Network-on-Chip Assembler Language

(Version 0.1)

Zhonghai Lu and Axel Jantsch

June 4, 2003

TRITA-IMIT-LECS R 03:02
ISSN 1651-4661
ISRN KTH/IMIT/LECS/R-03/02–SE

Laboratory of Electronics and Computer Systems
Department of Microelectronics and Information Technology
Royal Institute of Technology
Stockholm, Sweden.

Samsung Ex. 1016

Abstract

Network-on-chip (NoC) is deemed to be a paradigm to tackle design challenges in the billion transistor era. A NoC provides a reusable platform for integrating heterogeneous resources. This report discusses application design on NoC. We propose Network-on-Chip Assembler Language (NoC-AL) that serves an interface between NoC implementations and applications, very similar to the instruction set of a traditional CPU. A central part of NoC-AL is communication primitives. Every instance of a NoC must come with a NoC assembler, which translates NoC-AL programs into a set of NoC configuration files which are in turn handled by standard tools for hardware and software design. In this report, we motivate our NoC application design approach, and discuss NoC communications, in particular, channel communication. Moreover we define two sets of basic communication primitives for the two interprocess communication styles, *message passing* and *shared memory*. Furthermore, we discuss language binding and layers for implementing the NoC communication primitives.

Keywords: Network-on-Chip (NoC), Communication Primitive, Channel Communication, Design Methodology, System Design

Samsung Ex. 1016

Contents

1 Background **2**
1.1 Introduction . . . 2
1.1.1 The NoC Platform . . . 2
1.1.2 The NoC Application Design Tasks . . . 3
1.2 The NoC Assembler Language . . . 4
1.2.1 The NoC Application Design and Compilation Flow . . . 4
1.2.2 NoC-AL Communications in the OSI Seven-Layer Model . . . 5
1.3 Related Work . . . 6

2 NoC Communications **8**
2.1 Communications on NoC . . . 8
2.1.1 Types of Communications on NoC . . . 8
2.1.2 Communication Styles . . . 8
2.2 Communication Issues . . . 9
2.2.1 Naming and Addressing Issue . . . 9
2.2.2 Connection Issue . . . 10
2.2.3 Synchronization Issue . . . 11
2.3 Communication Channel Characteristics . . . 15

3 NoC-AL Communication Primitives **18**
3.1 Message Passing Primitives . . . 18
3.2 Shared Memory Primitives . . . 21
3.3 An Example of NoC-AL Program . . . 23

4 NoC-AL Implementation Issues **25**
4.1 Language Binding . . . 25
4.1.1 Data Type Mappings . . . 25
4.1.2 Expressions of Primitives . . . 26
4.2 Layered Implementation . . . 30
4.2.1 A Standard Interface . . . 30
4.2.2 Implementation of Primitives in the OSI Layers . . . 31
4.2.3 Implementation Layers . . . 31
4.2.4 Channel Features and their Required Actions . . . 33

5 Summary and Future Work **38**
5.1 Summary . . . 38
5.2 Future Work . . . 38

Samsung Ex. 1016

List of Figures

1.1 A NoC of Mesh Structure with 9 Nodes . . . 3
1.2 NoC Application Design Tasks . . . 4
1.3 The NoC Application Design and Compilation Flow . . . 6
1.4 NoC-AL Communications in OSI's Seven-Layer Model . . . 6

2.1 Addressing in the 2D Mesh . . . 10
2.2 Processes Share Memories . . . 12
2.3 Shared Memory as an API . . . 12
2.4 Shared Memory Synchronization Mechanisms . . . 13
2.5 Processors Share Memories: Dance-hall and Distributed Memory . . . 14
2.6 Message Passing Scenario from Source to Destination . . . 15
2.7 A Task Graph with Three Channels . . . 15
2.8 The Channel Reliability Constellation . . . 16
2.9 Reliable Session and Transport . . . 17
2.10 Unreliable Session but Reliable Transport . . . 17
2.11 Reliable Session but Unreliable Transport . . . 17
2.12 Unreliable Session and Transport . . . 17

3.1 Message Passing Procedure Between Processes . . . 18
3.2 The Shared Memory Procedure . . . 21
3.3 An Implementation Scheme of The Atomic Read-Modify-Write . . . 23
3.4 An Example of NoC Application in Task Graph . . . 24

4.1 A Standard Protocol Enables IP Reuse . . . 30
4.2 Software Implementation of Primitives . . . 31
4.3 A Communication Channel Connecting An Initiator and A Target . . . 32
4.4 The Stack of Communication Layers . . . 32
4.5 Latency Check . . . 34
4.6 Bandwidth Check . . . 34
4.7 Negotiation for Latency and Bandwidth During Channel Setup . . . 35

Samsung Ex. 1016

List of Tables

4.1 Data Type Mappings 26
4.2 Comparisons on Reliability Levels 36
4.3 Reliability Levels and Their Implications 37

Samsung Ex. 1016

Chapter 1

Background

This chapter first presents a context for Network-on-Chip design. Based on discussions on the design challenges of a future System-on-Chip (SoC), we point out that a NoC will be a system platform for integrating perhaps arbitrarily heterogeneous resources through communication interfaces. As an instance, we introduce a mesh-structured NoC proposed by KTH. We also identify the design tasks for NoC application designers. Next, we give a definition on NoC Assembler Language which targets NoC application design. It highlights NoC communications by means of communication primitives that nicely fits into the session layer of the OSI seven-layer model.

1.1 Introduction

1.1.1 The NoC Platform

Following Moore's law which has sustained in semiconductor industry for over 35 years, a single chip is predicted to be able to integrate four billion 50-nm transistors operating below one volt and running at 10 GHz by the end of the decade [1]. Due to its huge capacity the billion-transistor chip can take on very complex functionalities with interconnected hundreds of microprocessor-sized computing resources. Such resources may be programmable like CPUs, dedicated like ASICs, configurable like FPGAs, or passive like memories etc.. However, to fully exploit the capacity offered by the technology is facing challenges. At physical level, the interferences caused by crosstalk, power/ground plane noises etc. affect signal integrity to a larger extent. Inductance has to be taken into consideration more closely. A distributed RLC wire model is necessary to replace a lumped RLC wire model. This makes more difficulty to have physical properties under control. At logic level, the wire delay will soon dominate the gate delay. Lowering voltage to reduce power can not be furthered due to the limited space for thresholding a CMOS. If we move upwards, the difficulties do not decrease. On the contrary, they become even worse. At RT level, purely synchronous design approach is challenged because a global clock simply will be infeasible. The synchronous design has to be constrained in a small portion of a future chip. At system level, bus-based systems such as multiple-bridged buses, face even worse scalability problems. Limited bandwidth and bus length will make it difficult to interconnect many more resources. Also, at this level, the heterogeneous resources imply various interfaces or protocols, operating systems etc., which are hard to integrate on a single chip.

If we look from a design methodology angle, the problem looms ahead. The gap between the methodology capacity and the chip capacity is not decreasing but increasing. The ad hoc RTL design considers too many implementation details. Its design capacity (in terms of the number of transistors) and productivity (in terms of design time) will not be sufficient to design the billion transistor chip. To close the gap, the abstraction level of design has to be increased, and reuses at all levels of abstraction are a must. The design challenges of such a complex system on chip (SoC) spark new thoughts or ideas. *Platform-based design* [2] is now a hot topic in academia and industry. It has been very successful among PC-makers. Now the concept is moved into chip area. Although there is no commonly accepted definition for *platform-based de-*

Samsung Ex. 1016

sign, the basic idea is to enable architecture reuse besides IP reuse in order to shorten time-to-market, also simplify verification. It is neither a top-bottom nor a bottom-up approach. It is a meet-in-the-middle approach. There are trade offs between flexibility or generality and performance to come up with a platform. Various classes of applications demand various platforms. For any system platform, the programming model is essential.

Network-on-Chip is perfectly suited for such a platform concept which integrates any type of resources via communication interfaces. One NoC platform [3] proposed by KTH is a mesh structure composed of switches with each switch connecting to a resource, as shown in figure 1.1. The resources are placed on the slots formed by the switches. The switch network offer communication for resources. The resources perform their own computational functionalities and provide Resource-Network-Interfaces(RNI). The maximal resource area is defined by the maximal synchronous area of a technology. The reuse of system architecture to enable fast IP integration is one major advantage expected from the NoC platform. Meanwhile a mesh structure has well-controlled electrical properties which can largely alleviate the interconnection difficulties.

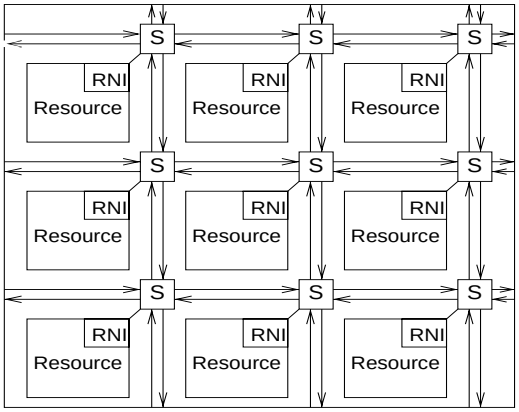


Figure 1.1: A NoC of Mesh Structure with 9 Nodes

1.1.2 The NoC Application Design Tasks

A NoC is inherently a heterogeneous distributed system. The resources are heterogeneous. They can be programmable (FPGAs), dedicated hardware (ASICs), various processor cores, memories, and IP blocks. Heterogeneity implies that different elements, like resources, switches and interfaces, are designed in various means. A number of languages, synthesis tools, software compilers and linkers exist for the design of individual elements. However, there is no single design flow which can be applied to the design of all these elements. The resources are distributed. Distribution implies that processes on different resources interact with each other via the on-chip communication network. NoC design will be communication-centric. In addition to the NoC architecture, we also have to address the design of process communications. If we treat NoC as individual elements, the design of NoC communications may be very complicated. Many communication standards and protocols are desired to coordinate programming NoC communications. The relatively independent design of processes makes it hard to integrate all these elements. For example, how to program a process running on an ARM microprocessor to communicate with a process running on an ASIC through the communication network? The question is not if we can do that. The question is portability or reusability, and productivity. One can image how much application design work will be repeated if there is no operating system to hide the details of the instruction set platform. Here the principle is the same. With a relatively static interface we can hide the dynamics of the lower layer details. Of course, one requirement is that the interface should give the designer enough control over the underlying resources. In this way, the design complexity is decoupled from individual cases. Essentially NoC is treated as a whole unit. Application designers only see the interface without knowing the implementation details of communication.

Samsung Ex. 1016

To facilitate the following discussions, we naturally partition a NoC into three parts: a *network backbone*, *computational resources*, and *communication interfaces*. Here we roughly give work definitions to some terms used in this document in order to avoid confusions. A *resource* is a processing element of any type. A *NoC backbone*, also called a *network architecture*, includes switching or routing network in whatever topology, and the network interfaces (NIs). A *network interface* (NI) wraps a switch or router to offer a standard interface or protocol to outside world. A *NoC architecture* includes not only a NoC backbone and also resources such as CPUs, DSPs, etc. as well as RNIs. Software platform such as operating systems for CPUs and DSPs is part of the NoC architecture. A *communication interface* called Resource-Network-Interface (RNI) can be either a hardware or software interface. In hardware, a RNI is an intermediate module connecting a resource to network. It may enable a resource to hopefully *plug-and-play* on the NoC backbone provided the RNI *speaks the same protocol* as the NI. In software, a RNI is also called a *communication stub* or *socket* which is software platform dependent. It enables custom software to use the network services provided by the NoC backbone.

A NoC is a system platform including both hardware and software platforms. Since some IP blocks, such as DSPs, CPUs, and memories etc. are reusable cores, we expect them to be pre-fabricated together with the NoC backbone. Thus, a NoC itself is a half-customized prototype. Ideally we expect that resources can simply *plug-and-play* on the NoC backbone. To this end the interface between NI and RNI should be standard. As the initial step for platform-based design, we need to define a NoC platform for a specific application class. Consequently we have an application specification on one hand, and a NoC platform on the other hand. In one word, the application design is to map this system specification onto the NoC platform. Specifically speaking, the application design tasks are aimed at custom hardware like ASIC, FPGA, and custom software for application tasks and custom communication interfaces in hardware and software, as illustrated in figure 1.2, where the bridge is a bus bridge acting as an interpreter between the local bus protocol and the network interface protocol. A NoC design methodology should be able to describe both the NoC *architecture* and the NoC *application*. NoC Assembler Language serves for this purpose.

