
  sensitive << s_RxAck[1];
  dont_initialize();
  SC_METHOD(do_RxAck_2);
  sensitive << s_RxAck[2];
  dont_initialize();
  SC_METHOD(do_RxAck_3);
  sensitive << s_RxAck[3];
  dont_initialize();
  SC_METHOD(do_delayed_output);
  sensitive << _e;
  dont_initialize();
```

```
    SC_METHOD(do_unlock0);
     sensitive << _e2[0];
     dont_initialize();
    SC_METHOD(do_unlock1);
     sensitive << _e2[1];
     dont_initialize();
    SC_METHOD(do_unlock2);
     sensitive << _e2[2];
     dont_initialize();
    SC_METHOD(do_unlock3);
     sensitive << _e2[3];
     dont_initialize();
    s_RxData[0].write(sc_dt::sc_lv<39>(0));
    s_RxData[1].write(sc_dt::sc_lv<39>(0));
    s_RxData[2].write(sc_dt::sc_lv<39>(0));
    s_RxData[3].write(sc_dt::sc_lv<39>(0));
    s_RxReq[0].write(sc_dt::sc_logic(0));
    s_RxReq[1].write(sc_dt::sc_logic(0));
    s_RxReq[2].write(sc_dt::sc_logic(0));
    s_RxReq[3].write(sc_dt::sc_logic(0));

    first_BE_flit = true;
  }

private:
  flit* TxReqQ[4];
  direction _dir[4];
  int _vc[4];
  sc_core::sc_event _e, _e2[4];
  bool first_BE_flit;

};

#endif
```

A.3.4 link_sink.h

This file contains a module which sits on the unconnected node outputs. It produces a warning if it receives a flit or an unlock.

```
/*
  Sink for dead end outputs from nodes
*/

#ifndef _LINK_SINK_H
#define _LINK_SINK_H

#include <systemc>
#include "../interfaces.h"
#include <ios>

class link_sink : public sc_core::sc_module, public
     link_transmitter_if, public link_receiver_if {

public:
  /* Ports */
  sc_core::sc_port<node_transmitter_if> nde_tx;
  sc_core::sc_port<node_receiver_if> nde_rx;

  /* Produce warnings if accessed */
  void unlock_gs(int i) {
    std::cerr << "Unlock sent to sink\n";
  }

  void credit_be(int i) {
    std::cerr << "Credit sent to sink\n";
  }

  void send(flit* f) {
    std::cerr << "Flit sent to sink\n";
  }

  link_sink(sc_core::sc_module_name n) : sc_core::sc_module(n) {}

};

#endif
```

A.3.5 OCP_cores.h

```
/*
  OCP cores used for test system
*/

```

```
#ifndef _OCP_CORES_H
#define _OCP_CORES_H

#include <systemc>
#include <iostream>

/* OCP Master Commands */
const sc_dt::sc_lv<3> MCmd_idle(0);
const sc_dt::sc_lv<3> MCmd_write(1);
const sc_dt::sc_lv<3> MCmd_read(2);

class OCP_cores : public sc_core::sc_module {

public:
  // Master ports
  sc_core::sc_out<bool> M_OCPClk;
  sc_core::sc_out<sc_dt::sc_logic> M_Reset_n;
  sc_core::sc_out<sc_dt::sc_lv<3> > M_OCPMCmd;
  sc_core::sc_in<sc_dt::sc_logic> M_OCPSCmdAccept;
  sc_core::sc_out<sc_dt::sc_lv<32> > M_OCPMAddr;
  sc_core::sc_out<sc_dt::sc_lv<32> > M_OCPMData;
  sc_core::sc_out<sc_dt::sc_lv<8> > M_OCPMBurstLength;
  sc_core::sc_out<sc_dt::sc_lv<3> > M_OCPMBurstSeq;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMBurstSingleReq;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMBurstPrecise;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMReqLast;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMDataLast;
  sc_core::sc_out<sc_dt::sc_lv<2> > M_OCPMConnID;
  sc_core::sc_out<sc_dt::sc_lv<3> > M_OCPMThreadID;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMDataValid;
  sc_core::sc_in<sc_dt::sc_logic> M_OCPSDataAccept;
  sc_core::sc_in<sc_dt::sc_lv<2> > M_OCPSResp;
  sc_core::sc_out<sc_dt::sc_logic> M_OCPMRespAccept;
  sc_core::sc_in<sc_dt::sc_lv<32> > M_OCPSData;
  sc_core::sc_in<sc_dt::sc_logic> M_OCPSRespLast;
  sc_core::sc_in<sc_dt::sc_lv<3> > M_OCPSThreadID;
  sc_core::sc_out<sc_dt::sc_lv<3> > M_OCPMDataThreadID;
  sc_core::sc_in<sc_dt::sc_logic> M_OCPSInterrupt;

