Low-Power NoC for High-Performance SoC Design

SYSTEM-ON-CHIP DESIGN AND TECHNOLOGIES

Series Editor: Farhad Mafie

Low-Power NoC for High-Performance SoC Design
Hoi-Jun Yoo, Kangmin Lee, and Jun Kyoung Kim

Design of Cost-Efficient Network-on-Chip Architectures: The Spidergon STNoC
Miltos D. Grammatikakis, Marcello Coppola, Riccardo Locatelli, Giuseppe Maruccia, and Lorenzo Pieralisi

Low-Power NoC for High-Performance SoC Design

Hoi-Jun Yoo
Kangmin Lee
Jun Kyoung Kim

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-5172-8 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Yoo, Hoi-Jun.
Low-power NoC for high-performance SoC design / Hoi-Jun Yoo, Kangmin Lee, and Jun Kyoung Kim.
p. cm. -- (System-on-chip design and technologies ; No. 1)
Includes bibliographical references and index.
ISBN 978-1-4200-5172-8 (alk. paper)
1. Systems on a chip. I. Lee, Kangmin , 1973- II. Kim, Jun Kyoung. III. Title. IV. Series.

TK7895.E42Y66 2008
621.3815--dc22 2008004885

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

Technologists in four different fields are showing great interest today in **Network on Chip (NoC):** parallel computing researchers, networking researchers, computer-aided design groups, and SoC (System on Chip) designers. Initial and basic research has been performed by parallel computing and networking engineers. CAD researchers have developed the concept and design methods of NoC utilizing previous research in the areas of computing and networking. However, the chip itself is most important, and many high performance chips will be developed by using NoC. In this regard, SoC designers have not had enough information and books to refer to on the NoC so far. In addition, researchers in other areas want to see how their theories are applied to real chip implementation. This book was planned to provide practical know-how and examples about how to use NoC in the design of SoC, explaining the "how to" of NoC design for real SoC to circuit designers.

Of course, there are many excellent books on NoC. However, most of these explain on-chip traffic only according to conceptual topologies or packet protocols. Some books explain the application of off-chip network or computer networks to on-chip, such as OSI-seven-layer protocol. In computer networking, like PC plugging to the Internet, real-time changes in the number of physical nodes are common. But in NoC, the number of nodes are fixed at design time and are never changed in the field of use. In addition, the computer network has many fancy and complicated features that are useless or difficult to use in real chip implementation. The chip implementation has clear limitations in silicon area and power consumption. Incorporation of too many fancy protocols and flexibility onto a silicon chip is impractical in relation to chip area and design time, and it is inefficient to extract the best performance out of the silicon.

SoC design is very complex. CAD tools are heavily used to design such complicated systems. However, most books regarding CAD are dealing with very abstract topics and are difficult for real chip designers to understand. Here, we try to provide a layman's version of such complicated concepts as models of computation and communication-computation partitioning. For that reason, we use UML (Unified Modeling Language) because its graphical notation looks friendly to practical circuit designers, and its formalism is well established in CAD and software engineering. For real chip implementation, the complicated features of network theory should be excused and more practical circuits adopted. Such fancy features as topology, routing, and packet switching of networking theory should be reexamined on the basis of chip implementation. It is our intent to provide guidelines on how to simplify complicated networking theory to design a working chip. Because the chip is designed under a strict time budget and with resource limitations, real-time decisions should be made on sound analysis and logic, although the very best solutions may not be achieved. Examples of these will be introduced with the BONE series.

It is our hope that through this book readers will obtain an essential understanding of NoC and how to apply it to their SoC design. Throughout its preparation our motto has been an old saying, truer than ever: **"Keep it simple and make it work for the given purpose!"**

Authors

Hoi-Jun Yoo graduated from the electronics department of Seoul National University and received M.S. and Ph.D. degrees in electrical engineering from KAIST. He invented 2D Array VCSEL at Bell Communications Research, Red Bank, New Jersey, and as the manager of the DRAM design group at Hyundai Electronics, he designed a fast-1M DRAM and 16M DRAM families, also developing the world's first 256M SDRAM. Currently, he is a full professor in the department of electrical engineering at KAIST and the director of a national research center, SDIA/SIPAC (System Integration and IP Authoring Research Center). From 2003 to 2005, he was full time advisor to the Korean Ministry of Information and Communication for SoC and next generation computing. His current research interests are bio-inspired IC design, network on a chip (NoC), multimedia system on a chip (SoC) design, and high-speed and low-power memory.

He is the author of more than 180 technical papers, the two books *DRAM Design* (in Korean, 1996) and *High Performace DRAM* (in Korean, 1999), and two chapters in *Networks on Chip,* Morgan Kauffman, 2006). He received the 1994 Electronic Industrial Association Award, 1995 Hyundai Development Award, 2000 KAIST Research Award, 2002 Korean Semiconductor Industry Association Award, and the 2007 KAIST Grand Research Award. He is an IEEE Fellow and is currently serving the IEEE ISSCC Executive Committee as a member and the Far East secretary; IEEE VLSI Symposia as an executive committee member; 2007 IEEE Asian Solid State Circuit Conference as the technical program co-chair; and 2008 IEEE Asian Solid State Circuit Conference as program chair.

Kangmin Lee received B.S., M.S., and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2000, 2002, and 2006, respectively. He joined the Wireless Modern Research Team, Telecommunication R&D Center, Telecommunication Network Business, Samsung Electronics Co., Ltd., Suwon, Korea, in 2006. He currently designs and develops 3G and 4G telecommunication baseband modem SoCs such as mobile-WiMAX and 3GPP-LTE for mobile phone and consumer electronics applications as a senior SoC architect engineer. His M.S. work concerning the design and implementation of a 10 Gbps/port shared-bus packet switch with embedded DRAM and Ph.D. thesis is about a low-power NoC implementation for high-performance SoC designs. His research concerns the theory, architecture, and implementation of high-performance and low-power SoC for mobile communication applications.

He has published more than 20 papers in international journals and conferences, and also wrote book chapters in *Network on Chips* (Morgan Kaufmann, 2006). Dr. Lee received the best design award and the silver prize at the 2002 and 2004 National Semiconductor IC Design Contest, respectively. He also won the outstanding design award at the 2005 IEEE Asian Solid-State Circuits Conference (A-SSCC) Design Contest. He is a member of Technical Program Committees of Design, Automa-

tion and Test in Europe (DATE) conference, International Conference on Computer Design (ICCD), and International Conference on Nano-Networks. He is also serving as a reviewer for IEEE Transactions and Journals actively.

Jun Kyoung Kim received M.S. and Ph.D. degrees at Systems Modeling Simulation Laboratory (SMSL) in electrical engineering, and in computer science from KAIST. He is with Semiconductor System Laboratory (SSL), KAIST, as a postdoctoral fellow.

During graduate study, Dr. Kim gained knowledge on systems theory, especially about discrete event modeling/simulation. Based on this, his research topic has moved to design methodologies, covering codesign, interchange format for hardware, retargetable processor description language, and design space representation/exploration. Dr. Kim constructed a design space exploration framework based on attributed AND/OR graph as its representation scheme, along with construction of database system.

His current research topic at SSL is UML-based high-level design methodology, which covers from requirement specification to HDL-level modeling. He is working on high-level design methodology based on UML (Unified Modeling Language), which is a graphical language for specification, modeling, analysis, and implementation.