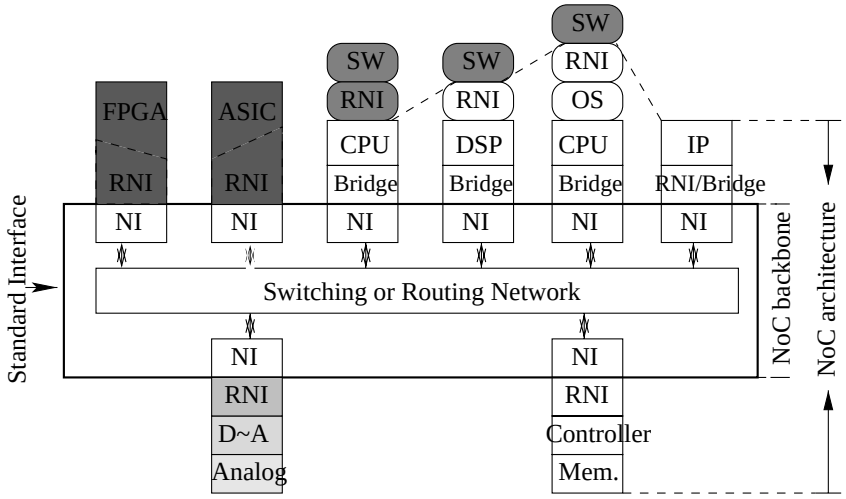


Figure 1.2: NoC Application Design Tasks

1.2 The NoC Assembler Language

1.2.1 The NoC Application Design and Compilation Flow

We define the NoC-AL [4] as follows:

> *NoC Assembler Language (NoC-AL) serves as an interface between NoC implementations and applications, very similar to the instruction set of a traditional CPU. A central part of*

Samsung Ex. 1016

NoC-AL will be communication primitives such as send and receive, open and close, and a standardized way of using shared memory. Every instance of a NoC must come with a NoC assembler, which translates NoC-AL programs into a set of NoC configuration files.

From the definition, we see that NoC-AL treats NoC as a whole instead of individual elements such as resources, switches, and interfaces etc. The design languages for these elements, e.g. VHDL/Verilog for hardware design, C/C++ for software design, SystemC [5] and SpecC [6] for both hardware and software design, are coherent parts of NoC-AL. The NoC-AL offers methods to describe NoC architecture and process communications besides computational tasks. The NoC architecture concerns NoC topology, resource list and process-to-resource mappings. The methods used for describing process communications are *communication primitives* which we will define in this report. Based on architecture and IP reuse, a NoC is a communication platform on which applications run. In application, we separate computation from communication to highlight the communication problems in a NoC. The difference between computation and communication lies in that the former uses only processing elements, while the latter uses both processing elements and communication media [7]. A conceptual illustration of a NoC-AL program is shown below, where the NoC backbone is the 2D mesh. Please notice that some additional statements may be needed to coordinate the computation code and the communication code of a process, but are not shown in the conceptual code. Also, the computation code and the communication code of a process may be actually combined into one file.

```
NoC Architecture Description
 {Topology: mesh 2 x 2
  Resource List:Row1: R1=SHARC DSP, R2=ARM CPU
                Row2: R3=FPGA, R4=ASIC}
NoC Application {
  R1:{P11:{computation_file11.c; communication_file11.c}
      P12:{computation_file12.c; communication_file12.c}...}
  R2:{P21:{computation_file21.cpp; communication_file21.cpp}
      P22:{computation_file22.cpp; communication_file22.cpp}...}
  R3:{P31:{computation_file31.vhdl; communication_file31.vhdl}
      P32:{computation_file32.vhdl; communication_file32.vhdl}...}
  R4:{P41:{computation_file41.verilog; communication_file41.verilog}
      P42:{computation_file42.verilog; communication_file42.verilog}...}
 }             /*Pij stands for process j on resource i.*/
```

A NoC-AL program includes NoC architecture description and application description. A NoC application can be expressed as a *task graph*, and then mapped onto the given NoC architecture after iterations of refinement and hardware/software partitioning until satisfaction. Then we can write NoC-AL programs for custom hardware like ASIC and FPGA, custom software, and communication interfaces in hardware and software. To translate NoC-AL programs into NoC configuration files including both hardware and software parts, we need a *NoC assembler* which does source-to-source processing. Further on, standard tools for hardware synthesis and software compilation & linking are used to generate lower abstraction level codes. This procedure is illustrated in figure 1.3, where the libraries are implementations of NoC primitives, for example, communication primitives.

There are a lot of open NoC-related issues, such as what NoC architectures/topologies are good for which applications, how to efficiently map processes to resources at run time by task migration, and so on. Most of the topics are beyond the scope of this report. In this report we concentrate on the central issue of NoC-AL, the NoC communications, in particular, communication primitives for both *message passing* and *shared memory*.

1.2.2 NoC-AL Communications in the OSI Seven-Layer Model

Network communications are usually expressed using the ISO's OSI seven-layer reference model. In this layered model, the NoC-AL communications handle the session layer, as shown in Figure 1.4. Seen from the session layer, the implementation details from the transport layer down to the physical layer are hidden. The session layer offers a network InterProcess Communication (IPC) service. It should also provide service access points to its upper layer, the presentation layer (if any) or the application layer, and uses services at its lower layer, the transport layer. A session connection provides a relationship between functions

Samsung Ex. 1016

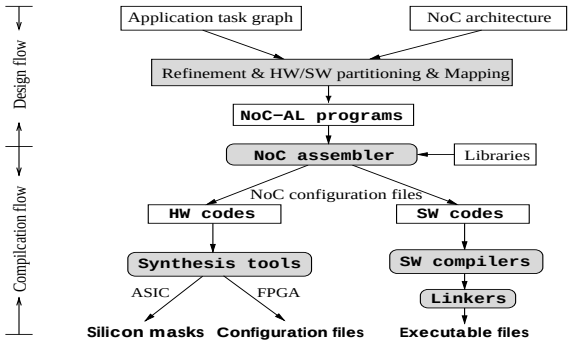


Figure 1.3: The NoC Application Design and Compilation Flow

located in a pair of cooperating systems, established for the purpose of information transfer between them. Information transfer, originated by application processes, is carried through the application and presentation layers to the session layer (ISO 8326, 1990). The services provided by the session layer are concerned with the management of a coherent dialogue between cooperating systems: session establishment and termination, interaction management, synchronization, data transfer and exception reporting [8].

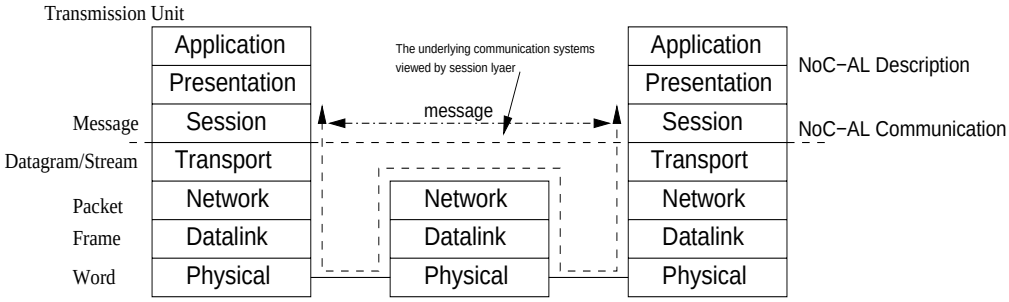


Figure 1.4: NoC-AL Communications in OSI's Seven-Layer Model

1.3 Related Work

While the NoC design process is new, the design of embedded systems and distributed systems has been well-developed. The central part of NoC-AL deals with the inter-process communications (IPC) on NoC. IPC has been extensively researched and used in multitasking operating systems, computer networks and distributed computing [9] [10] [11]. And a lot of network applications such as client-server, producer-consumer, interacting peers paradigms, Remote Procedural Call (RPC) make use of IPC mechanisms. However, the IPC only handles software communications. No matter it works on a distributed or non-distributed system, processes are communicating via software, the underlying of which is always an operating system. However, from the task level, a process can be either a piece of hardware or a piece of software. Hardware/software is the result of a partitioning. NoC-AL communications must deal with processes in both hardware and software forms, resulting in required concerns for chip features, such as power and latency etc.

Within the context of a SystemC model, channels provide means of communication between modules and between processes within a module [5]. SystemC places few restrictions on the functionality of channels. Thus channels may vary widely in complexity - from hardware signals to complex protocols with embedded processes. SystemC defines two types of channels, *primitive channels* and *hierarchical channels*. A primitive channel is one that supports the *request-update* method of access. A few examples of primitive channels are the hardware signal sc_signal, the FIFO channel, sc_fifo, the mutual-exclusion lock, sc_mutex. Primitive channels are limited to the simpler communication mechanisms only. On the

Samsung Ex. 1016

other hand, hierarchical channels, which, as modules that implement one or more interfaces, can have internal processes, offer a more powerful method for modeling complex communication structures, for instance, the on-chip bus (OCB), and also useful for the refinement of primitive channels. Since the SystemC channel supports hierarchical communication and communication refinement, it provides a very good back-end support for implementing the channel we will put forward in the report. We will discuss this further in chapter 4.

One question arises when defining NoC message passing primitives: why not simply adopt Message Passing Interface (MPI) [12]? MPI has become the de facto standard for distributed programming that defines a message passing API library. It comprises 129 functions offering extensive functionality, flexibility, and generality. The implementation of MPI demands the support of powerful operating systems which are very often not the cases for embedded operating systems that are real-time oriented and compact. This may lead to implement MPI on NoC difficult, and less efficient for a specific application. On the other hand, although NoCs use network communication to overcome the scalability problem of bus-based System-on-Chips (SoCs), some communication features, such as transmission latency, bandwidth, and traffic type etc. need to be reserved. However, these features are not available in MPI, but important issues for chip design in order to achieve efficient implementations in terms of speed, area as well as power.

The rest of this report is organized as follows. Chapter 2 discusses NoC communication including communication types and styles, common communication issues and communication channel characteristics. Based on the work in Chapter 2, we define primitives for the two communication styles, *message passing* and *shared memory* in Chapter 3. In Chapter 4, we present some implementation issues of primitives defined in Chapter 3. We summarize this report together with future work in Chapter 5.

Samsung Ex. 1016

Chapter 2

NoC Communications

After classifying the two types of communications on NoC, we briefly look at the two communication styles, *m*essage passing and *s*hared memory. Then we discuss the general communication issues concerning naming and addressing, connection, and synchronization. At last we propose communication channel that carries a set of characteristics imposed by an application.

2.1 Communications on NoC

2.1.1 Types of Communications on NoC

Communications can be classified either physically or logically. From physical point of view, NoC communications can be either intra-resource or inter-resource. *Intra-resource communication* is between processes on the same resource. *Inter-resource communication* is between processes on different resources. From a logical point of view, NoC communications can be either intra-process or inter-process. *Intra-process communication* takes place inside a process, *inter-process communication* (IPC) between processes. Obviously inter-resource communication belongs to distributed interprocess communication since messages are passed through the chip network. Distributed processes are concurrent processes that communicate using the message-passing mechanisms found with IPC facilities.

Although the following discussions are also well suited with intra-resource communication, we focus on inter-resource communication in this report. It is interconnecting heterogeneous resources that makes on-chip communication become a bottleneck for system performance.

2.1.2 Communication Styles

Memory organization plays a decisive role in interprocess communication styles. The memories in NoC can be either *public/shared* or *private/local*. If they are public or shared, the memories are organized as a single global address space. Processes implicitly communicate via *shared variables*. Shared memory can be designed as shared centralized or distributed memory. If the memories are private or local, that means the NoC has multiple address spaces. Processes communicate by *message passing*, i.e. explicitly send and receive messages. In the style of shared memory IPC, concurrent processes share one or more variables and use the changes in state of these variables to communicate. In message passing-based technique, processes send and receive messages explicitly instead of examining the state of a shared variable.

We should mention that the two styles of interprocess communication, *shared memory* and *message passing* are logically equivalent, i.e. given one, you can build an interface that implements the other. However, some programs may be easier to write using one rather the other. In addition, the hardware platform may make one easier to implement or more efficient than the other [13].

Samsung Ex. 1016

2.2 Communication Issues

Communication is concerned with: Who communicates with whom in which language by which media? And how is the communicating procedure conducted? The three elements involved in any communication are communicating entities WHO (sender, receiver), MEDIA (dedicated or shared channel, connection-less or connection oriented channel), and PROTOCOL. The protocol is viewed in two aspects. One is message format, like language and grammar, which enables to correctly interpret data. A typical protocol data unit (PDU) consists of three parts: header, payload (data), and trailer. Obviously all the communicating entities at the same layer must agree on a message format. The other is synchronization regarding how the actual communication is proceeded. Depending on the communication style, the synchronization and its implication vary. For instance, in shared memory style, how to allow concurrent access to a piece of shared code? In message passing style, can both communicating entities perform send operation simultaneously? Or is it only allowed to have that one is sending while the other is receiving, which is similar to that one is talking while the other is listening? A solution for network-based communication has to address the following three major issues [14]:

- Naming and addressing issue – How to designate the communicating entities/parties?
- Connection issue – What type of connection exists between senders and receivers?
- Synchronization issue – How to synchronize between write and read, between send and receive?

2.2.1 Naming and Addressing Issue

To denote the communication parties, a certain type of naming and addressing scheme is used. There are many ways to designate whom you want to communicate with:

- By name (e.g. object X)
- By address (e.g. object at destination X)
- Group identifier (e.g. all objects related to X) used to identify a NoC multi-cast group.