  // Slave ports
  sc_core::sc_out<bool> S_OCPClk;
  sc_core::sc_out<sc_dt::sc_logic> S_Reset_n;

  sc_core::sc_in<sc_dt::sc_lv<3> > S_OCPMCmd;
  sc_core::sc_out<sc_dt::sc_logic> S_OCPSCmdAccept;
  sc_core::sc_in<sc_dt::sc_lv<32> > S_OCPMAddr;
  sc_core::sc_in<sc_dt::sc_lv<32> > S_OCPMData;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMDataValid;
  sc_core::sc_out<sc_dt::sc_logic> S_OCPSDataAccept;
  sc_core::sc_in<sc_dt::sc_lv<8> > S_OCPMBurstLength;
  sc_core::sc_in<sc_dt::sc_lv<3> > S_OCPMBurstSeq;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMBurstSingleReq;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMBurstPrecise;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMReqLast;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMDataLast;
  sc_core::sc_in<sc_dt::sc_lv<2> > S_OCPMThreadID;
  sc_core::sc_in<sc_dt::sc_lv<2> > S_OCPMDataThreadID;
  sc_core::sc_out<sc_dt::sc_logic> S_OCPSRespLast;
  sc_core::sc_out<sc_dt::sc_lv<2> > S_OCPSThreadID;
  sc_core::sc_out<sc_dt::sc_lv<2> > S_OCPSResp;
  sc_core::sc_out<sc_dt::sc_lv<32> > S_OCPSData;
  sc_core::sc_in<sc_dt::sc_logic> S_OCPMRespAccept;
  sc_core::sc_out<sc_dt::sc_logic> S_OCPRespDone;
  sc_core::sc_out<sc_dt::sc_logic> S_OCPSInterrupt;

  void ocp_master();
  void ocp_slave();
  void wr_m_clock();
  void wr_s_clock();

  SC_HAS_PROCESS(OCP_cores);

  OCP_cores(sc_core::sc_module_name);

  /* Clocks */
  bool m_clock;
  bool s_clock;

  /* Small delay */
  sc_core::sc_time ch0in_speed;

private:
  void read_vectors(const char*, std::vector<sc_dt::sc_lv<76> >&);
  void do_ocp_master(sc_dt::sc_lv<76>, bool);
  void do_print();

  /* Test vectors */
  std::vector<sc_dt::sc_lv<76> > prog_vecs;
  std::vector<sc_dt::sc_lv<76> > test_vecs;
```

```
  /* Logging of transmission and reception times */
  std::vector<double> tm_time;
  std::vector<double> rs_time;
  std::vector<double> ts_time;
  std::vector<double> rm_time;
  sc_dt::sc_lv<76> idle_vec;
  bool int_done;
  sc_core::sc_time int_time;

  /* Disable clocks to stop simulation */
  bool clk_enable;
};

#endif
```

A.3.6 OCP_cores.cpp

```
#include "OCP_cores.h"

/* OCP Master */
void OCP_cores::ocp_master() {
  std::vector<sc_dt::sc_lv<76> >::iterator iter;