Part 1

NoC-Based System-Level Design

1 NoC and System-Level Design

1.1 INTRODUCTION TO SOC DESIGN

Recently, the term SoC (System on Chip) has replaced VLSI (very-large-scale integration) or ULSI (ultra-large-scale integration) as the key word in information technology (IT). The change of name is nothing but a reflection of the shift of focus from "chip" to "system" in the IT industry. Before the SoC, semiconductor technology and circuits themselves played the central role as a discipline and industry, and engineers needed to master semiconductor circuits and technology to enhance the performance of components in the known target system. However, in the SoC era, the engineer is required to provide a system solution to the target problem with the final end application in mind. These SoCs are widely used in portable and handheld systems such as cell phones and portable game devices, as shown in Figure 1.1 and 1.2.

You may wonder what the system is and how it is different from the chip design in providing a system solution. There are many definitions according to different viewpoints, but in this book *system* is defined as "a set of components connected together to achieve a goal as a whole for the satisfaction of the user." It is clear that a system has an end user who wants to use it to do a specific job; it is a little bit different from the component semiconductor chip used as a part of the system. A system usually has an embedded user interface as a form of software and encompasses many components inside, not only the hardware but also the software that constitutes the system. Such a complicated entity can be handled only with computer-aided design tools, automatic synthesis of the physical layouts, and sound software engineering knowledge. In addition, the system functions to achieve a specific goal, as a whole, are usually described in algorithms that should satisfy user requirements in time, conveniently.

Therefore, the discipline of SoC design is intrinsically complicated and covers a variety of areas, such as marketing, software, computing system, and semiconductor IC design, as described in Figure 1.3. SoC development requires hexagonal expertise in not only technological areas such as IC technology, CAD, software, and algorithm but also in management techniques—complicated team, project, and customer research. In this chapter, we will take a look at different views on SoC design, overall, and its relation to Network on Chip (NoC) in regard to low power consumption.

The first concept in SoC was just to copy the system implemented in a PCB onto a silicon chip. By adopting the same bus architectures as those used in the PCB, the processing of embedded applications was implemented on a single chip by assembling dedicated hardwired logics and known general-purpose processors. This concept gave birth to the idea of *reuse of the design,* as off-the-shelf standardized IC components were soldered on a PCB. The functional circuit blocks can be predesigned and verified for later integration into the SoC as a design component,

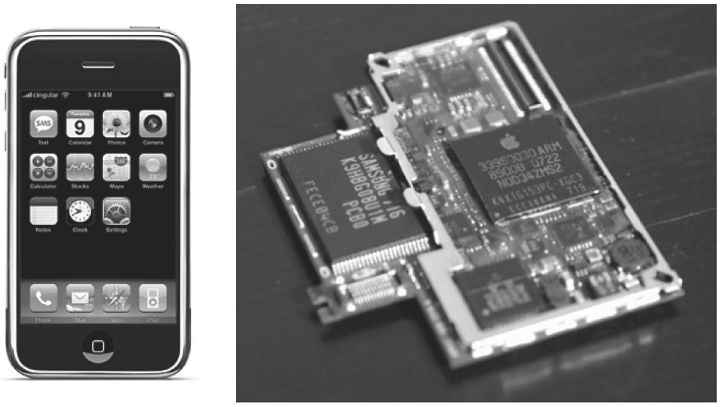


FIGURE 1.1 Apple i-Phone and its system board.

FIGURE 1.2 Sony PSP and its inside boards.

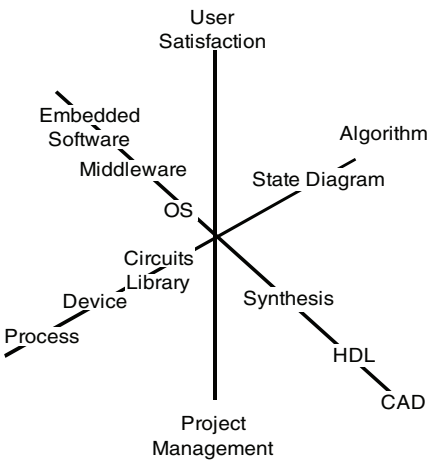
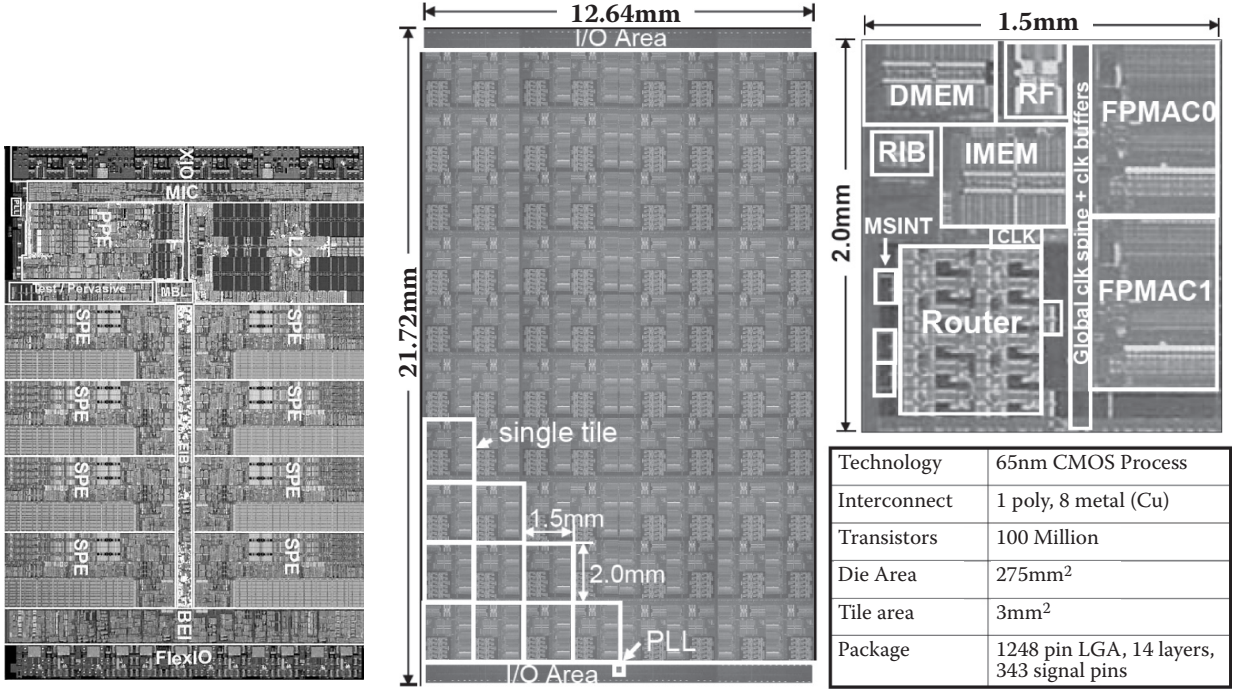


FIGURE 1.3 Disciplines required for the design of SoC.

not a fabricated chip. However, such preverified and reusable designs, called *intellectual property* (IP), were difficult to assemble because they were developed for performance optimization in house, neglecting the general interface matching. Besides, PCB buses were not appropriate for the on-chip environment. However, on-chip bus performance can be further improved by using specific on-chip characteristics such as increased bit width, low power, higher clock frequency, and tailored interface architectures. The mismatch of interfaces can be solved if IP interfaces are fixed or standardized. Figure 1.4 shows examples of large-scale SoCs, which are fashioned by design reuse. Figure 1.4a shows a chip photomicrograph of the



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm^2
Tile area	3mm^2
Package	1248 pin LGA, 14 layers, 343 signal pins

(a) (b) (c)

FIGURE 1.4 Chip photomicrograph of (a) Sony Emotion Engine, (b) Intel 80-core Processor, and (c) the detail of a unit processor.