Resource naming and addressing depends on the network architecture and the communication protocol. The principle of naming lies in uniqueness and efficiency. On a single resource the name or process-id of a process is sufficient. If a process name is used, the operating system must resolve the name to a process-id. In a NoC, a (*resource address, process-id*) pair can be used to uniquely identify a process on a resource. We use P_{ij} to represent the process j on the resource i.

The destination resource addressing (routing) concerns routing algorithm, routing mechanism, and routing mode. The routing algorithm determines which of the possible paths from source to destination are used as routes and how the route followed by each particular packet is determined. It restricts the set of possible paths to a smaller set of legal paths. The routing algorithm may be *deterministic*, *adaptive*, *minimal*. One property of any algorithm has to achieve is deadlock free. The routing mechanism selects an output port for each input packet. Usually there are three types of mechanisms: *dimension-order routing*, *source-based routing*, and *table-driven routing*. The routing mode determines how a packet proceeds along its routing path. Typically it includes *store-and-forward*, *cut through*, and *warm-hole*. The three schemes differ in buffer size requirement in switches and packet delivery latency. An interesting extreme case of non-minimal adaptive routing is what is called "hot potato" routing. In this scheme, the switch never buffers packets. If more than one packet is destined for the same output link, the switch sends one toward its destination and temporarily "misroutes" the rest into other output link. The detailed text on routing issues can be found in [15].

In the mesh structure, it is easy to address a resource by making use of the two-dimensional Cartesian coordinate (x, y), so called dimension-order routing. Figure 2.1 shows two possible routes from the source resource $(2, 2)$ to the destination resource $(0, 0)$ in the 2D mesh. Since a resource and a switch are connected in a one-to-one pair, a resource and the connected switch can share the same Cartesian coordinate (x, y). In figure 2.1, they are together viewed as one node.

Samsung Ex. 1016

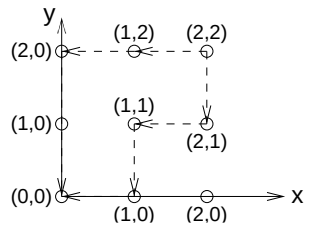


Figure 2.1: Addressing in the 2D Mesh

2.2.2 Connection Issue

If processes are all located inside a resource, connection issue is relatively simple. They are connected by atomic connections such as *signal, FIFO, mutex*, or bus connections. Data flow between processes reliably. Connection issue becomes more critical and intractable when dealing with network message passing. There are two types of connections, *connection-oriented* or *connection-less*. Connection-oriented communication can take place in two kinds of networking schemes, *packet-switching* and *circuit-switching*. Circuit switching is inherently connection-oriented. After a connection setup phase, the channel provides dedicated circuit connection for both sides. It has guaranteed constant data rate service, resulting in *Quality of Service* (QoS). After data transmission, the circuit connection is torn down. A typical example of circuit switching is public telephony. Packet switching is based on a routing layer. It may provide both *connection-oriented virtual circuit* and *connection-less* service. In connection-less service, each message called *datagram* is routed immediately and independently without the knowledge of its previous message. A familiar analogy is a courier service (e.g. postal mail). Each message has to have fully-addressed destination. This incurs much overhead to the message. Although datagram-oriented connection-less service has no setup delay and tear down cost, the packets may be lost due to network congestion or buffer contention, and the data sequence can not be guaranteed after transmission. Also the data may be duplicated. In connection-oriented service, it is called virtual connection because it is achieved in software [16]. It also has a channel setup phase. And a route may be shared among several logical circuits. The routing decision is decided once. Next, the messages called *data streams*, *byte streams*, or simply *streams* can be addressed by a simple virtual channel identifier and follow the same path from the source to destination. It may be delayed at intermediate nodes but the data sequence is guaranteed. In summary, a message passing service can be accomplished by one of the three means, *dedicated connection, connection-oriented virtual circuit* and *datagram-oriented connection-less* message passing. Each of these methods has strengths and weaknesses, and fits different applications.

Basically we distinguish four types of connections:

- Dedicated connection. For instance, circuit-switching connection typically for telephony is dedicated. A channel is a point-to-point hardwired connection between an initiator and a target. It provides constant data rate, and guaranteed services. Data order is preserved. The QoS is high while the interconnect network utilization is lower.

  The other three cases deal with packet switching network where connection implies if a sender is needed to keep the transmission state information, such as which data is being transferred, which have been acknowledged, which can be sent right now, and so on.

- Connection-oriented virtual circuit. An end-to-end protocol, similar to TCP [17, 18]. In virtual circuit services, messages are delivered in order with guaranteed bandwidth. The latency varies within a bounded range. The upper bound is the worst case latency along the virtual circuit path.

- Connection-less. Similar to UDP [17, 18]. In connection-less or best effort service, messages are independently routed in the network. Neither the bandwidth nor the latency can be guaranteed.

- Raw. This opens up a possibility for the session layer to bypass the transport layer and thus directly talk to the network layer [11].

Samsung Ex. 1016

The details of the connection issue are dealt with by the transport layer. For instance, if the connection-oriented service is chosen, the transport layer is required to support the following:

- Connection management: The signaling procedures are required to open, maintain and tear-down connections between communicating entities if connection-oriented service is used. Concrete protocols are needed for the three phases.
- Acknowledgments: An acknowledgment scheme is used by receivers to notify senders of the successful or unsuccessful reception of data. In shared memory communication, it is particularly important to have acknowledgment for the completion of write operations to maintain *correctness* [15].
- Error handling: Data loss due to transmission errors (serious signal integrity problem arising from deep sub-micron implementation) and buffer overflow at the network and the receivers may occur, necessitating error detection and recovery schemes.
- Congestion control and flow control: Flow control involves preventing senders from over-running the capacity of receivers. Congestion control involves preventing too much data from being injected into the network, thereby causing switches or data links to become overloaded. Congestion happens at the intermediate routing nodes during buffer contention. Flow control is an end-to-end issue, while congestion control is concerned with how hosts and networks interact [19].

If the connection-less service is chosen, messages from the session layer are simply encapsulated with the header of the transport layer protocol, and then routed away via the network to destination.

2.2.3 Synchronization Issue

The fundamental problem introduced by concurrent execution of processes is the possibility of interference leading to inconsistent states. The role of synchronization is to avoid the possible histories of a concurrent execution of processes to forbidden states. In other words, to constraint the possible histories to good states. With respect to the two communication styles, there are two classes of synchronization. One is for the shared memory. The other is for the message passing.

Shared Memory Synchronization

Figure 2.5 illustrates processes sharing memory resources. The memories, which can be physically centralized or distributed, are organized as a single address space. Allowing multiple processes to share data structures has to have a *memory consistency model*. This model, which functions as a set of rules, in fact serializes concurrent randomly unordered accesses to memories, i.e. the order of execution of memory accesses from multiple processes. There are several different memory consistency models defined for multiprocessors systems. *Sequential consistency* [20] is the strongest model; it guarantees that memory updates will appear to occur in some sequential order, and that every processor will see the same order. Some consistency models relaxing the ordering constraint of sequential consistency by distinguishing between accesses to synchronization variables and ordinary data. Synchronization operations are dealt with at a high level of consistency, usually sequential consistency. Ordinary operations are processed with a weaker consistency such as processor consistency and release consistency, but the presence of synchronization operations enforces additional ordering restrictions on ordinary operations. Dubois et al. defined weak consistency model [21]. Their model requires (a) access to global synchronizing variables are strongly ordered, i.e. sequentially consistent. (b) no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed. (c) no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed. An ordinary data access is issued either before or after a synchronization operation. Thus this model requires strong consistency of global data accesses with respect to synchronization variables. In another word, all processes must see the normal data access occur in an order with respect to synchronization operation.

The memory consistency models present tradeoffs between ease of programming and implementation overhead. Sequential consistency makes a programmer feel ease because he or she does not need to explicit

Samsung Ex. 1016

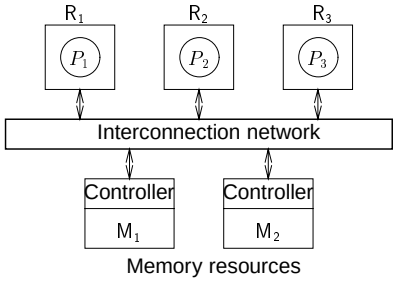


Figure 2.2: Processes Share Memories

y achieve synchronization. However it is costly to implement, and the performance is slower. Consequently multiprocessors typically implement a weaker memory consistency model and leave to programmers the task of inserting memory synchronization instructions. Compilers and libraries often take care of this so the application programmer does not have to do so [22].

A NoC itself can be also a multiprocessor system although it may be heterogeneous. The heterogeneity refers to heterogeneous processor-memory resources on a NoC instead of homogeneous processor-memory resources found in many actually implemented multiprocessors systems, for example, the famous Cosmic cube [23]. This further implies heterogeneous software platforms on the processor-memory resources, such as operating systems and middle-ware libraries. A NoC architecture implements a relaxed memory consistency model. It is the programmers' job to write *synchronized* programs [15] where programs are labeled with synchronization events. To this end, we need a primitive for application programmers to explicitly label synchronizing operations upon synchronization variables.

With memory consistency models, the concept of shared memory is no longer tied to the physical implementation of memory banks. A programmer can write a correct program using the abstractions of concurrent processes and shared memory with little knowledge about the underlying memory implementation that will eventually execute the program. All that the programmer needs to know is the consistency model enforced by memory. This leads to take the shared memory as an application programming interface [24], as shown in figure 2.3. The program and memory agree on a consistency model.

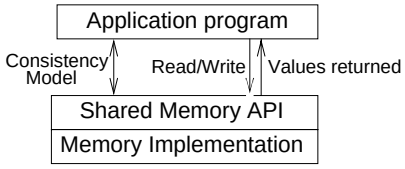


Figure 2.3: Shared Memory as an API

In our NoC memory consistency model, we distinguish synchronization variables from ordinary data. Each critical section is associated with a synchronization variable. The critical section is protected by the entry and exit protocol. Thus we assume that n processes sharing a critical section have the following form:

```
Process Critical_Section [i=1 to n] {
        while (true) {
                entry protocol;
                critical section;
                exit protocol;
                non-critical section;
        }
}
```

The entry and exit protocol work upon the synchronization variable. To implement the synchronization, we need special-purpose messages to the memory, and also specific protocol to handle these messages. In

Samsung Ex. 1016

figure 2.5, synchronization messages are sent from the processes P_1, P_2 and P_3, and handled by the memory controllers in the memories M_1 and M_2 .

Shared memory synchronization can be achieved by mutual exclusion and/or condition synchronization [22], as shown in figure 2.4. The synchronization problem occurs when multiple processes simultaneously try to access the critical section which can only be executed by one process. The critical section is a sequence of execution that accesses the shared objects (variables or resources). Condition synchronization is concerned with delaying an action until the state satisfies a Boolean condition, e.g. a flag is set, a semaphore is up, all processes have reached a barrier. Mutual exclusion is a control mechanism that guarantees only one process can enter, execute and exit the critical section at a time. Entering and exiting the critical section must be atomic, i.e. its execution can not be interleaved with other statements leading to invisible state transformation to the outside processes. Locks and their variants such as spin locks and queue locks are the common mechanisms to provide mutual exclusion. Semaphores provide a low-level but efficient signaling mechanisms for both mutual exclusion and condition synchronization. A semaphore is a special kind of shared variable that is manipulated by two atomic operations, P(s) and V(s). Each is an atomic operation with one argument. Let s be a semaphore. The value of a semaphore is a nonnegative integer. Then the definitions of P(s) and V(s) are as follows:

```
P(s): <await (s>0) s=s-1;>    #wait, down
V(s): <s=s+1;>                #up, signal
```

where the angle brackets $<$ and $>$ specify atomic actions. For example, $<$ `await` (B) S $>$ constructs a general *coarse-grained* atomic action. Boolean expression B specifies a delay condition and S is a sequential statements that is guaranteed to terminate.

The P operation decrements the value of s, but to be sure that s is never negative, the P operation waits until s is positive. The V atomically increments the value of s. The main disadvantage of semaphores is that it is a low-level mechanism and thus difficult to use. Monitors and condition variables are two high-level synchronization mechanisms combined together in order to achieve both mutual exclusion and condition synchronization. Monitors provide implicit mutual exclusion and condition variables provide explicit condition synchronization with blocking semantics. A monitor is an abstract data type that defines monitor variables and atomic operations on them. Collective processes may use barriers for synchronization. A barrier is synchronization point that all processes must reach before any process is allowed to proceed. The basic implementation semantics for such synchronization schemes are busy waiting and blocking. In busy waiting, a process repeatedly checks a condition until it becomes true. Blocking synchronization requires a mechanism to suspend and resume processes (context switching) and to maintain a queue of delayed processes.