  /* Initialise port values */
  M_OCPMCmd.write(MCmd_idle);
  M_OCPMAddr.write(sc_dt::sc_lv<32>(0));
  M_OCPMData.write(sc_dt::sc_lv<32>(0));
  M_OCPMBurstLength.write(sc_dt::sc_lv<8>(1));
  M_OCPMBurstSeq.write(sc_dt::sc_lv<3>(0));
  M_OCPMBurstSingleReq.write(sc_dt::sc_logic(0));
  M_OCPMBurstPrecise.write(sc_dt::sc_logic(0));
  M_OCPMReqLast.write(sc_dt::sc_logic(0));
  M_OCPMDataLast.write(sc_dt::sc_logic(0));
  M_OCPMThreadID.write(sc_dt::sc_lv<3>(0));
  M_OCPMDataValid.write(sc_dt::sc_logic(0));
  M_OCPMConnID.write(sc_dt::sc_lv<2>(0));
  M_OCPMRespAccept.write(sc_dt::sc_logic(0));
  M_OCPMDataThreadID.write(sc_dt::sc_lv<3>(0));

  M_Reset_n.write(sc_dt::sc_logic(1));
  wait(); wait();
  wait(); wait();
  wait(ch0in_speed);

  /* Reset */
  M_Reset_n.write(sc_dt::sc_logic(0));

  /* Wait some time */
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();
  wait(); wait();

  wait(ch0in_speed);

  /* Done reset */
  M_Reset_n.write(sc_dt::sc_logic(1));

  wait(); wait();
  wait(); wait();
  wait(ch0in_speed);

  // Programming phase
  for (iter = prog_vecs.begin(); iter != prog_vecs.end(); ++iter) {
    do_ocp_master(*iter, false);
  }

  // Wait phase
  do_ocp_master(idle_vec, false);
  for (int i = 0; i < 100; ++i) {
    wait(); wait();
  }

  // Data phase
  std::cout << simcontext()->time_stamp() << " : Network programmed
      ... Activity\n";
  int nr;
  for (nr = 0, iter = test_vecs.begin(); iter != test_vecs.end();
      ++iter, ++nr) {
    if (nr % 1000 == 0) {
      std::cout << ".";
```

```
      }
      do_ocp_master(*iter, true);
    }
    std::cout << "\n";

    /* Output log to files */
    do_print();

    /* Stop simulation */
    clk_enable = false;
    std::cout << "Simulation done... stoping\n";
}

/* Do one OCP transaction */
void OCP_cores::do_ocp_master(sc_dt::sc_lv<76> vec, bool record) {
    /* Write Master Command */
    M_OCPMCmd.write(vec.range(75, 73));

    if (vec.range(75, 73) == MCmd_idle) {
      wait(); wait();
      wait(ch0in_speed);
    } else if (vec.range(75, 73) == MCmd_write) {
      M_OCPMConnID.write(vec.range(71, 70));
      M_OCPMThreadID.write(vec.range(68,66));
      M_OCPMAddr.write(vec.range(64, 33));
      M_OCPMData.write(vec.range(31, 0));
      if (record) {
        tm_time.push_back(simcontext()->time_stamp().to_double());
      }
      wait(); wait();
      while(M_OCPSCmdAccept.read() != sc_dt::sc_logic(1)) {
        wait(); wait();
      }

      wait(ch0in_speed);
    } else if (vec.range(75, 73) == MCmd_read) {
      M_OCPMConnID.write(vec.range(71, 70));
      M_OCPMThreadID.write(vec.range(68,66));
      M_OCPMAddr.write(vec.range(64, 33));
      M_OCPMData.write(vec.range(31, 0));
      if (record) {
        tm_time.push_back(simcontext()->time_stamp().to_double());
      }
      wait(); wait();
      while(M_OCPSCmdAccept.read() != sc_dt::sc_logic(1)) {
        wait(); wait();
      }

      wait(ch0in_speed);
      M_OCPMCmd.write(MCmd_idle);

      // Wait for response
      while (M_OCPSResp.read() == sc_dt::sc_lv<2>(0)) {
        wait(); wait();
      }
      if (record) {
        rm_time.push_back(simcontext()->time_stamp().to_double());
      }
      M_OCPMRespAccept.write(sc_dt::sc_logic(1));

      /* Stop if request failed */
      if (M_OCPSResp.read() == sc_dt::sc_lv<2>(3)) {
        sc_core::sc_stop();
      } else if (M_OCPSResp.read() == sc_dt::sc_lv<2>(2)) {
        sc_core::sc_stop();
      }
      while (M_OCPSResp.read() != sc_dt::sc_lv<2>(0)) {
        wait(M_OCPSResp.default_event());
      }
      M_OCPMRespAccept.write(sc_dt::sc_logic(0));
      wait(ch0in_speed);
    }
}