Sony Emotion Engine [1], which has eight high-performance processors integrated on a chip. Intel's research chip, which has 80 CPUs inside, is shown in Figure 1.4b and the unit CPU, in Figure 1.4c [2].

1.1.1 System Model and Design Flow

System design is the process of implementing the desired application for the end user by integrating a set of physical components, software, and functions. It cannot be realized straightaway, but as design time proceeds, it gets closer to being manufactured, depending on the given constraints. Because of its complexity, it is more practical and convenient to describe and handle the system in an abstract form, ignoring the detailed physical components and functions during the early design phase. A system can be divided into three conceptual parts: a collection of simpler subsystems, the method for the subsystems to be connected together, and the collective behavioral operations of the entire system. The three parts present an analogy with the automobile, as shown in Figure 1.5. The subsystems such as the engine, tires, and wheels should be assembled by "a specific connection method" to make a car. The connection method is the key to making a unique car model with a specific performance. In addition, the hardware, in total, should provide the system "behaviors" for end-user applications such as speed driving, touring and, baggage transporting.

The method for the selection and connection of the subsystems to implement the functions is called the *model* [3]. Further, it can be divided into three submodels, the system model, the function model, and the architecture model, as shown in Figure 1.6. Only the system's behaviors are visible to the end user, and its essential characteristics and architectures will be designed based on those behaviors. Usually, models use a particular language such as C or C++ or, of late, a graphical language such as Unified Modeling Language (UML) to describe the system.

First, the system requirements (e.g., what the SoC wants to realize) should be examined. Then, these are refined into system specifications, in which the performances of the system are specified; however, the implementation details are not

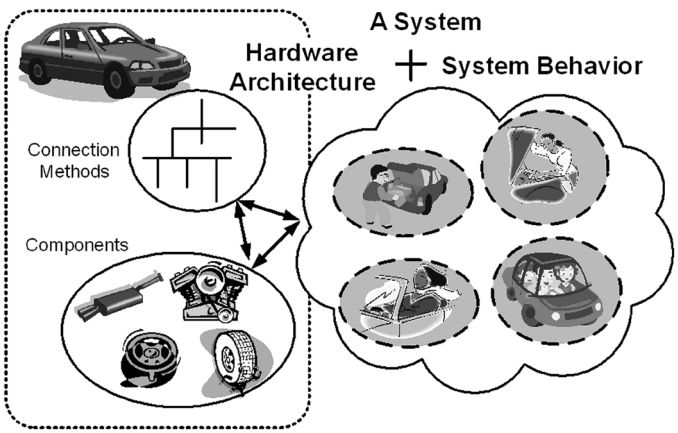


FIGURE 1.5 Concept of system, behavior, and architecture.

determined yet. Because the system solution needs to be implemented on an SoC, the software running on the embedded CPU also should be designed concurrently (see Figure 1.6). This is the major difference between the SoC and VLSI designs. Therefore, the design process involves the determining which part of the specification will be implemented in hardware and which, in software.

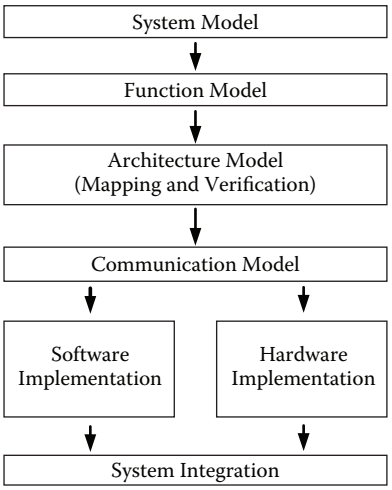


FIGURE 1.6 SoC design flow.

Once the concept of the target system is grasped, the set of functions to realize the system specification should be derived and divided into more affordable unit functions. Therefore, the functional specification of a system is determined as a set of functions, which calculate the outputs from the inputs. In this design stage, UML is frequently used and will be briefly explained in Subsection 1.1.2. The system model is developed using the system's behavioral specification, and it is executable in a high-level language and linked to an executable specification of the system's environment. For example, the CDMA or DVB-H standards can play the role of a virtual test bench of the system and can be used for verification of the system model of the SoC. By integrating behavioral libraries and algorithms, the system behaviors are verified at a pure system behavioral level without any implementation dependency. In the system model, the system is one entity, and its software and hardware parts are not clearly divided. After full verification of the system model with the environmental model (test bench), the model is grouped, divided, and modified for its functionality to be realized as specific hardware and software.

In the *function* or *behavior model*, the candidate functional blocks and software will be defined. It consists of the hardware/software system components and connecting rules, which represent the system's characteristics. Of course, the target functions, such as microprocessor, DSP, memory component, and peripheral and hardware accelerator functions, may be included in this model. The availability of specific components and their manufacturability are not of concern, but the realization of system characteristics does matter. The model should be formal without ambiguity, complete to describe the entire system, visible to engineers and designers, and flexible for modification [3]. Finite state machine (control flow graph), data-flow graph, and program state machine are commonly used as function models to describe in detail how the target system will work. Generally, the function model step does not specify how the system is to be manufactured.

After the functional specification of a system is determined, the architecture exploration stage follows. It searches for the most appropriate way to specify the number and types of components as well as the connections between them. An example is shown in Figure 1.7 where the behavior of a setup box system is mapped

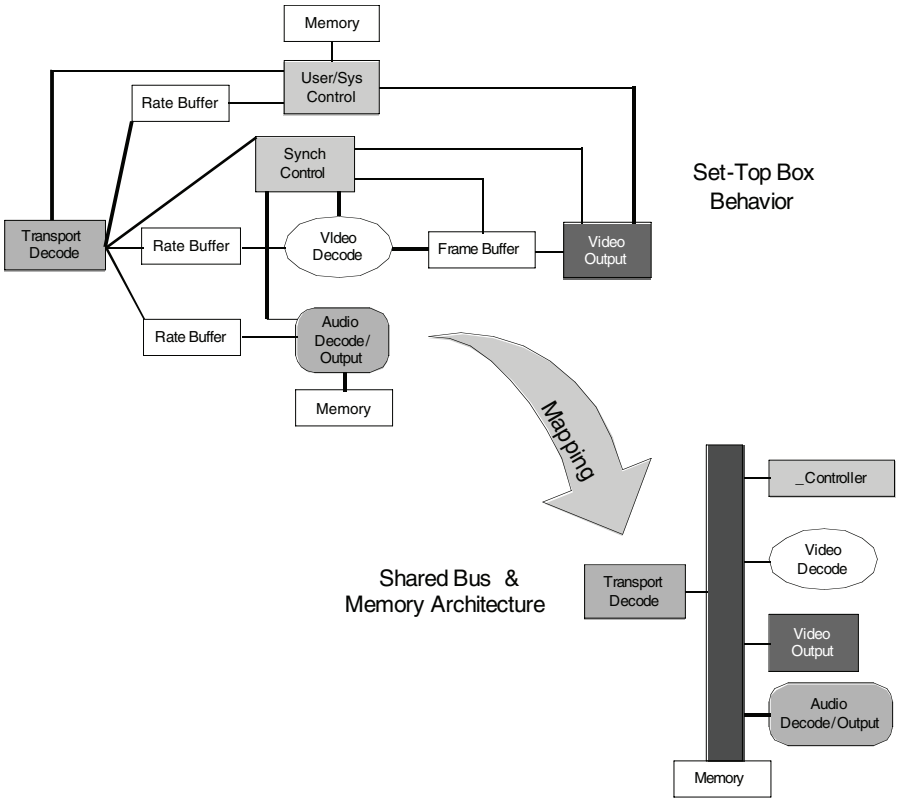


FIGURE 1.7 Mapping of functions to an architecture [4].