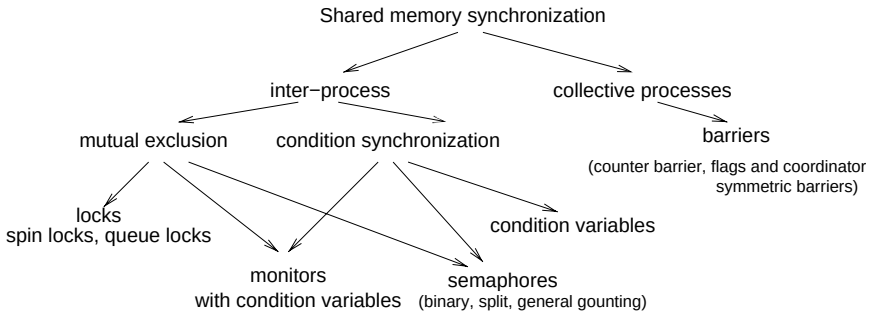


Figure 2.4: Shared Memory Synchronization Mechanisms

Implementing any atomic operations such as barriers, locks, P(s) and V(s) require synchronization primitives that realize an *atomic read-modify-write* operation. For instance, many synchronization primitives have been developed in the form of software procedures, such as test-and-set, compare-and-swap, fetch-and-add, fetch-and-increment, to realize locks, i.e. lock on entering the critical section and unlock on exiting the critical section. These coarse-grained operations are built on some special fine-grained machine instructions that provide atomicity during reading and writing memory. For example, in general-purpose

Samsung Ex. 1016

processors, special load-linked (ll) and store conditional (sc) instructions (e.g. "ll" and "sc" for MIPS4000 or "lwarx" and "stwcs" in MPC860) are implemented in hardware. However, the need for the special load (ll) and store (sc) can be removed if the underlying hardware architecture has some special functional units such as arbiter and hardware locks in addition to memory controller in a multiprocessor SoC [25]. In summary, the implementations of atomic operations require the support of the underlying system architecture. But this does not mean that the microprocessors must offer special machine instructions to support an atomic read-modify-write operation. If there are specific hardware synchronization units handling access to shared memory, the multiprocessors can exclusively access shared memory without using special machine instructions to achieve an atomic read-modify-write instruction.

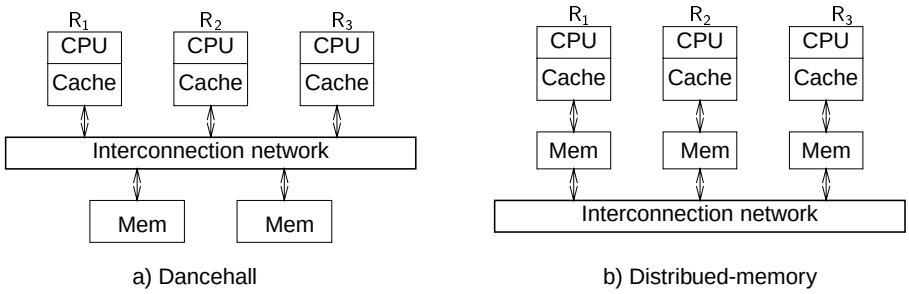


Figure 2.5: Processors Share Memories: Dance-hall and Distributed Memory

To explore temporal and spatial locality, microprocessors usually have local caches to enhance computing performance, as shown in figure 2.5. The memory in figure 2.5(a) is organized symmetrically - all of main memory is uniformly far away from all processors - but its limitation is that all of memory is indeed far away from all processors. Several "hops" or switches in the interconnect must be traversed to reach any memory module from any processor. The approach in figure 2.5(b), which uses distributed memory, is not symmetric. A scalable interconnect is located between processing nodes, but each node has its own local portion of the global main memory to which it has faster access. By exploiting locality in the distribution of data, more cache misses may be satisfied in the local memory and may not have to traverse the network [15]. Shared-memory multiprocessors have an advantage: the simplicity of sharing code and data structures among the processes comprising the parallel application. Process communication, for instance, can be implemented by exchanging information through shared variables. This sharing can result in several copies of a shared block in one or more caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is the cache coherence/consistency problem. Cache coherence schemes tackle the problem of maintaining data consistency in shared-memory multiprocessors. They rely on software, hardware, or a combination of both [26]. Hardware-based protocols include snoopy cache protocols, directory schemes and cache-coherent network architectures. An excellent text on cache coherency can be found in [26, 27].

Message Passing Synchronization

If processes use local memories, the synchronization characteristics of a messaging protocol govern whether a process stops running when it executes a send or receive [14]. In general message passing can be synchronous or asynchronous depending on what synchronization schemes are employed by the send and receive. Basically a send and a receive can be *blocking* or *nonblocking*. A nonblocking operation allows the process to continue execution whereas a blocking operation suspends the process until a certain prespecified condition turns true, e.g. receiving acknowledgment for send, or data available for receive, or timeout. A blocking condition decides when the blocking function call will return. The reliability requirement on communication determines the blocking conditions. The nonblocking operation should allow later to check its status if it has been completed.

Figure 2.6 shows the communication scenario from a source process to a destination process. We assume that there are buffers at both the session layer and the transport layer. At the source, a message is first delivered to the session layer buffer, and then moved to the transport layer buffer. After routing in the

Samsung Ex. 1016

network, the message arrives at the transport layer buffer, and then dispatched to its correspondent session layer buffer at the destination.

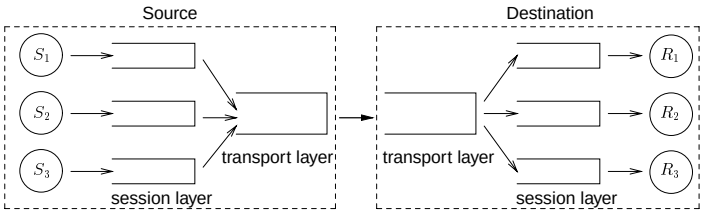


Figure 2.6: Message Passing Scenario from Source to Destination

The blocking send works as follows:

1. If the source process (the application process) asks no acknowledgment, the blocking send returns when the message is delivered to the transport layer successfully. To complete this, first the session layer buffer should be available, and then the transport layer buffer is available.

2. If the source process requires acknowledgment from its destination process, the blocking send returns until receiving acknowledgment.

The nonblocking send returns when the message is successfully delivered to the session layer. The requirement to complete this is that the session layer buffer has to be available.

The blocking receive works as follows:

1. If the source process asks no acknowledgment, the blocking receive returns after receiving a message from its session layer buffer.

2. If the source process asks acknowledgment, the blocking receive returns until receiving a message from its session layer buffer and finishing in sending back an acknowledgment. This implies that there is a session layer buffer which is available for the send back of acknowledgment.

The nonblocking receive returns after checking its session layer receive buffer.

2.3 Communication Channel Characteristics

An application is composed of concurrent communicating processes. It may be represented as a directed graph known as *task graph* in which a node denotes a process, and an arc a communication channel. Figure 2.7 shows a task graph with three channels.

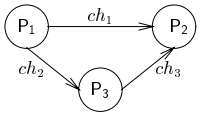


Figure 2.7: A Task Graph with Three Channels

A channel is an *arc* in the task graph connecting a communicating pair. It is an abstraction of communication media, either dedicated or shared. At the task level, it does not incorporate implementation details, such as interfaces, but a set of characteristics regarding performance, cost and Quality-of-Service (QoS) which are required by the application in order to satisfy design goals under given constraints. In the following, we use C syntax and C conventions to facilitate defining communication primitives when necessary. However, they are language-independent, and can be bound to various hardware and software design languages. We will show these bindings in the next chapter.

We identify and define some important channel characteristics in a *struct* as follows:

Samsung Ex. 1016

struct **channel_feature** {*direction, burstiness, latency, bandwidth, quality_class, reliability*}

- **direction:** it gives the orientation of message transfer. It may be simplex, half-duplex, or duplex. For half-duplex, control messages are needed to switch the message transfer direction.
- **burstiness:** it reflects the channel traffic characteristics, which can be periodic or aperiodic. If periodic, it has a burst cycle with maximum burst length. It can model a channel with constant data rate. If aperiodic, it may have minimum burst interval and maximum burst length. For example, keyboard typing is aperiodic, but reasonably it has maximum speed (frequency). This parameter may be useful for modeling chip network traffic, saving power consumption, and allocating bandwidth.
- **latency:** the absolute time of a single unit of data transmitted from the sender to the receiver. It may be also measured in relative time in terms of the number of clock cycles. The latency is one of the requirements from the application. It may have three values: minimum, average, and maximum.
- **bandwidth:** it defines the channel ability of transferring data. It can be measured by different units, for instance, the number of bits per second, the number of frames per second, etc. Different layers have different transfer units. The physical layer transmits words, the datalink layer frames, the network layer packets, the transport layer datagrams or byte streams depending on the connection type, the session layer messages. Message is referred to as the natural communication unit of an algorithm; in general, a message must be broken up into packets to be sent on the network. Among those the size of a frame, packet, and a message is variable within a range. It is infeasible to use the units with variable length to represent bandwidth. A unit with fixed size has to be adopted. We use the number of bytes per second to denote channel bandwidth that is also one of the requirements from the application. It may have three values: minimum, average, and maximum.
- **quality class:** a natural number to represent the quality level of a channel. It is useful for scheduling purpose, for example, when multiple channels are contending for a shared resource, like buffers, communication links etc. It is a quality of service metric at the channel level. The Quality Class (QC) is closely related to the connection states of the channel that are built on the NoC backbone communication services. As discussed previously, the connection can be basically connection-oriented and connection-less.
- **reliability:** In a communication sense, it means that data are sent and received correctly, i.e. no corruption, no loss, no duplication, no out-of-order. Reliability is orthogonal. Any channel can be designed with a certain reliability no matter the underlying layer is reliable or not. The channel directly deals with the session layer whose lower layer is the transport layer. When defining the channel reliability, we have to take into account the reliability of both the transport layer and the session layer. We define four levels of reliability as illustrated in figure 2.8, and describe them as follows:

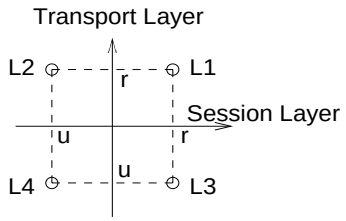


Figure 2.8: The Channel Reliability Constellation

  - Level 1, $S_r T_r$ means that both the session layer and the transport layer are reliable. The session layer at the destination sends back acknowledgment to its counterpart at the source, as shown in figure 2.9. In figure 2.9, the transport layer also sends acknowledgment back. Please note, this scenario only refers to the packet-switching network where the network layer is not reliable. A typical example is TCP built on the Internet. In the case of the circuit switching where the network layer is reliable, the transport layer acknowledgment may not be necessary.

Samsung Ex. 1016

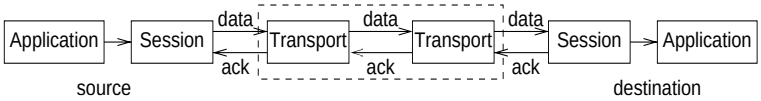


Figure 2.9: Reliable Session and Transport

– Level 2, $S_u T_r$ refers to an unreliable session layer but a reliable transport layer. The session layer at the destination does not send back acknowledgment to its counterpart at the source, as shown in figure 2.10. Similarly to the Level 1, the transport layer does not necessarily send back acknowledgment if the underlying network is circuit switching.

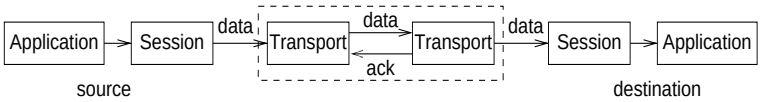


Figure 2.10: Unreliable Session but Reliable Transport

– Level 3, $S_r T_u$ refers to an reliable session layer but an unreliable transport layer. Like the reliability Level 1, the session layer at the destination sends back acknowledgment to its counterpart at the source, as shown in figure 2.11. This actually implies that the session layer has to consider retransmission, thus message labeling and timeout. If a nonblocking sender cares about its messages, it should be prepared to re-send messages if the recipient does not respond after a reasonable amount of time. This means that the sending process should not terminate until it is assured that its messages were indeed received.

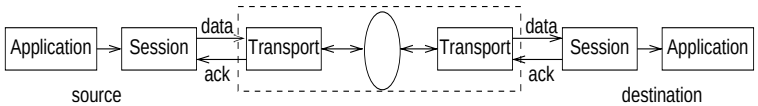


Figure 2.11: Reliable Session but Unreliable Transport

– Level 4, $S_u T_u$ means that both the session layer and the transport layer are not reliable. It is the lowest level of reliability. The transmitted data will neither be acknowledged at the session layer nor the transport layer, as shown in figure 2.12.

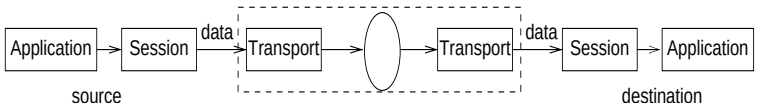


Figure 2.12: Unreliable Session and Transport

Samsung Ex. 1016

Chapter 3

NoC-AL Communication Primitives

This chapter defines primitives for the two communication styles, *message passing* and *shared memory*, followed by a simple application example of using the primitives.