/* OCP Slave Core */
void OCP_cores::ocp_slave() {
    /* Initialise port values */
    S_OCPSCmdAccept.write(sc_dt::sc_logic(0));
    S_OCPSDataAccept.write(sc_dt::sc_logic(0));
    S_OCPSRespLast.write(sc_dt::sc_logic(0));
    S_OCPSThreadID.write(sc_dt::sc_lv<2>(0));
    S_OCPSResp.write(sc_dt::sc_lv<2>(0));
    S_OCPSData.write(sc_dt::sc_lv<32>(0));
    S_OCPRespDone.write(sc_dt::sc_logic(0));
    S_OCPSInterrupt.write(sc_dt::sc_logic(0));

    S_Reset_n.write(sc_dt::sc_logic(1));

    wait(); wait();
```

```
    wait(); wait();
    wait(ch0in_speed);

    /* Reset */
    S_Reset_n.write(sc_dt::sc_logic(0));

    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();
    wait(); wait();

    wait(ch0in_speed);

    S_Reset_n.write(sc_dt::sc_logic(1));
    /* Reset done */

    wait(); wait();
    wait(); wait();

    /* Wait for transaction */
    while (true) {
      wait(S_OCPMCmd.default_event());
      if (S_OCPMCmd.read() == MCmd_idle) {
      } else if (S_OCPMCmd.read() == MCmd_write) {
        rs_time.push_back(simcontext()->time_stamp().to_double());
        wait(ch0in_speed);
        S_OCPSCmdAccept.write(sc_dt::sc_logic(1));
        S_OCPSThreadID.write(S_OCPMThreadID.read());
        wait(); wait();
        wait(ch0in_speed);
        S_OCPSCmdAccept.write(sc_dt::sc_logic(0));
      } else if (S_OCPMCmd.read() == MCmd_read) {
        rs_time.push_back(simcontext()->time_stamp().to_double());
        wait(ch0in_speed);
        S_OCPSCmdAccept.write(sc_dt::sc_logic(1));
        S_OCPSThreadID.write(S_OCPMThreadID.read().to_int());
        wait(); wait();
        ts_time.push_back(simcontext()->time_stamp().to_double());
        wait(ch0in_speed);
        S_OCPSCmdAccept.write(sc_dt::sc_logic(0));
        S_OCPSData.write(S_OCPMAddr.read());
        S_OCPSResp.write(sc_dt::sc_lv<2>(1));
        S_OCPSRespLast.write(sc_dt::sc_logic(1));
        while (S_OCPMRespAccept.read() != sc_dt::sc_logic(1)) {
          wait(); wait();
        }
        wait(ch0in_speed);
        S_OCPSData.write(sc_dt::sc_lv<32>(0));
        S_OCPSResp.write(sc_dt::sc_lv<2>(0));
        S_OCPSRespLast.write(sc_dt::sc_logic(0));
      } else {
        /* Unknown command */
      }

  }
}

/* OCP Master clock, period 4 ns */
void OCP_cores::wr_m_clock() {
  m_clock = !m_clock;
  M_OCPClk.write(m_clock);
  if (clk_enable) {
    next_trigger(2., sc_core::SC_NS);
  }
}

/* OCP Slave clock, period 3 ns */
void OCP_cores::wr_s_clock() {
  s_clock = !s_clock;
  S_OCPClk.write(s_clock);
  if (clk_enable) {
    next_trigger(1.5, sc_core::SC_NS);
  }
}

OCP_cores::OCP_cores(sc_core::sc_module_name n) : sc_core::
      sc_module(n), ch0in_speed(0.4, sc_core::SC_NS), idle_vec(0),
      int_time(2000, sc_core::SC_NS), int_done(false) {
  /* Initialise values */
  clk_enable = true;
  m_clock = false;
  s_clock = false;

  /* Read programming and test vectors */
  read_vectors("prog_vecs.in", prog_vecs);
```

```
  read_vectors("test_vecs.in", test_vecs);

  /* Setup idle vector */
  idle_vec.set_bit(72, sc_dt::sc_logic(1).value());
  idle_vec.set_bit(69, sc_dt::sc_logic(1).value());
  idle_vec.set_bit(65, sc_dt::sc_logic(1).value());
  idle_vec.set_bit(32, sc_dt::sc_logic(1).value());