to a shared bus-type hardware architecture. In the functional specification, there is no clear separation yet between software and hardware, just mapping or partitioning the functional blocks of the function model onto the architecture model by assigning every function to a specific hardware or software resource. The assigned hardware may be a dedicated hardware block or one mode of a dedicated hardware block, and the assigned software, a task running on a general or specialized processor. In addition, depending on the availability of IPs, some of the hardware parts will be newly designed and some will just use the existing IPs. Based on the performance analysis of the assembled system, it can be decided which part of the target system will be implemented by hardware and which part by software. For the software part, the compiler techniques will be used according to requirements. On the contrary, for the hardware part, because only the functional behavior has been determined, its architectures should be derived from the functions. The function set to describe the behaviors is analyzed and its common parts are selected to describe in the transaction level. That is, the hardware is divided into multiple modules of appropriate size for processing, whereas the behaviors of the individual modules are described by a high-level modeling language. Then, the resulting behavior of the target system is applied to the collective operation and communication among multiple modules. The detailed communication mechanism among multiple modules should be mod-

eled and analyzed independently as if they were separate hardware modules. This will be explained further in Subsection 1.3.2. The hardware modules are converted into register transfer level (RTL) description by high-level synthesis tools. Although the appropriate architecture model can be found by trial-and-error-based mapping in many practical cases, there have been many studies on how to get the optimal mapping automatically under the given system, hardware, and software constraints. Most of the architectures are made of the integration platform composed of communication networks, RISC, DSP, and parallel processors such as VLIW, SIMD, and MIMD.

It is found that system integration is hard to obtain by simple partition and mapping of functions. The design complexity is too high for the subsystems to be assembled in a given time with full verification of their functionality. A subsystem cannot provide full performance to the system due to the complexity of the assembly. Separate optimization and methodology of the connection and communication of the subsystem becomes necessary. As the architecture model is improved as the design process is further iterated, the communication mapping started at a very high level of abstraction is refined to a more detailed level as well. It is common that the communication resources are scarce and should be shared by many blocks. When the communication function is mapped onto a communication resource on a one-to-one basis, there is no contention for the resource. It is natural to apply on-chip network concepts to enhance the scalability and performance of the communication resources. This is why NoC was introduced. Because co-optimization of the communication network and the architectural blocks is difficult and complicated, the maturity of the communication network may be used as a criterion for the progress of the design process. In that sense, NoC is the most advanced technology in SoC design. The communication network itself plays the real integration platform, and once it is specified, the architectural blocks can be plugged to the network for the SoC design. This approach will in the long run dominate SoC design and is the main theme of this book.

1.1.2 System Analysis with UML

The design of a complicated system requires high-level abstraction to reduce its design complexity. External specification is required for supporting the system application, whereas internal specification is necessary for a clear definition of the design scope and hardware implementation. Recently, an object-oriented approach has been applied to the analysis of the system specification to enhance the readability and re-usability of the design. UML has unified the existing object-oriented design methodologies and design documentation methods, and is widely used in software engineering. It was standardized by OMG (Object Management Group) in 1999 as UML 1.1 and, in 2004, as UML 2.0. Here, we would like to introduce UML as a good vehicle to analyze the design of the SoC, too. Although UML 1.1 has nine types of diagrams, only three (use case diagram, class diagram, and sequence diagram) are useful in deriving the behavioral specification from system requirements for the SoC design.

UML can be used in the development of SoC in three different ways. The first is to use UML to get a rough sketch of a system's behavior. In this case, the UML

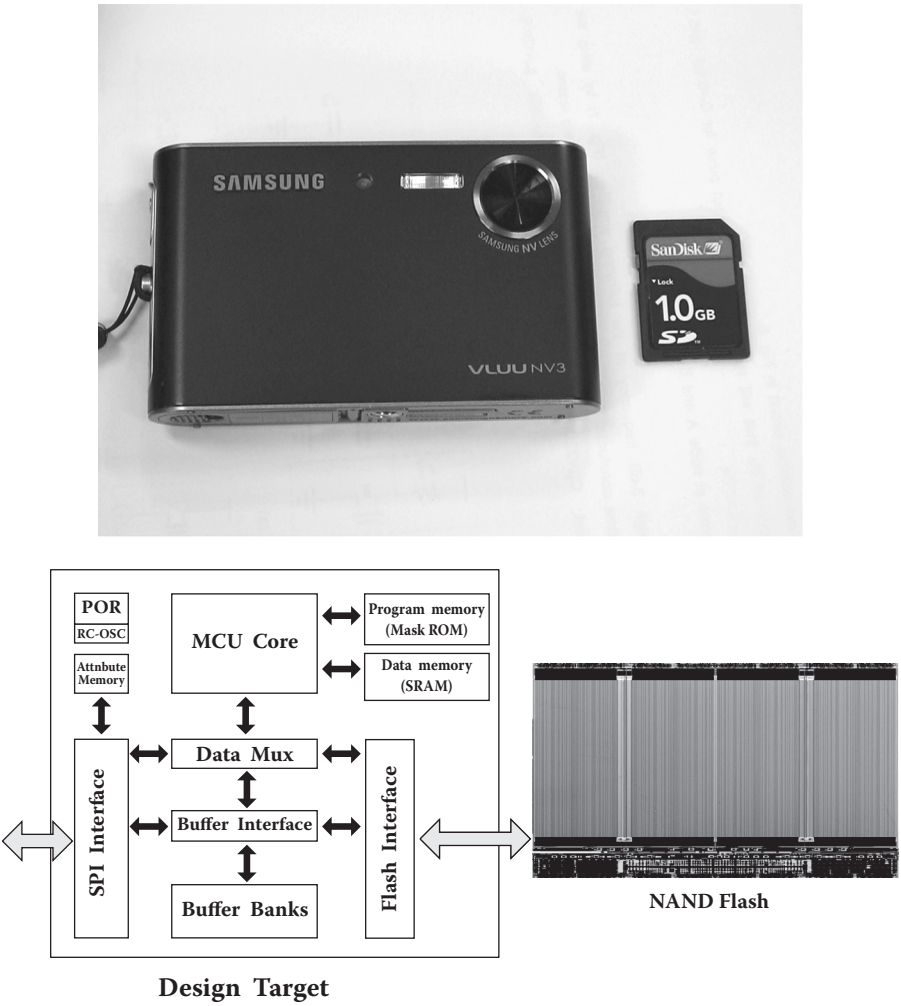


FIGURE 1.8 SD Flash Card and its controller with NAND Flash memory.

diagrams are drawn for a better understanding of the overall system behavior, to provide efficient communication in the design team, and for a clear documentation format. The second approach is to regard UML as a design language and use it actively in the detailed hardware and software design. The third way is to use UML and the source codes in parallel to design a SoC. The UML generates a skeleton model of a target source code C++, and later the designer completes the detailed program by adding the required scripts to the skeleton model. The second and third approaches of using UML for the design of SoC will be explained in Part II. By a combination of the first and second approaches, the system specification analysis using UML will be introduced here, with the Flash memory controller as an example. Of course, there are many text books explaining the UML for software engineering. However, we try to summarize the UML for the SoC design following M. Fujita's approach [5]. More

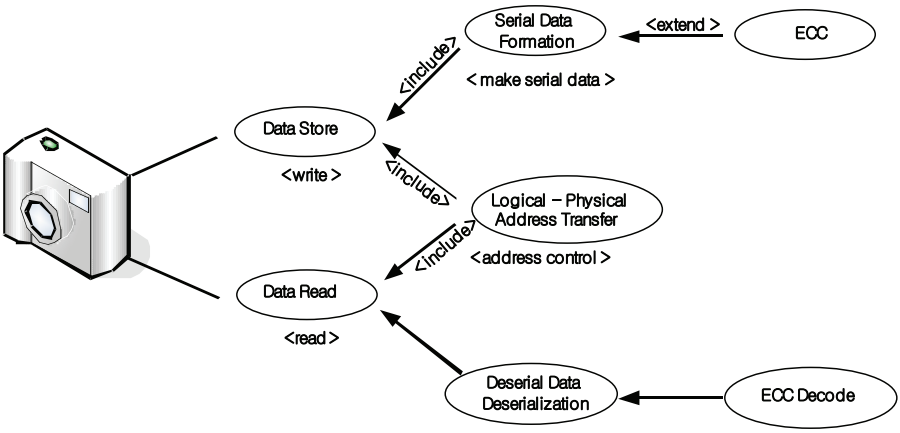


FIGURE 1.9 Use case diagram for SD Flash card.

specifically, SD (secure digital) memory card, shown in Figure 1.8, which is widely used in digital cameras and personal digital assistants (PDAs), will be examined as an example.