3.1 Message Passing Primitives

A process is uniquely identified by a tuple (*resource_number, process_number*). A channel is shared by a source process and a destination process, thus identified by a (*source_process, destination_process*) pair. We treat a channel as an object, and define a set of methods to operate on it. A message passing procedure is a channel-based data transaction that consists of three phases: *channel setup*, *data transmission*, and *channel tear down*, as illustrated in figure 3.1. A channel is set up by *request* and *response*. The concrete channel setup like three-way handshaking is implemented at the transport layer. For the session layer, it only needs to know what kind of channel the application asks to establish. This handshaking procedure may not actually take place if the initiator asks for a connection-less channel. In this case, opening a channel just assigns the destination address and the channel parameters to the initiator, and does not expect any response. That means, the channel establishment is handled locally. If the initiator asks for a connection-oriented channel, the initiating process sends the setup message to negotiates with the network for bandwidth and delay during the channel setup phase. Once the request is granted, the channel path is fixed, and the bandwidth is reserved. Data transmission may be one-way or two-way. The channel initiator/creator does not necessarily send message first because a channel request may be initiated by the destination who wants specific channel characteristics/parameters. After data transmission phase, the channel can be torn down by either end of the communicating processes.

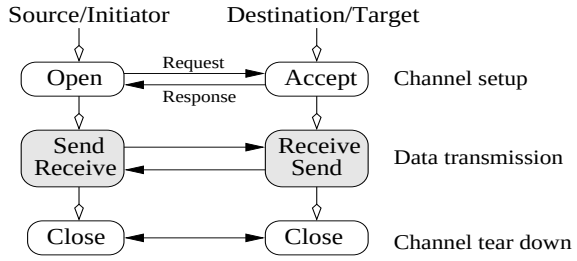


Figure 3.1: Message Passing Procedure Between Processes

In terms of the message passing procedure, we define the following communication primitives for message passing:

- Open a channel

  int **channel**(*source_process, destination_process, channel_feature*)

Samsung Ex. 1016

Description:

- This function initiated by the *source_process* opens a channel between the *source_process* and the *destination_process*. It is carried out by the initiator who wants to contact the other end, the destination. It returns a local *channel* descriptor, which is a nonnegative integer, if successful, or a different negative integer for each of the different reasons of failure, such as the network bandwidth requirement not satisfied, or the destination not available.
- The *source_process* is the process who initiates the channel setup.
- The *destination_process* is the process whom the initiator wants to communicate with.
- The *channel_feature* is defined as a *struct* reflecting a set of channel characteristics as defined in the previous chapter.

- Listen to channel

  int **listen**(*maxQueueLimit*)

  Description:

  - This function sets the maximum size *maxQueueLimit* of channel request queue, and causes internal state changes to permit channel requests.

- Accept a channel

  int **accept**(*channel*)

  Description:

  - This function used by the destination to read one channel request from incoming buffer, stores into *channel*, and responses the channel initiator if necessary. It returns the integer 1 on success, or a different negative integer for each of the different reasons of failure, such as channel request not available, parity check/checksum error, requested channel features not met.
  - The *channel* is the address to put an incoming channel setup request.

- Bind a channel

  int **bind**(*expected_channel, channel*)

  Description:

  - This function checks if an accepted *channel* matches an *expected channel*. It returns 1 for successful matching, -1 for failure.

- Send message

  int **send**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, out-of-band, request*)

  Description:

  - This function sends a message to the specified *channel*. It returns 1 for success, a different negative integer for a different reason of failure, and 0 when the timeout occurs if the send is blocking.
  - The *channel* is the channel descriptor where the messages are sent to.
  - The *msg* is the initial address of the message to be sent.
  - The *msg_size* is the size of the message.
  - The *msg_type* is the datatype of the message. Datatype is one of the basic features of message. Different data types take up different amount of memory. The representations of data types in different microprocessors and design languages may differ. It is necessary to explicitly send the data type of the message in order to avoid misinterpretation and wrong conversion.

Samsung Ex. 1016

- The *msg_id* is the identity number of the outgoing message. It is a natural number used to tag each message. A message can be corrupted, lost, duplicated, and delivered out of sequence during the course of network transmission. To have a reliable transmission, we can use *positive acknowledgment with retransmission*. A message ID enables the sender to do retransmission, and the receiver to maintain correct message sequence and avoid message duplication.
- The *sync_flag* with value 1 or 0 specifies whether the **send** function is blocking or nonblocking.
- The *timer* is used for two purposes. If the **send** is blocking, the timer specifies the maximum amount of time the sender waits for that the blocking condition is violated, e.g., the acknowledgment is received. When timeout occurs, it informs the application by returning 0. Whether to retransmit is up to its application. If the **send** is nonblocking, the timer specifies the minimum amount of waiting time before retransmission, and if the timer equals to -1, no acknowledgment from the receiver is required, thus no automatic retransmission.
- The *out-of-band* is a flag with value 1 or 0 to distinguish *out-of-band* data from *in-band* data. Out-of-band data is considered higher priority than the normal data (sometimes called in-band data). It is useful for conveying control information if something important occurs at one end of the connection and that end wants to inform its peer quickly.
- The *request* is an optional object associated with nonblocking send and receive. It is used later to query the status of the nonblocking communication or wait for its completion.

- Receive message:

  int **receive**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, request*)

  Description:

  - This function is used by a destination process to receive data from the specified *channel*. It returns 1 upon success, 0 when timeout occurs if the receive is blocking, and a different negative integer for some reason of failure.
  - The *msg* is the initial address of the incoming message buffer.
  - The *msg_size* is the size of the message to be taken from the incoming buffer.
  - The *msg_type* is the data type of the incoming message.
  - The *msg_id* is the identity number of the incoming message.
  - The *sync_flag* specifies whether the **receive** function is blocking or nonblocking.
  - The *timer* is used for two purposes. If the **receive** is blocking, the timer specifies the maximum amount of time the receiver waits for that the blocking condition is violated, e.g. a message is available. When timeout occurs, it informs the application by returning 0. Whether to continue polling the channel is up to its application. If the **receive** is nonblocking, the timer specifies the minimum amount of waiting time before re-polling the channel.
  - The *request* is similar to that explained in the **send** function.

- Check nonblocking completion

  int **check**(*request, status*)

  Description:

  - This function checks if the operation identified by *request* completes. It returns information on the operation in *status*, which may be an integer value of 1 or 0 representing completion or not-yet-completion, respectively.

- Close a channel

  int **close**(*channel*)

  Description:

  - This function closes the specified *channel*. It can be initiated by both communicating ends. It returns either 1 for success, or -1 for failure. One possible reason of failure is trying to close a reliable channel before all ongoing messages are received and acknowledged.

Samsung Ex. 1016

In addition to these basic primitives described above, we need some other primitives for channel management such as **getchannelopt**() and **setchannelopt**(), data conversions, multi-cast, and so on.

3.2 Shared Memory Primitives

Shared memories can be used statically like defining global shared variables, or dynamically. Here we are concerned with dynamic use of shared memory. The ways of using shared memories should be standardized. *Shared memory* means data is written into a global memory first by a writer, and then read by a reader. This is in contrast to *message passing* where the sender directly/explicitly passes data to the receiver.

A memory sharing procedure consists of three phases: *memory allocation*, *memory access*, and *memory release*, as shown in figure 3.2. Memory can be written and read in two ways, one-byte-based or multiple-byte-based. That means, data can be written and read one byte at a time, or as a burst of data (multiple bytes) at a time.

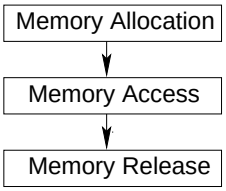


Figure 3.2: The Shared Memory Procedure

- Memory allocation

  int **memory**(*resource, start_address, end_address, memory_type*)

  int **memory**(*resource, number, memory_type*)

  Description:

  - The two functions request a memory segment from the memory *resource*. The first one specifies *start_address* and *end_address*. The second one gives the requested *number* of bytes space. If successful, both return a *memory* descriptor, which is a nonnegative integer. On failure, both return a different negative integer for each of the different reasons of allocation failure, such as memory resource not available, memory full etc.
  - The *resource* is the location of the memory.
  - The *start_address* and *end_address* specifies the range of the requested memory segment.
  - The *number* is the size of the requested memory segment.
  - The *memory_type* is defined as a *struct* concerning if the memory allows multiple concurrent reads:

    struct **memory_type**{read: multiple | single; write: single;}

- Memory access – Read one or multiple bytes

  int **read**(*memory, number, start_address, dataarray*)

  Description:

  - This function reads a bunch of data from the specified *memory* address space. It returns a positive integer denoting the number of bytes being successfully read upon success, and a different negative integer for the different reasons of failure such as read contention, memory address error etc.
  - The *memory* is a memory descriptor.

Samsung Ex. 1016

- The *number* is the number of data bytes to be read out. If the *number* equals to 1, this function reads only one byte.
- The *start_address* specifies the starting memory address for reading.
- The *dataarray* is the pointer where the data is to be written into.

- Memory access – Write one or multiple bytes

  int **write**(*memory, number, start_address, dataarray*)

  Description:

  - This function writes one or multiple data bytes to the specified memory address space. It returns a positive integer denoting the number of bytes being successfully written upon success, and a different negative integer for the different reasons of failure such as write contention, memory address error etc.
  - The *memory* is a memory descriptor.
  - The *number* is the number of bytes of the data array. If the *number* equals to 1, this function writes only one byte.
  - The *start_address* specifies the starting memory address for writing.
  - The *dataarray* is the pointer where the data are to be read out.

- Memory release

  int **free**(*memory*)

  Description:

  - This function releases the allocated *memory* space. It returns 1 for successful release or -1 for failure.

- Atomic read-modify-write

  int **rmw**(*variable, relation, value, operation*)

  Description:

  - This function, which is adapted from the general atomic instruction of the Cedar supercomputer[28], does atomic read-modify-write of the synchronization *variable*. The semantic is

    $<$ await (condition == true) operate on the variable $>$.

    The value this function returns depends on the specific atomic operation it does. For example, if it does Test-and-Set(lock), it returns the variable value before the operation. If it does Test-and-Set(unlock), it returns the variable value after the operation.
  - The two fields *relation* and *value* form one of testable conditions between the variable and the value. In general, the condition takes one of the three formats: variable == value, variable $>$ value, and NULL. The value is a nonnegative integer.
  - The *operation* is applied to the synchronization variable. It has four options: INCrement, DECrement, ADD, SET to 1, and RESET to 0. All of the operations are indivisible.

  With this general atomic function, it is straightforward to derive equivalent primitives for Test-and-Set, Fetch-and-Increment, Fetch-and-Add, as well as the semaphore operations P(s) and V(s).

```
Test-and-Set(lock): int mw(lock,''==''',0,SET)
Test-and-Set(unlock): int mw(lock, NULL, NULL, RESET)
Fetch-and-Increment(s): int mw(s, NULL, NULL, INC)
Fetch-and-Add(s): int mw(s, NULL, value, ADD)
Semaphore wait: <wait until s>0, s=s-1>
        P(s): int mw(s, ''>'',0,DEC)
Semaphore up: <s=s+1>
        V(s): int mw(s, NULL, NULL, ADD)
```


Samsung Ex. 1016

As discussed previously, the implementation of atomic operations needs the support of the architecture. To implement this general atomic conditional operation, we can add some special hardware logic (basically an adder and some control logic) in the memory module [28] to realize the following instruction format: {Variable-Address; (Condition)*; Operation on Variable}. Otherwise, one can use the resource's processor, then a lot of back-and-forth communication between a processor and a memory are incurred. In addition, before doing so, one has to lock the variable first. The communication overhead is very high.

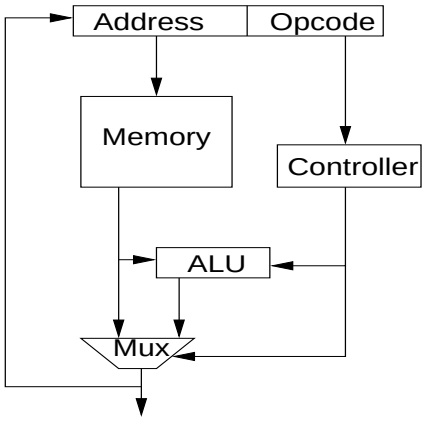


Figure 3.3: An Implementation Scheme of The Atomic Read-Modify-Write

A possible implementation scheme is shown in figure 3.3. The microprocessor who tries to operate on the synchronization variable sends a special message to the shared memory with the variable address and the operational code representing the *condition* and the *operation* in the functional call. The atomicity is achieved at the memory level. The memory has a small control unit (limited number of states) and data path (fixed point arithmetic) to realize conditional operation accordingly. To implement blocking semantics for semaphores, a queue associated with a semaphore may be built additionally. The advantage of this general atomic operation lies in that it does not need any special instruction support from the microprocessor. Thus it is potentially scalable across heterogeneous microprocessors. One limitation of this approach is that the synchronization variable is non-cacheable. This sounds a drawback. But the overhead of implementing cache coherency based on an interconnected on-chip network is so high that researchers are trying with cache-less memory, for example, VTT's ECLIPSE [29].