  SC_THREAD(ocp_master);
  sensitive << M_OCPClk;
  SC_THREAD(ocp_slave);
  sensitive << S_OCPClk;
  SC_METHOD(wr_m_clock);
  SC_METHOD(wr_s_clock);
}

/* Output logged times to files */
void OCP_cores::do_print() {
  std::ofstream req("req_times.out");
  std::ofstream resp("resp_times.out");
  std::vector<double>::const_iterator iter1, iter2;


  if (!req.is_open()) {
    /* Error opening file */
  }
  for (iter1 = tm_time.begin(), iter2 = rs_time.begin(); iter1 !=
        tm_time.end() && iter2 != rs_time.end(); ++iter1, ++iter2) {
    req << (int) *iter1 << " " << (int) *iter2 << "\n";
  }
  req.close();


  if (!resp.is_open()) {
    /* Error opening file */
  }
  for (iter1 = ts_time.begin(), iter2 = rm_time.begin(); iter1 !=
        ts_time.end() && iter2 != rm_time.end(); ++iter1, ++iter2) {
    resp << (int) *iter1 << " " << (int) *iter2 << "\n";
  }
  resp.close();

}

/* Read vectors into v */
void OCP_cores::read_vectors(const char* fname, std::vector<sc_dt::
      sc_lv<76> >& v) {
  char c;
  int nr = 0;

  std::ifstream f(fname);
  if (!f.is_open()) {
    std::cout << "Error opening vector file " << f << "... stoping
          .\n";
    sc_core::sc_stop();
    return;
  }
  std::cout << "Reading file " << fname << "... ";
  while (!f.eof()) {
    sc_dt::sc_lv<76> tmp(0);
    for (int i = 75; i >= 0; --i) {
      if (!(f >> c)) {
        f.close();
        std::cout << nr << " entries found\n";
        return;
      }
      tmp.set_bit(i, sc_dt::sc_logic(c).value());
    }
    v.push_back(tmp);
    ++nr;
  }
  f.close();
  std::cout << nr << " entries found\n";
}

#include <systemc.h>
SC_MODULE_EXPORT(OCP_cores);
```

A.3.7 gen_test_vecs.cpp

```
/*
  Small program to generate random test vectors
*/

#include <fstream>
#include <iostream>
#include <string>

const int w0 = 0;
```

```
const int r0 = 0;
const int w1 = 10000;
const int r1 = 5000;
const int w2 = 0;
const int r2 = 0;

const double idle_chance = 0.8;

const std::string idle("00010010001"
                 "00000000000000000000000000000001"
                 "00000000000000000000000000000000\n");

int main(int argc, char* argv[]) {
  unsigned int r;
  std::cout << RAND_MAX << "\n";
  std::ofstream f("test_vecs.in");
  if (!f.is_open());
  /* Writes on GS1 */
  for (int i = 0; i < w1; ++i) {
    if (((double) rand()) / RAND_MAX < idle_chance) {
      --i;
      f << idle;
    } else {
      f << "0011011000100000101";
      r = (unsigned int) rand();
      for (int j = 23; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "1";
      r = (unsigned int) rand();
      for (int j = 31; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "\n";
    }
  }
  /* Reads on GS1 */
  for (int i = 0; i < r1; ++i) {
    if (((double) rand()) / RAND_MAX < idle_chance) {
      --i;
      f << idle;
    } else {
      f << "0101011000100000101";
      r = (unsigned int) rand();
      for (int j = 23; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "1";
      r = (unsigned int) rand();
      for (int j = 31; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "\n";
    }
  }
  /* Writes on GS2 */
  for (int i = 0; i < w2; ++i) {
    if (((double) rand()) / RAND_MAX < idle_chance) {
      --i;
      f << idle;
    } else {
      f << "0011101000100000101";
      r = (unsigned int) rand();
      for (int j = 23; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "1";
      r = (unsigned int) rand();
      for (int j = 31; j >= 0; --j) {
        f << ((r & (1 << j)) >> j);
      }
      f << "\n";
    }
  }
  f.close();
}
```