SD memory communicates with the host systems such as digital cameras, MP3 players, and cell phones through serial peripheral interface (SPI), and transacts data with the internal NAND Flash chips through Flash memory interface. For the data transfer, the host system with the SPI interface regards SD memory as a kind of writable disk system by transforming the logical address into a Flash memory physical address.

In the conventional hardware design, the designers themselves had to understand detailed specification of the target system, divide it into multiple manageable subsystems, and implement the target system based on their own experience in hardware design. On the contrary, system design with an object-oriented method begins with use case analysis. This is a process to define the system requirements in the form of use case diagrams, starting from basic functions of the system and proceeding to other related functions. In addition, it is possible to make a clear distinction between the system and the system interface, and to provide a detailed description of the external input and output functions. Figure 1.9 shows a use case diagram for the SD memory card. To describe the use of the target system, in the first place, the type of user, such as "actor," who controls the operation of the system should be specified. If a digital camera reads or stores data through SPI, the camera is placed in the left side of the use case diagram as an actor. Its basic functions are to read data and to store data operations, as shown in Figure 1.9. For data storage, parallel data is converted into serial data, and the logical address of the host memory space is transformed into the physical address of the NAND Flash memory. In this example, wear-leveling is adopted to prevent the Flash memory block from wearing out [6]. In addition, comments can be added to explain each use case diagram in more detail.

When the use case diagram is drawn, the following points should be taken care of.

1. System boundary and definition of the "actor": The actor should be outside the system and interact with the system directly.
2. Use case diagram: It should contain in detail what kind of functions will be possible by the actor selected according to item 1. The designer, from the view point of the actor, should use the active form and present tense of verbs or phrasal verbs to describe the required functions. The internal processing in the system should be clearly differentiated from the functions provided by the actor. Then, the designer completes the use case diagram by linking the functions with a use case scenario. That is, the individual should describe what kind of actions are done, and in what sequence.
3. Alternative sequences of the functions: Not only the basic sequence but also possible alternative sequences of the functions should be indicated.

Now, the system is described by the use case diagrams, but it is still too complicated to be used in the SoC design. The use case diagram is divided into proper scopes and redrawn as the *requirement analysis class diagram*. This process makes the system design robust under the specification changes, and highly reusable. Many approaches have been proposed to choose the candidates of the *class*; the *noun phrase* method is most frequently used. This method is used to get combinations of *noun and verb* from the use case of the diagram and take them as the candidates for *class and method*. It is quite easy to draw the class diagram from the use case diagram using the noun phrase method. For example, for the use case diagram in Figure 1.9, "data", "store request" and "exception" can be chosen as candidates for the class.

Some tips for the derivation of the class candidates are as follows:

1. If a noun is selected as a candidate for a class, other nouns with similar meaning are also taken as candidates for the class.
2. Each class candidate should be in charge of a unit job. A unit job is a behavior that an object is responsible for performing. In addition, the possibility of a change in system specification and reusability of the system are taken care of in the selection of the class candidate.
3. A review process for the selection of class candidates is required. For example, the following points should be considered:
   a. What is the unit job of the class? Is it clear?
   b. Do multiple classes have the same attributes or not?
   c. Can you make a clear differentiation between managing classes and managed classes?
   d. Can you find a connection for the classes with related concepts or similar meanings? Is it simple and clear?
   e. Can you denote ownership and implementation relationships correctly?
   f. Can the class diagrams cover all the implemented functions?

A class diagram for an SD memory card is shown in Figure 1.10. From the left to the right of the figure, the class diagram can be grouped into three groups, the classes related to SPI, the buffer, and the NAND Flash memory. Now, the design and analysis of the total system are reduced to those of the internal parts of three different

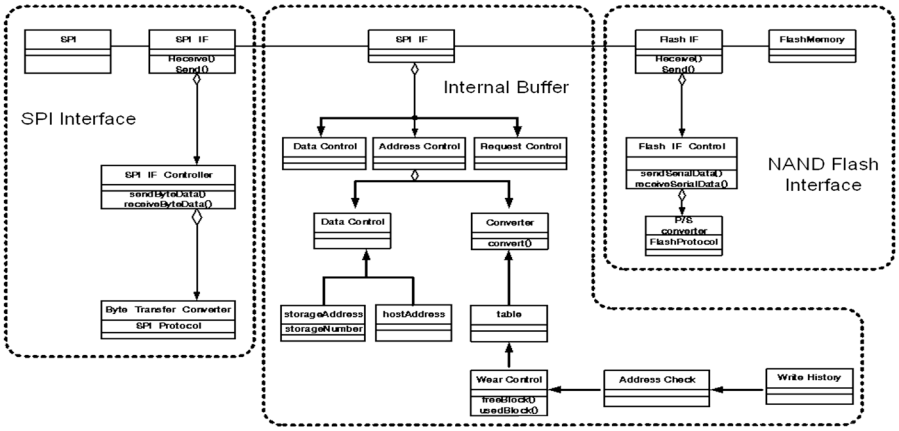


FIGURE 1.10 System analysis class diagram.

groups for better understanding, and for the clear definition of the functions required to be implemented.

The class diagram or the system analysis class diagram is for converting the use case diagram into a group of functions that the designer can understand. However, it is not sufficient for the design of the real SoC. A more detailed class diagram is required to match the system analysis class diagram with the real SoC design. That is, the designer needs to match each class in the system analysis class diagram with the related function that will be used in the real design. The set of these functions is the function specification of the target system. In addition, the relationship among classes (the lines connecting classes to each other) represents the data transactions; it later evolves into the communication channels and NoC integrating the functions, which will be discussed in more detail in Subsection 1.3.2 and Chapter 4. To derive function specification, the class diagram should be reviewed on the following points:

1. Derivation of necessary and sufficient methods: In each class, methods to achieve the assigned unit job should be clearly specified.
2. Derivation of the parameters to be saved after the method finishes: The lifetime of the parameters should be examined to get the internal parameters. If the parameters need to be saved for future use after completion of the methods, they should be shown as class attributes in the class diagram.
3. Multiplicity and the number of instances: The required number of instances for the real implementation should be carefully examined. This can be determined by the system specification; for example, multiple NAND Flash memories are required to provide the required memory capacity. The function requirement can give the number of instances. For example, the number of buffers needed to save the address is decided based on the address conversion algorithm and its speed, as well as performance improvement or the run-time work load. That is to say, parallel processing is possible with multiple instances of the functional block.