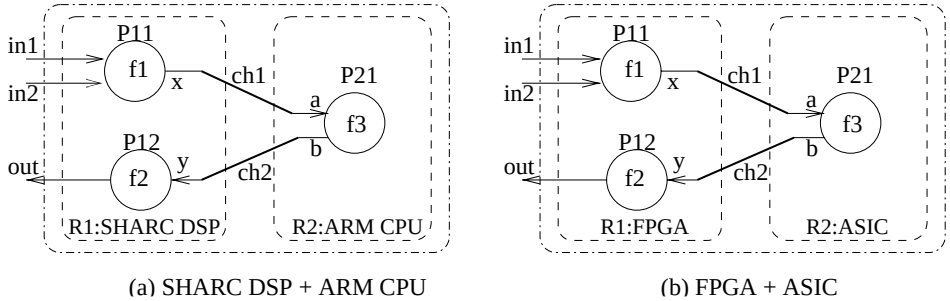
3.3 An Example of NoC-AL Program

Suppose there is a simple application illustrated by the task graph in figure 3.4.(a). Assume we have a NoC which only consists of two resources, a SHARC DSP and an ARM microprocessor.

We manually map the processes P11 and P12 to R1, the SHARC DSP, the process P21 to R2, the ARM CPU. Using the proposed primitives, a NoC-AL program might be coded as follows:

```
NoC Architecture {
  Topology: mesh 1 x 2
  Resource List: Row1: R1=SHARC DSP, R2=ARM CPU}
NoC Application {
R1:{#include <NoC-AL-SHARC.h>
    #include <f1.h>
    #include <f2.h>
    double in1,in2,out,x,y; int ch1=0, ch2=0, ach2=0;
    Process P11 {
     while (1) {
      x=f1(in1,in2);
      while (ch1<=0) {ch1=channel(P11,P21);} //Open channel ch1 until success
```


Samsung Ex. 1016



(a) SHARC DSP + ARM CPU (b) FPGA + ASIC

Figure 3.4: An Example of NoC Application in Task Graph

```
      while (send(ch1,x)!=1) {continue;}}    //Wait for send to ch1 success
     while (close(ch1)!=1) {continue;}}      //Close channel ch1
    Process P12 {
     while (1) {
      while (ach2<=0) {ach2=accept(&ch2);}   //Wait for channel ch2 accepted
      while (receive(ch2,y)!=1) {continue;}  //Wait for receive from ch2 success
      out=f2(y);}}}

R2:{#include <NoC-AL-ARM.h>
    #include <f3.h>
    double a,b; int ch1=0, ch2=0, ach1=0;
    Process P21 {
     while (1){
      while (ach1<=0) {ach1=accept(&ch1);}   //Wait for channel ch1 accepted
      while (receive(ch1,a)!=1) {continue;}  //Wait for receive from ch1 success
      b=f3(a);
      while (ch2<=0) {ch2=channel(P21,P12);} //Open channel ch2 until success
      while (send(ch2,b)!=1) {continue;}}    //Wait for send to ch2 success
     while (close(ch2)!=1) {continue;}}}}    //Close channel ch2
```

Here the **channel**, **accept**, **send**, **receive** and **close** primitives are simplified without additional arguments. These primitives are implemented in software libraries “NoC-AL-SHARC.h” for SHARC DSP, and “NoC-AL-ARM.h” for ARM CPU. Both libraries are used as *include* files.

We should note that the task graph can be also mapped onto hardware resources, or both hardware and software execution resources. If we map the processes to hardware resources, say, FPGA and ASIC, the primitives are used similarly, but implemented in VHDL/Verilog/SystemC libraries depending on which language we are utilizing to describe hardware processes. Figure 3.4.(b) reflects one possibility of the mappings. Accordingly its architecture and application description will be modified. We show its architecture description below:

```
NoC Architecture {
  Topology: mesh 1 x 2
  Resource List: Row1: R1=FPGA, R2=ASIC}
```


Samsung Ex. 1016

Chapter 4

NoC-AL Implementation Issues

In this chapter we address the implementation issues of the NoC-AL communication primitives. We focus on the following two points:

- Language binding: the proposed primitives are language-independent. Although they appeared in a form similar to C syntax and C conventions, they are independent of a specific design language. And those primitives can be bound to a specific hardware/software design language, such as VHDL, C, or SystemC. In this chapter, we take VHDL and C to discuss binding the primitives to hardware and software, respectively.
- Layered implementation: the primitives are high-abstraction level primitives. To be incrementally synthesized, the implementations should have a clear layering from their abstract definitions down to implementations. To this end we adopt the System communication layers.

4.1 Language Binding

There are two issues concerning binding the abstract primitives to target languages. One is data type mapping, the other being expression of primitives.

4.1.1 Data Type Mappings

Data type is fundamental to any design language. The ways to represent data types and the operations on them are various in design languages which adopt various syntax and assume various capacity of the underlying processing elements, such as microprocessors, FPGAs etc. For example, a floating-point variable can be defined as single precision, and double precision in C. In VHDL, it is defined as a **real** type. An integer is defined as **int** in C. In VHDL, an integer may be defined with a constraint range like subtypes, in order to be effectively synthesized.

Due to heterogeneity, NoC resources may work with various data length and data types (16-bit, 32-bit, fixed-point, floating-point etc.), and various representations for the same/similar data types in different design languages. To allow data interoperability among heterogeneous resources, we do the following: (1) use uniform data types, which serve as an abstraction of data types. Each of them has a direct mapping to a data type in the target design language. (2) explicitly send the uniform data type with message. One place to handle the mappings is the NoC assembler.

From a language point of view, there are basically four classes of data types as described in the VHDL 1076 specification:

1. **Scalar types** represent a single numeric value or, in the case of enumerated types, an enumeration value. They are ordered in some way so that relational operations (such as greater than, less than, etc.) can be applied upon them. The standard types that fall into this class are integer, real (floating point) and enumerated types.

Samsung Ex. 1016

NoC *DataType*	C *DataType*	VHDL *DataType*
NoC_CHAR	signed char	string
NoC_INT	signed int	integer
NoC_FLOAT	float	real
NoC_DOUBLE	double	real
NoC_WORD	pointer	bit/std_logic vector
NoC_Array	array	array
NoC_Struct	struct	record

Table 4.1: Data Type Mappings

2. **Composite types** represent a collection of values. Basic composite types are arrays containing elements of the same type and records containing elements of different types. Composite data types offer freedom for users to define custom data types.

3. **Access/Pointer types** provide references to objects/data.

4. **File types** reference objects (typically disk files) that contain a sequence of values.

NoCs as heterogeneous systems should adopt uniform data types, which are used as an intermediary to map into target data types. Table 4.1.1 shows the mappings from some NoC data types to their correspondent VHDL/C data types. This table is by no means complete. A further elaboration is needed to introduce some other useful data types, for example, *packed* or *encrypted* data types.

4.1.2 Expressions of Primitives

The primitives are abstract. They can be bound to any hardware and software design language. To bind the primitives to a particular language, we should follow the syntax of the target language.

At first we express the two data types **channal_feature** and **process** in C and VHDL.

- The **channel_feature** data type

  struct **channel_feature** {*direction, burstiness, latency, bandwidth, quality_class, reliability*}

  – Data type **channel_feature** in C:

```
typedef enum {Periodic, Aperiodic} burstKind;
typedef struct {burstKind kind; int MinCycle; int MaxBurstLength} burstType;

typedef enum {Absolute, Relative} latencyKind;
typedef struct {float min_latency; float avg_latency; float max_latency} latencyValue;
typedef struct {latencyKind kind; latencyValue value} latencyType;

typedef struct {int min_bandwidth; int avg_bandwidth; int max_bandwidth} bandwidthType;
typedef struct
        {int direction, burstType burstiness, latencyType latency,
         bandwidthType bandwidth, int quality_class,
         int reliability} channel_feature;
```

  – Data type **channel_feature** in VHDL:

```
TYPE burstKind IS (Periodic, Aperiodic);
TYPE burstType IS RECORD
        burstKind:kind; MinCycle:integer; MaxBurstLength:integer;
END RECORD;

TYPE latencyKind IS (Absolute, Relative);
TYPE latencyValue IS RECORD
```


Samsung Ex. 1016

```
        min_latency: real; avg_latency: real; max_latency: real;
END RECORD;
TYPE latencyType IS RECORD
        kind:latencyKind; value: latencyValue;
END RECORD;

TYPE bandwidthType IS RECORD
        min_bandwidth: integer; avg_bandwidth: integer; max_bandwidth: integer;
END RECORD;

TYPE channel_feature IS RECORD
      direction: integer; burstiness:burstType;
      latency:latencyType; bandwidth:bandwidthType;
      quality_class:integer; reliability:integer;
END RECORD;
```

- The **process** data type

  A process is notated as P_{ij} denoting the process j on resource i. We define the data type **process** in C and VHDL as follows:

  – Data type **process** in C:

  ```
  typedef struct {int rscNumber; int prsNumber} process;
  ```

  – Data type **process** in VHDL:

  ```
  TYPE process IS RECORD
          rscNumber:integer; prsNumber:integer;
  END RECORD;
  ```

In the following we give expressions of the communication primitives one-by-one in the two languages C and VHDL:

- Open a channel

  int **channel**(*source_process, destination_process, channel_feature*)

  – Function **channel** definition in C:

  ```
  int channel(process source_process, process destination_process,
              channel_feature channel_type);
  ```

  – Function **channel** definition in VHDL:

  ```
  FUNCTION channel(source_process:process;
                   destination_process:process;
                   channel_type:channel_feature) RETURN integer;
  ```

- Listen to channel

  int **listen**(*maxQueueLimit*)

  – Function **listen** definition in C:

  ```
  int listen(int MaxQueueLimit);
  ```

  – Function **listen** definition in VHDL:

  ```
  FUNCTION listen(MaxQueueLimit:integer) RETURN integer;
  ```


Samsung Ex. 1016

- Accept a channel

  int **accept**(*channel*)

  - Function **accept** definition in C:

    ```
    int accept(int *channel);
    ```

  - Function **accept** definition in VHDL:

    ```
    FUNCTION accept(channel:access) RETURN integer;
    ```

- Bind a channel

  int **bind**(*expected_channel, channel*)

  - Function **bind** definition in C:

    ```
    int bind(int expected_channel, int *channel);
    ```

  - Function **bind** definition in VHDL:

    ```
    FUNCTION bind(expected_channel:integer; channel:access)
            RETURN integer;
    ```

- Send message

  int **send**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, out-of-band, request*)

  - Function **send** definition in C:

    ```
    int send(int channel, void *msg, int msg_size, int msg_type,
            int msg_id, int sync_flag, float timer, int out-of-band,
            int *request);
    ```

  - Function **send** definition in VHDL:

    ```
    FUNCTION send(channel:integer; msg:access; msg_size:integer;
                  msg_type:integer; msg_id:integer; sync_flag:bit;
                  timer:real; out-of-band:bit;
                  request:access) RETURN integer;
    ```

- Receive message

  int **receive**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, request*)

  - Function **receive** definition in C:

    ```
    int receive(int channel, void *msg, int msg_size, int msg_type,
                int msg_id, int sync_flag, float timer, int *request);
    ```

  - Function **receive** definition in VHDL:

    ```
    FUNCTION receive(channel:integer; msg:access; msg_size:integer;
                     msg_type:integer; msg_id:integer; sync_flag:bit;
                     timer:real; request:access) RETURN integer;
    ```

- Check nonblocking completion

  int **check**(*request, status*)

  - Function **check** definition in C:

    ```
    int check(int *request, int status);
    ```


Samsung Ex. 1016

– Function **check** definition in VHDL:

```
FUNCTION check(request:access; status:bit) RETURN integer;
```

- Close a channel

  int **close**(*channel*)

  – Function **close** definition in C:

  ```
  int close(int channel);
  ```

  – Function **channel** definition in VHDL:

  ```
  FUNCTION close(channel:integer) RETURN integer;
  ```

The following are expressions of shared memory primitives in languages C and VHDL.

- The **memory_type** data type is defined as:

  struct **memory_type**{read: multiple | single; write: single;}

  Since always only one writer is permitted to access a memory, we only need to define the read mode.

  – Data type **memory_type** in C:

  ```
  typedef enum {MultipleRead, SingleRead} memoryType;
  ```

  – Data type **memory_type** in VHDL:

  ```
  TYPE memoryType IS (MultipleRead, SingleRead);
  ```

- Memory allocation

  int **memory**(*resource, start_address, end_address, memory_type*)

  – Function **memory** definition in C:

  ```
  int memory(int resource, int start_address, int end_address, memoryType memory_type);
  ```

  – Function **memory** definition in VHDL:

  ```
  FUNCTION memory(resource:integer; start_address:integer;
                  end_address:integer; memeory_type:memoryType)
          RETURN integer;
  ```

  int **memory**(*resource, number, memory_type*)

  – Function **memory** definition in C:

  ```
  int memory(int resource, int number, MemoryType memory_type);
  ```

  – Function **memory** definition in VHDL:

  ```
  FUNCTION memory(resource:integer; number:integer;
                  memeory_type:MemoryType) RETURN integer;
  ```

- Memory access – Read one or multiple words

  int **read**(*memory, number, start_address, dataarray, flag*)

  – Function **read** definition in C:

  ```
  int read(int memory, int number, int start_address, void *dataarray, int flag);
  ```


Samsung Ex. 1016

– Function **read** definition in VHDL:

```
FUNCTION read(memory:integer; number:integer;
          start_address:integer; dataarray:access; bit:flag)
        RETURN integer;
```

- Memory access – Write one or multiple words

  int **write**(*memory, number, start_address, dataarray, flag*)

  – Function **write** definition in C:

```
int write(int memory, int number, int start_address, void *dataarray, int flag);
```

  – Function **write** definition in VHDL:

```
FUNCTION write(memory:integer; number:integer;
          start_address:integer; dataarray:access; bit:flag)
        RETURN integer;
```

- Memory release

  int **free**(*memory*)

  – Function **free** definition in C:

```
int free(int memory);
```

  – Function **free** definition in VHDL:

```
FUNCTION free(memory:integer;) RETURN integer;
```

4.2 Layered Implementation

Layering is a powerful means to cope with complexity.

4.2.1 A Standard Interface

To enable IP integration onto a NoC backbone, a standard interface is a cheap solution. In figure 1.2, the interconnection is wrapped by NIs which speak the standard protocol. The resources are wrapped by RNIs speaking also the same protocol. If the resources are pure hardware execution resources, like FPGAs and ASICs, parts of the hardware resources are RNIs. If the resources are software execution resources, like DSPs and CPUs, we assume they have a local memory bus structure. In this case, there is a need of a *bus bridge* interpreting its specific bus protocol and the standard protocol. A good candidate for the standard protocol may be OCP protocol [30] or VCI protocol [31]. The standard protocol acts as an interconnection protocol. All IPs wrapped by this protocol can be simply *plug-and-play* on the interconnection platform, as shown in figure 4.1. For some applications, there may be more than one standard interface.

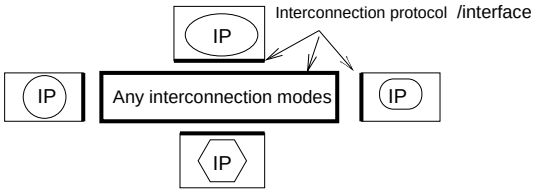


Figure 4.1: A Standard Protocol Enables IP Reuse

Samsung Ex. 1016

4.2.2 Implementation of Primitives in the OSI Layers

For a hardware execution resource, a HW RNI is responsible for implementing the communication primitives. For a software execution resource, a SW RNI, which can also be called *communication stubs*, implements the communication primitives. The SW RNI is built on the operating system, if any. Otherwise, if there is no operating system, the function libraries may contain implementations of the primitives. The device driver may be needed if the local microprocessor system connects to the NoC backbone through an I/O device. The SW RNI is illustrated in figure 4.2, where we assume that the device driver is part of the operating system.

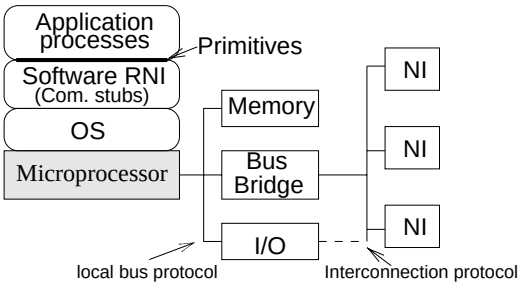


Figure 4.2: Software Implementation of Primitives

In ISO's OSI seven-layer model, a resource implements the higher four layers, i.e. from the application layer down to the transport layer, but not the lower three layers, i.e. from the network layer down to the physical layer. This is in contrast to computer networks where each computer node has a full implementation of the seven layers. In resources, HW/SW RNIs should implement both the session layer and the transport layer. For some custom FPGA/ASIC-type hardware resources, the transport layer RNI may be standardized while the session layer RNI is customized. This separation can make the transport layer relatively stable. In such cases, only the session layer RNI needs to be customized. The same argument may be also suitable for some custom SW RNIs. The NoC switches routing packets from source to destination implement the network layer, the data link layer, and the physical layer. The basic assumption of the packet-switched network communication is *unreliability*. The effects of unreliability usually lead to data loss due to network congestion, data corruption due to interference and coupling etc. at the physical layer, data delivery out of sequence due to routing packets along different routes, and data duplication due to extra retransmission. Each layer should have its own ability to provide services to achieve a certain reliable data delivery, e.g. the data link layer may have error detection and/or error correction. If the network layer doesn't have a reliable transmission scheme, the transport layer is needed to have a reliable scheme, usually by labeling message, acknowledgment and retransmission.

Offering network IPC, the proposed primitives deal with the session layer. Applications directly use the primitives as APIs to program NoC communications. Let us analyze some features at the primitives' layer from an implementation angle:

- It allows abstract data types. The four classes of data types can be directly used.
- It does not incorporate interfaces with both the initiator and the target.
- The control scheme for starting communication is request/response pair which has different implications at various channel requirements.
- Untimed communication behavior. The synchronization scheme for send/receive is either blocking or nonblocking.

4.2.3 Implementation Layers

As mentioned previously, to guarantee interoperability and compatibility among IP cores, a standard interface is required. Here we adopt the OCP protocol which aims to be a hardware interconnect standard for IP

Samsung Ex. 1016

cores and interconnect models to facilitate true plug-and-play methodology [30]. In addition, it has been published a white paper on *SystemC-Based SoC Communication Modeling for the OCP Protocol* [32]. The conceptual framework for modeling communication is based on channel communication. This channel implements the communication between two modules where a module is an initiator or target or both, as illustrated in figure 4.3. It is generic in the following aspects:

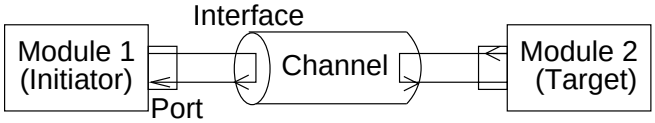


Figure 4.3: A Communication Channel Connecting An Initiator and A Target

- No assumptions on the communication protocol between the two modules are being made. The channel just implements the communication; the protocol must be implemented in the modules.
- The same channel can be used at different abstraction layers.

The white paper defines a four-layer communication abstraction, which allows refinements down to RTL level. Each communication layer consists of three agents: *Initiator*, *Channel* and *Target*. The *Initiator* is connected to the *Target* with the *Channel*, through an *Interface*. The interface presents the target and the initiator with the services the channel offers, thus the channel implementation can be modified without the target and the initiator knowing as long as the minimum services required by the initiator and target are provided. The communication layers support true *interface-based* design methodology. The interoperability of initiators and targets from different communication abstraction layers can be implemented with adapter components such as a layer-wrapper or multi-layer capable channels.

	Abstraction removes:
Message Layer (L−3)	Resource sharing, time
Transaction Layer (L−2)	Clock, protocols
Transfer Layer (L−1)	Wires, registers
RTL Layer (L−0)	Gates, gate/wire delays

Figure 4.4: The Stack of Communication Layers

The layers are defined as follows [32]:

- Layer-3 – Message Layer

  Layer-3 systems are untimed. The system executes event-driven. A single message transmission between initiator and target involves the transfer of several data, which can be of very abstract data types. This layer provides point-to-point initiator-target connections.

- Layer-2 – Transaction Layer

  Layer-2 systems are timed, but not cycle-accurate. The system executes event-driven. A single transaction between initiator and target involves the transfer of several data (i.e. a burst, or a partial burst of data). Normally they are independent of bus protocols since bus protocols can only be implemented with cycle-true systems.

- Layer-1 – Transfer Layer

  Layer-1 systems are characterized by cycle-true behavior. Layer-1 channels provide a fully cycle and protocol accurate connectivity. Most layer-1 functionality can be achieved in RTL. However the

Samsung Ex. 1016

benefits of the Layer-1 over RTL are: simpler netlist, only single wire for the whole communication interface. The netlist does not need to be changed with parameter (communication protocol or functionality) changes; faster simulation as well as simpler interface code.

- Layer-0 – RTL

  RTL layer is pin/bit accurate, register transfer accurate. It is written in final VHDL/Verilog/Synthesizable SystemC.

With layers going down, we are closer to a final implementation. Data types are proceeded from abstract data types, to burst of data, to data until bit vector. Control flow is proceeded from blocking/nonblocking semantics to clocked operations, time is added from untimed system to timed systems. This is actually a refinement procedure from abstraction down to implementation. Clearly the SystemC channel and the communication layers are well-suited for our implementation purposes. We share the same concept of *channel* at a high abstraction level. Thus the SystemC channel can be used for our channel implementation. Also the communication layers allow refinements, thus enable an incremental design principle. In addition to the definitions of the communication layers, in [32] there are library functions proposed with respect to the top three layers. One implication is that we can use the functions to program a SystemC implementation of the communication primitives at each of the communication layers. Hopefully the implementation can be synthesized by industrial tools. Because the library functions are not yet stable today, we can't do this right now. However, we believe a layered implementation approach for communication refinement will play an important role in communication-centric designs in the near future.

4.2.4 Channel Features and their Required Actions

One of important things regarding implementation is the channel characteristics/features. What are the implications of the channel features? What should an implementation do in response to a channel feature requirement? This subsection answers these questions. In the following, we discuss them feature by feature. When speaking of channel setup/tear down phase, we use terms *initiator/source* and *target/destination*. When talking about data transmission, we use the terms *sender* and *receiver*.

1. Direction. It has three options: simplex, full-duplex, and half-duplex. Simplex channel is used in a non-acknowledged producer-consumer paradigm. Data is always sent from a producer to a consumer. Duplex channel works with interacting peers, client-server paradigms. The sender and receiver interact with each other. The third option, half-duplex, is allowed. A sender can send a control message to a receiver to switch the roles of sending and receiving.

   Direction only refers to data transmission phase. During channel setup and close phase, two-way communication may be needed.

2. Burstiness. It may have a great impact on power savings. If transmission is bursty with period cycle and burst length, the intermediate nodes and the receiver may switch to idle states or other lower-power states during no-transmission phase. If transmission is random, it may not be efficient to do so. During the channel establishment phase, the burstiness parameter can be used to direct the receiver and the channel nodes (if there is a virtual or dedicated channel path) enter into power-aware states.

3. Latency. A pair of time values $\{minL, maxL\}$ is set by an application. It may be required to meet deadline constraint of real-time applications. Suppose we are using absolute time values. To check if a channel meets this requirement, the initiator who negotiates with the network needs to calculate the round trip time. The initiator transmits a couple of single unit of data to the target, and waits for acknowledgment. Upon receiving acknowledgment, it calculates the round-trip time. The initiator also starts a timer to specify the maximum waiting time for the acknowledgments. The calculated latency value is simply derived by half of the average of these round-trip time values. If the calculated latency value falls into the range of $\{minL, maxL\}$, the latency requirement is satisfied. Otherwise, it fails. This latency check procedure is illustrated in figure 4.5. The destination should acknowledge the latency checking messages as soon as possible. The latency check is carried out during the establishment phase of a connection-oriented channel.

Samsung Ex. 1016

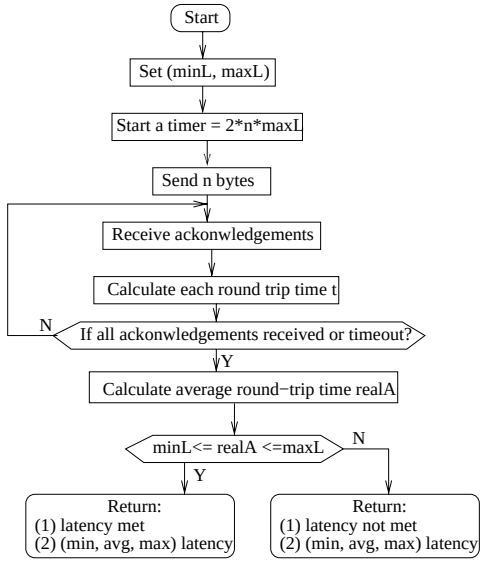


Figure 4.5: Latency Check

4. Bandwidth. A pair of bandwidth values $\{minB, maxB\}$ is set by an application to check if a channel can be established meeting this requirement. After fixing a circuit path, the initiator sends *maxB* bytes to the target within one second, and waits for acknowledgments within two seconds, and then calculate the real allowable bandwidth $realB$ according to the number of acknowledgments. Here we assume that messages are individually acknowledged. If the real bandwidth value $realB$ falls in the range $\{minB, maxB\}$, that means the channel meets this constraint. If the calculated value is lower than $minB$, the initiator will be warned that the channel bandwidth can not be satisfied. The bandwidth check procedure is illustrated in figure 4.6. Similarly, the destination should acknowledge the bandwidth checking messages as soon as possible. The bandwidth can only be reserved with a connection-oriented channel during channel establishment phase.