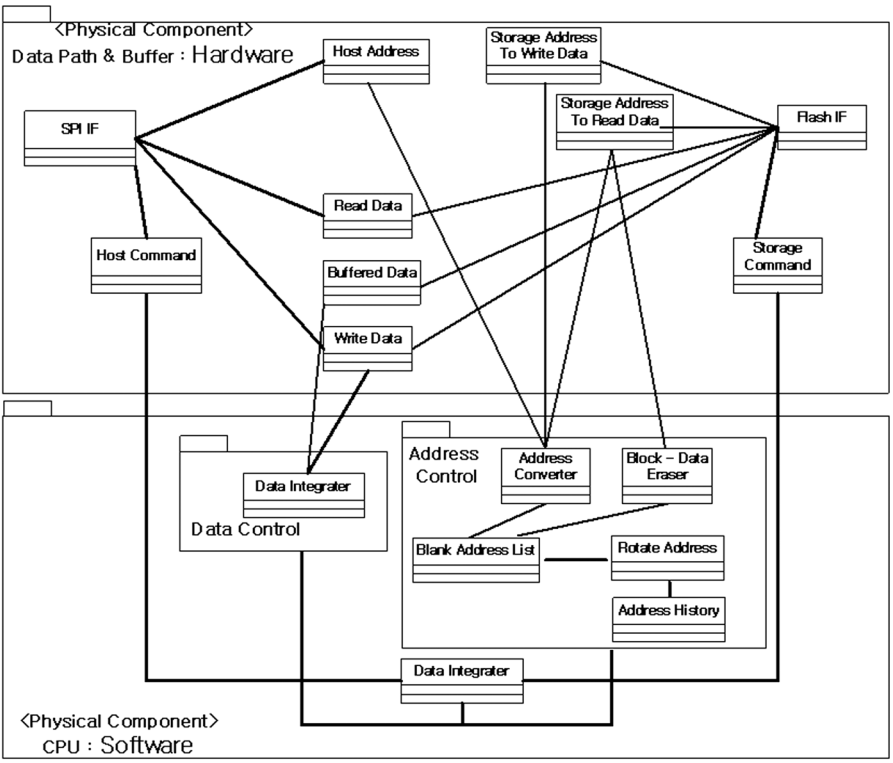


FIGURE 1.11 Class diagram for design.

The resulting new class diagram is the design-level class diagram as shown in Figure 1.11, which is based on Figure 1.10. In the system analysis class diagram, only the functions are depicted for the analysis and understanding of the system. However, in the design-level class diagram, the system is implemented in two parts, either in hardware or software, and the classes are divided and allocated to the software and the hardware groups. In Figure 1.11, the datapath and buffer part will be implemented by hardware; the CPU part will be implemented on the CPU, or by software. Of course, at this stage the communication among classes or sets of classes should be clarified for later use. A new class in charge of communication among classes can be introduced if there are many internal modules, such as the hardware and software parts, as in Figure 1.11. By introducing this class only for communication, detailed methods can be understood and described more accurately.

Next, the designer draws the sequence diagram to analyze the dynamic operation of the system. The sequence diagram is a picture showing the events that the external actors introduce into the system, their order, and interactions among them. When the sequence diagram is drawn, it is assumed that the message or information will be transacted only through the relationships (called *association* in UML) depicted on the class diagrams. First, the functions determined by use case analysis are assigned to groups of the classes on the class diagram. Then, based on a typical scenario of the use case diagram, the sequence of events from the actor to the system

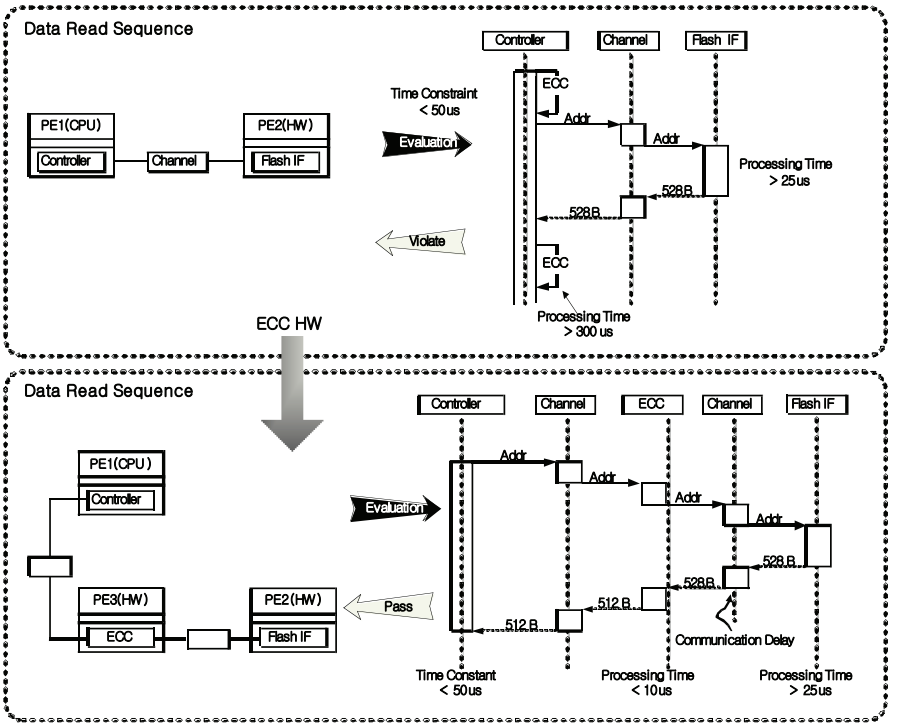


FIGURE 1.12 An example of sequence diagram.

or from the system to the actor is illustrated with the specific messages transacted at each event. If the data size and protocols for the transaction among classes are known in advance, the communication capacity can be estimated from the sequence diagram as shown in Figure 1.12, one based on the class diagram in Figure 1.11 for the sequence of data read from the NAND Flash memory. In this case, the classes are grouped into three major blocks as shown in Figure 1.10, and the messages between major blocks are assumed to be the real dataflow. As shown in Figure 1.12, a rough estimation of performance is possible; e.g., required processing time, available data transfer rate, and required bandwidth.

If the estimated performance is not satisfactory (for example, if the processing time is found too long), the designer looks for the most time-consuming function in the software part and realizes it by hardware to accelerate its processing speed. In the example shown in Figure 1.12, error correction processing in the Wear-Leveling class consumes the most time and deserves to be realized by hardware. In this case, to get better performance, the new sequence diagram becomes the lower graph of Figure 1.12. A dedicated hardware realizes the Wear-Leveling algorithm; so the time required for Wear-Leveling processing is reduced, and the total processing time decreases, too. For the realization of Wear-Leveling by hardware the class diagram has to be modified, as shown in Figure 1.13. The hardware part of Figure 1.10 is divided into two, and a dedicated module for Wear-Leveling processing is defined.