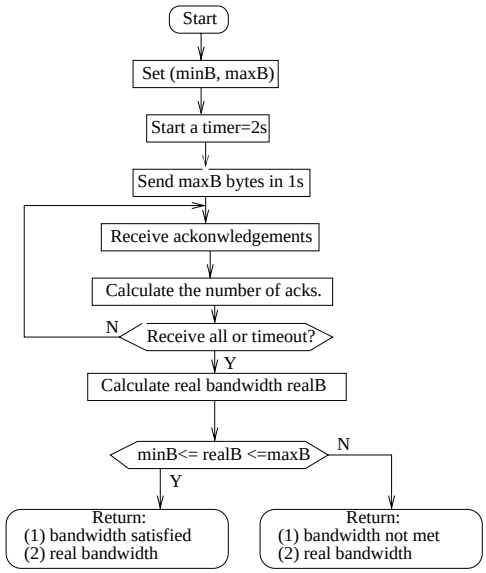


Figure 4.6: Bandwidth Check

From the above description, we see that both the latency and the bandwidth check is taken after a

Samsung Ex. 1016

virtual circuit is granted, i.e., (1) Fixing a virtual circuit path by sending the setup message using the best-effort service, and then waiting for acknowledgment. (2) Check the latency and the bandwidth along this circuit path by the procedures in figure 4.5 and figure 4.6, respectively. This two steps may be iterated until accepting the results, as shown in figure 4.7. This negotiation process is conducted in the session layer.

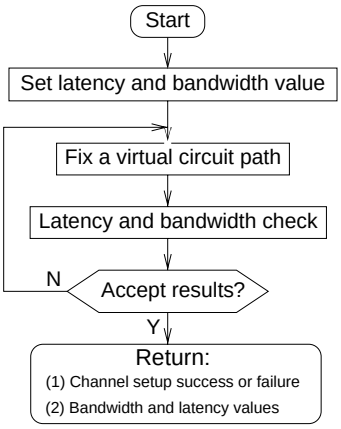


Figure 4.7: Negotiation for Latency and Bandwidth During Channel Setup

5. Quality Class (QC). Reflecting an application requirement, it is defined for scheduling purpose. It is used in three aspects. (1) For best-effort datagram. A higher QC datagram will be routed first or buffered ahead when contending for a communication link, compared with a lower QC datagram. This happens when routing connection-less channel datagrams, system configuration or network management datagrams, and during establishing connection-oriented channels when channel setup messages are routed using the best-effort service. (2) For virtual circuit packets. If channels are overlapped somewhere (part of virtual circuit to be shared), higher QC packets are switched out first. (3) For local resource sharing. One resource may have established multiple connection-oriented channels that talk to the transport layer of the resource. Although all these channels' bandwidth may be guaranteed, the latency may be satisfied with commitment or with *relaxed commitment*. In the latter case, the channel does its best to deliver messages. The upper bound for message delivery is the worst case latency along the virtual circuit path. In a resource with multiple opened connection-oriented channels, messages from a higher QC channel are scheduled first. Clearly the QC is related to Quality of Service, since the difference between a connection-oriented channel and a connection-less channel lies in that the commitment for bandwidth and latency differ. Both the transport layer and the network layer need to be aware of the class levels. We define the following four quality classes basically according to the fulfillment of the channel bandwidth and latency requirement.

   - QC QC_3: Neither bandwidth nor latency asks for guarantee.
   - QC QC_2: Bandwidth needs to be guaranteed, but latency commitment may be relaxed.
   - QC QC_1: Both bandwidth and latency need to be guaranteed.
   - QC QC_0: The highest class is reserved for network management/system reconfiguration or initialization messages. For example, a failure of a switch node makes it necessary to dynamically adapt the routes of the virtual circuits. In such circumstances, the NoC works in a supervising mode. Some of the virtual circuit services, even granted, may be temporarily interrupted.

6. Reliability. We have defined four levels of reliability which put different requirements on the initiator, the channel nodes and the target during channel setup, data transmission as well as channel tear-down phase.

Samsung Ex. 1016

	$L_1: S_rT_r$	$L_2: S_uT_r$	$L_3: S_rT_u$	$L_4: S_uT_u$
Connection	con.-oriented dedicated	con.-oriented dedicated	connection-less	connection-less
Session direction	duplex	simplex	duplex	simplex
Msg. acknowledgment	yes	no	yes	no
Msg. labeling	no	no	labeled	no
Msg. sequence	maintained	maintained	maintained	no promise
Msg. Retransmission	no	no	done by the session layer	no (up to the application)
Session sender	no copy	no copy	keep copy	no copy
Intermediate nodes	be aware	be aware	not aware	not aware
Session receiver	ack.	no ack.	ack.	no ack.

Table 4.2: Comparisons on Reliability Levels

- $L_1 : S_rT_r$. It is built on the connection-oriented packet switching network or dedicated connection circuit switching network. The source application process sends messages to the session layer, and waits for acknowledgments from the destination application process. Since the lower layer, the transport layer is reliable, the session layer does not need to retransmit message. In other words, the transport layer promises to deliver message correctly. If necessary, the transport layer has to take care of retransmission, thus labeling, assembling and time-out. The intermediate nodes at the network must be aware of the channel messages delivered as connection-oriented, since they have to follow the same routing path.
- $L_2 : S_uT_r$. The transport layer is reliable. But the destination application process does not send back acknowledgments.
- $L_3 : S_rT_u$. The transport layer is not reliable. The data integrity and sequence are not guaranteed. The transport layer does not maintain packet sequence. However, the messages must be delivered correctly to the destination in order. The session layer has to take care of retransmission. The destination is required to send back acknowledgments. Achieving this level of reliability results in a costly session layer because the unreliability of the transport layer is compensated at the session layer.
- $L_4 : S_uT_u$. The source application process sends messages to the session layer which in turn handles over to the transport layer. It is a purely simplex transfer of messages. And the destination application process will not send back acknowledgments.

Connection and reliability are closely tied to each other. The four levels of reliability are based on the basic two type of connections: connected-oriented and connection-less. If the interconnect network is circuit-switched, it provides reliable service. If the interconnect network is packet-switched, it offers either reliable or unreliable service. We compare the four levels of reliability in table 4.2.

Different levels of reliability have different implications on the two basic sending and receiving schemes, i.e. blocking send and nonblocking send, blocking receive and nonblocking receive. We assume that the session layer (SL) and the transport layer (TL) have their own send and receive buffers. Table 4.3 shows when the function calls for the blocking and nonblocking send/receive will return. If a session layer buffer is available, a message will be delivered to the session layer successfully. If a transport layer buffer is available, a message will be successfully delivered to the transport layer.

It is worth mentioning that communication protocols are agreements to guarantee the required channel features, not a feature itself. A protocol is a set of control and data structures understood by the communicating entities for synchronizing emission and reception of data and interpreting data. In ISO's OSI seven-layer reference model, a channel deals with the *session layer* offering network interprocess communication services. Implementing a channel with various features requires the support of communication protocols

Samsung Ex. 1016

	$L_1: S_rT_r$ and $L_3: S_rT_u$	$L_2: S_uT_r$ and $L_4: S_uT_u$
Blocking send	SL receives ack.	SL delivers msg to TL
Nonblocking send	Msg is delivered to SL	Msg is delivered to SL
Blocking receive	SL receives msg. and sends back ack.	SL receives msg
Nonblocking receive	SL checks its receive buffer	SL checks its receive buffer

Table 4.3: Reliability Levels and Their Implications

at the lower layers. We have mentioned that SystemC channels support hierarchical communication and communication refinement [5], thus it provides a good reference for NoC channel implementations. For software implementation, Berkeley *sockets* interface [9] [11] [14] was designed to provide generic access to IPC services implemented by whatever protocols on a particular platform. In this sense, the *sockets* API is also a good reference for implementing NoC channels in software.

Samsung Ex. 1016

Chapter 5

Summary and Future Work

Network-on-Chip is receiving more and more attentions in academia, and perhaps in the industry. It is regarded to be a solution to cope with future complex System-on-Chip challenges. It aims to provide a network backbone to integrate IP resources via communication interfaces. While a lot of research is going on regarding the platform itself, how to design applications on NoC, in particular, interprocess communications, is also a challenge. Due to its heterogeneity and distribution nature, no existing design flow can be directly applied to the NoC application design.

5.1 Summary

In this report, we have defined NoC Assembler Language that targets the NoC application design. It serves as an interface between applications and NoC implementations. Subsequently we put forward a NoC application design and compilation flow, which enables reuses of design languages and tools that are familiar to SoC designers. The central part of NoC-AL is communication primitives, which fit into the session layer in the OSI seven layer model. A NoC-AL program consists of both NoC architecture description and application description. In application, we separate communication from computation. The methods to design computational tasks are handled by one of the design languages such as VHDL/Verilog/SystemC. The methods used for describing communications are communication primitives. To translate NoC-AL programs into NoC configuration files, a NoC assembler is required to do source-to-source processing before standard tools for hardware and software design are used. We advocate the channel communication for NoC IPC. A channel is naturally an arc in the task graph representing an application. Every channel has its own features regarding QoS and performance under design constraints. Moreover, we have proposed a set of basic primitives for the two basic communication styles, *message passing* and *shared memory*. Furthermore, we have discussed two implementation issues of the NoC communication primitives. One is language binding. As the proposed primitives are abstract, they must be bound to a target design language. Second the implementations should be layered which allow incremental refinement to add details step by step. Furthermore we have discussed some channel feature implications upon its implementations.

5.2 Future Work

Just as NoC research is in its infancy, so does the NoC-AL. There are a lot of future work opportunities. There will be a huge amount of work on the implementations of the proposed primitives. Although a complete set of implementations doesn't fit well with a research project, a sample implementation for some of the primitives in hardware, say VHDL, and in software, say C, will be most beneficial for evaluating the feasibility of the primitives, and for exploring NoC communication characteristics from high-abstraction level. Another easier way of evaluating those primitives can be done with a NoC simulator. According to feedbacks from evaluations, some of the primitives may be further elaborated, and more primitives may be added. Also one part of the NoC-AL, NoC architecture description has not been addressed in detail.

Samsung Ex. 1016

Moreover, the development of the NoC assembler can be a long-term goal. Although all the ideas must be tested in practice before a final judgment can be made, we expect that the NoC communication primitives can be used for future NoC application design.

Samsung Ex. 1016

Bibliography

[1] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*. 2001.

[2] K. Keutzer et. al. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12), December 2000.

[3] S. Kumar et. al. A network on chip architecture and design methodology. In *IEEE Computer Society Annual Symposium on VLSI*, 2002.

[4] A. Jantsch. Networks on chip. In *Proceedings of the Conference Radio vetenskap och Kommunication*, Stockholm, June 2002.

[5] T. Grotker et. al. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] D. D. Gajski et. al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[7] T. Yen and W. Wolf. Communication synthesis for distributed embeded systems. In *IEEE International Conference on Computer-Aided Design*, 1995.

[8] Richard Lai and Ajin Jirachiefpattana. *Communication Protocol Specification and Verification*. Kluwer Academic Publishers, 1998.

[9] W. Richard Stevens. *Unix Network Programming, Volume 1 - Networking APIs: Sockets and XTI, second edition*. Prentice Hall, 1998.

[10] W. Richard Stevens. *Unix Network Programming, Volume 2 - Interprocess Communications, second edition*. Prentice Hall, 1999.

[11] Alok K. Sinha. *Network Programming in Windows NT*. Addison-Wesley Publishing Company, 1996.

[12] Message Passing Interface. http://www-unix.mcs.anl.gov/mpi.

[13] Wayne Wolf. *Computers as Components*. Morgan Kaufmann, 2001.

[14] Paul E. Renaud. *Introduction to Client/Server Systems - A Practical Guide for Systems Professionals*. John Wiley & Sons, Inc., 1993.

[15] David E. Culler, Annop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture, A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc, 1999.

[16] Douglas E. Comer. *Computer Networks and Internets with Internet Applications, Third edition*. Prentice Hall, 2001.

[17] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley Professional, 1994.

[18] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison Wesley Professional, 1995.

Samsung Ex. 1016

[19] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach, Second Edition*. Morgan Kaufmann Publishers, Inc, 2000.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[21] M. Dubois, C. Scheurich, and F.A. Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Annual International Symposium on Computer Architecture*, pages 434–442, Stockholm, June 1986.

[22] G. R. Andrews. *Foundations of Multithreaded Parallel and Distributed Programming*. Addison Wesley Longman, Inc., 2000.

[23] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[24] Robert Christian Steinke. *Consistency Model Transitions in shared Memory*. PhD thesis, University of Colorado, 2001.

[25] Bilge E. Saglam and Vincent J. Mooney III. System-on-a-chip processor synchronization support in hardware. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, Munich, Germany, 2001.

[26] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[27] David J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Computing Surveys (CSUR)*, 25(3):303–338, 1993.

[28] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.

[29] Martti Forsell. A scalable high performance computing solution for networks on chips. *IEEE Micro*, 2002.

[30] Open Core Protocol. http://www.ocpip.org.

[31] VSI Alliance. http://www.vsia.org.

[32] T. Haverinen et. al. *SystemC based SoC communication modeling for the OCP protocol*. www.ocpip.org, 2002.

Samsung Ex. 1016