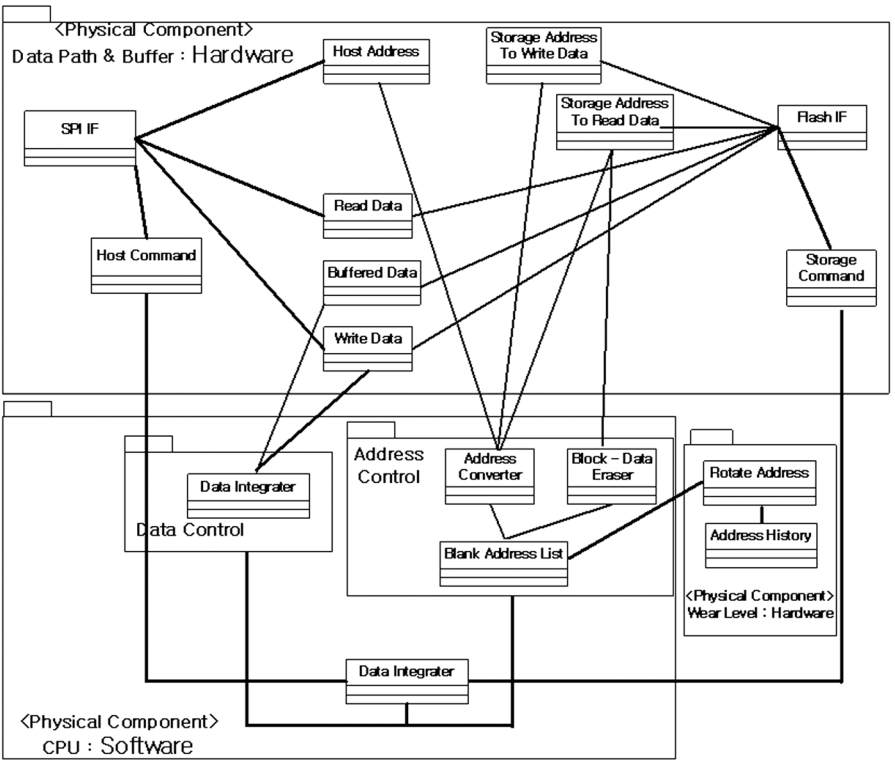


FIGURE 1.13 Refined class diagram.

Thus performance estimation using the sequence diagram improves the design, including the hardware–software partitioning. Although an example of performance estimation and design improvement is shown here, in reality, at this early stage, detailed hardware–software partitioning and the identification of system bottlenecks are not easy. More detailed application of UML in the analysis and design of SoC will be explained in Part II with performance simulation.

1.1.3 Architecture Design

Gajsky et al. proposed SpecC to describe the specification of a system [7]. SpecC can be converted into the RTL in the long run. Gerstlauer and Gajsky introduced the details of relationship between functional model (behavior diagram in Spec C) and architecture model [8]. Here, we summarized their methods to explain how to divide the communication from the computation.

Behavior Design: In Subsection 1.1.2, UML system analysis resulted in a function model as a set of functions and their intercommunication for the required system operations. Figure 1.14 is an example of behavioral specification. Usually, system-level functions are referred to as *behaviors* to avoid confusion with software functions. Individual behaviors can access the shared variables to accomplish concurrency if the behaviors require communication and synchronization. To define

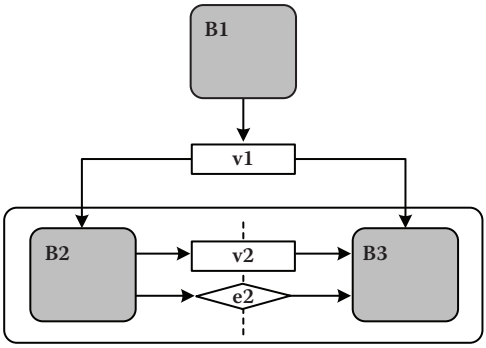


FIGURE 1.14 An example of Behavior Model [9].

the sequence of behaviors, a mechanism for achieving synchronization is required. Generation and consumption of an event are frequently used for synchronization. The consumption of an event to start a behavior means that the behavior has waited for the generation of the event. In this way, the cause and result, or sequence among behaviors, can be obtained. In this section, the mapping of behaviors to the processing elements, or the architecture design and model, will be discussed.

Architecture Design: In the architecture design, a macroscopic structure of the target system is determined by mapping the behavior of the processing element (PE), and then basic methods to implement the PE are decided. The result of the architecture design is an architecture model. In this stage, it is determined whether each PE is newly designed into hardware, reuses the preverified hardware design, or is implemented as software on a different PE. It is also determined whether the PEs operate in sequence or in parallel. In addition, once the PE and its implementation are determined, processing time can be estimated based on previous experience. Even computer simulation is possible if the architecture model is described in C or C^{++}, and the system timing can be analyzed to obtain the optimal system architecture.

The purpose of architecture design is to transform the function specification into the architecture model and, more specifically, to divide it into five design steps: PE allocation, behavior partitioning, variable partitioning, channel partitioning, and scheduling.

1. PE allocation: This stage determines the types and numbers of the components constituting the system. Hardware components are PE (RISC, CISC, VLIW, SIMD, MIMD, DSP, hardware accelerator, etc.), memory (SRAM, DRAM, FIFO, frame buffer, register, etc.), bus, and on-chip network. In addition, standardized or well-defined components can be selected from the IP library. However, details of the PE's internal operation are left undecided and will be designed in a later step.
2. Behavior partitioning: The individual behavior is divided as allocated to the PE. The communication description is modified accordingly.
3. Variable partitioning: The shared variables are divided and allocated to memories. Internal variables inside the PE are mapped to local memories or registers. If necessary, each PE can hold a copy of the variable, and one

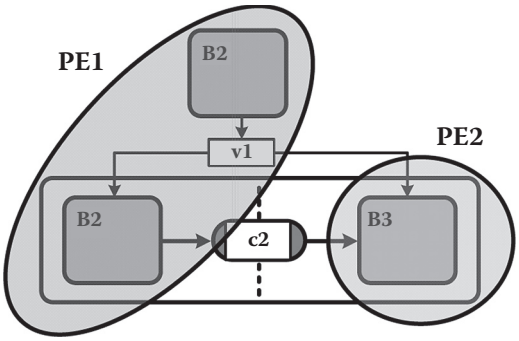


FIGURE 1.15 PE allocation and behavior partitioning [10].

PE will operate as the server. The communication description is modified accordingly.

4. Channel partitioning: In conventional design, the inter-PE channels are divided and allocated to buses. For this purpose, the designer usually introduces a channel representing the system bus and encloses other channel descriptions inside. For 1:1 communication, the bus is not necessary. As the number of PEs increases, the channel partitioning becomes complicated and an on-chip network is introduced. This is the main theme of this book.
5. Scheduling: The parallel behaviors allocated on the same PE are time-shared, or the PE operations are scheduled.

In Figure 1.15, PEs are allocated to the behaviors shown in Figure 1.14. Behaviors B1 and B2 are allocated to PE1, and behavior B3 to PE2. Each PE can operate concurrently, and in the PE allocation stage the processing power required for each behavior should be taken into consideration. In this case, a cost function such as the minimum processing time is derived, and through trial and error the PE allocation is performed to minimize the cost function. In Figure 1.15, behavior B3 requires more processing power and so is allocated to the dedicated processor, PE2, which may be a hardware accelerator, to speed up its processing.

Figure 1.16 shows the system architecture model after the PE allocation. PE1 and PE2 operate in parallel, but the behavior B3 in PE2 starts its operation in parallel with behavior B2 in PE1 only after behavior B1 in PE1 finishes its operation. For this purpose, a new behavior, B13snd, is inserted into PE1 which starts behavior B3 in PE2 in parallel with behavior B2 in PE1. B13snd sends a "B3 start signal" through the channel cb13. A new behavior, B13rcv, is inserted into PE2 to monitor the "B3 start signal" by accepting the start signal through the channel cb13. To inform PE1 of the end of behavior B3, a new behavior, B34snd, is inserted into PE2 and another new behavior, B34rcv, is inserted into PE1. Therefore, behavior B2 communicates with behavior B3 through three channels, as shown in Figure 1.16, unlike communication through the shared variable and the event in Figure 1.14. As PE2 will be implemented with hardware, the channel should be described to easily match PE2's hardware architecture.

The last step in the architecture design is to schedule execution of multiple behaviors on PEs, which are sequential machines. Usually, the number of behaviors

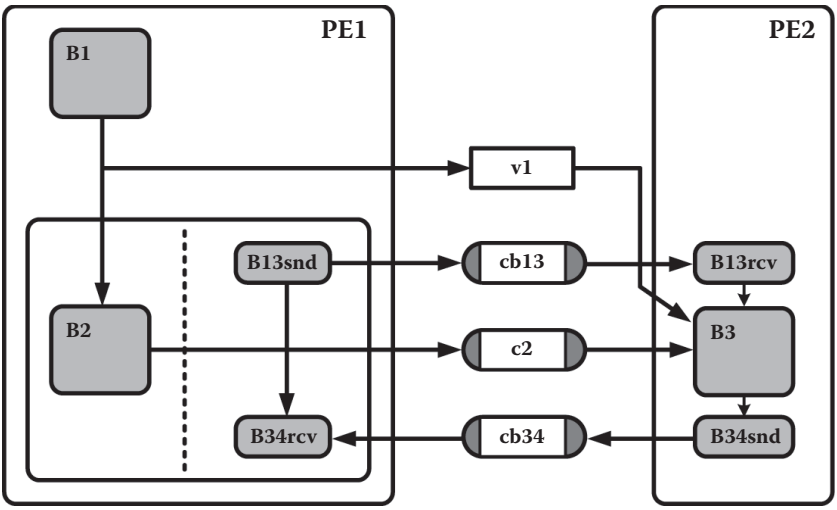


FIGURE 1.16 Architecture model after behavior partitioning [11].

is greater than the number of PEs, and multiple behaviors are processed by a single PE by time sharing or in sequence. The purpose of scheduling is to put the predetermined order of execution in each behavior. In Figure 1.17, behaviors B2, B13snd, and B34rcv are serialized such that B13snd precedes B2, and B34rcv is scheduled after B2. The scheduling result is shown in the right-hand side of Figure 1.17. Of course, the order of execution can be dynamically determined so that one behavior from a pool of behaviors is allocated to a PE during the runtime according to a given scheduling algorithm. In this case, the scheduler behaves like an operating system.

The addition of new behaviors and channels during PE allocation modifies the system model into the architecture model. After PE allocation, the main issue is how to implement the PEs by either software or hardware. Cost functions can be derived from system constraints, and based on the results of the simulation or proper estimation, the PE allocation can be retried. Such simulation or estimation to look for the optimum design is called *design space exploration*.

Communication Design: The architecture model may result in multiple PE architectures that are in parallel operation and need to communicate with each other. A detailed description of their communication is essential; this is called a communication design, and its result is called a communication model. For example, the communication model in Figure 1.18 shows that the shared variable v2 and the event e2 provide communication synchronization to the parallel behaviors B2 and B3. Although this communication architecture with a shared variable is possible if both behaviors, B2 and B3, are processed by the same processor with software, it cannot be realized if any one of the behaviors is processed by the hardware. Because the hardware usually communicates with the help of signals through I/O ports, the communication model should be modified to use a message-passing channel as in Figure 1.18 (bottom). Putting the event synchronization and shared variable into a channel is referred to as *encapsulating*. This provides the designer with abstract tools, which clarifies the communication and can later be synthesized as an RTL description.

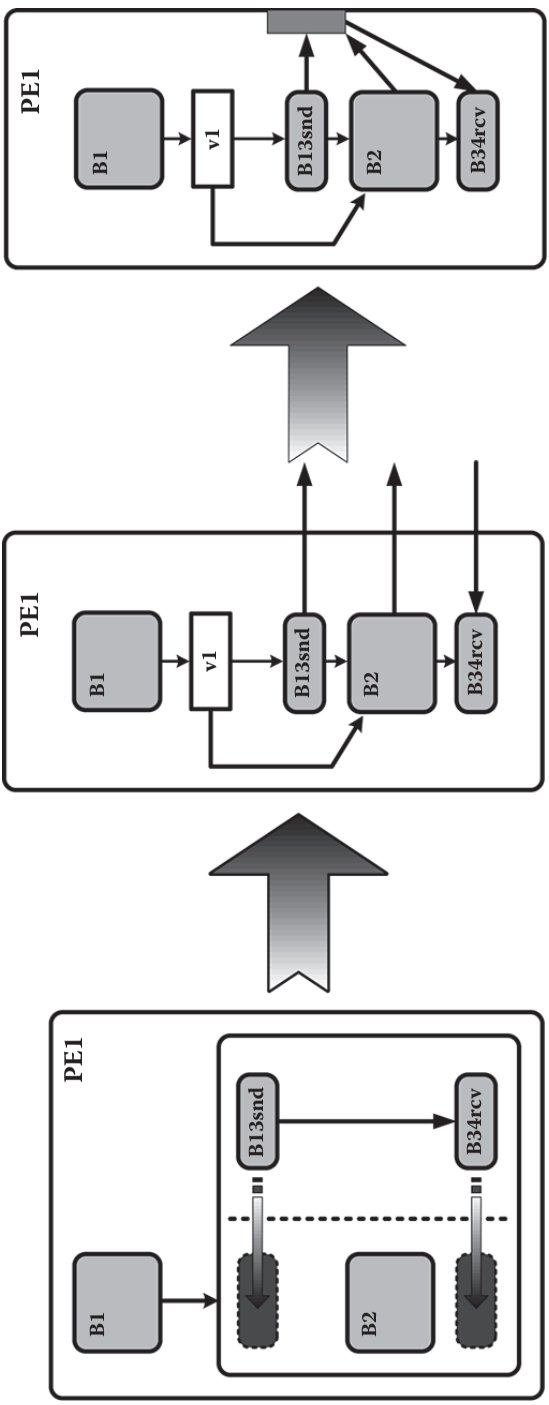


FIGURE 1.17 Model after scheduling and channel partitioning.

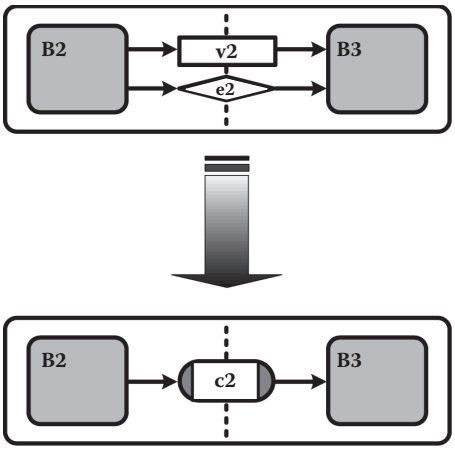


FIGURE 1.18 Message-passing channel communication model.

Next, communication synthesis decides how to realize the channels. Multiple channels are combined into one channel, which will be implemented as a system bus, as shown in Figure 1.19. A communication protocol that can implement the total communication of the combined channels is determined and inserted into the communication model as shown in Figure 1.20. Once the communication protocol is determined, the models of PEs to be attached to the system bus should be modified according to the protocol. Two methods are used for this purpose. The first is to modify the PE model itself, and the second is to put a wrapper into the channel to translate the protocols between the system bus and the PE. Of course, such a wrapper may be incorporated into the PE. But if the IP is reused as the PE, the modification of the IP itself is difficult; so the wrapper is included in the channel. In this case, it is as if a specific PE, i.e., the wrapper, is added to the system. Figure 1.20 shows that a handshake protocol is adopted in the system bus by adding two control signals, ready and ack. The communication PE can be implemented by either software or hardware in the same way as other PEs.

1.2 PLATFORM-BASED SOC DESIGN

A platform is a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and delivered to customers for quick deployment [2].

1.2.1 Concept of the Platform

As the complexity of the chip design increases, RTL-based designs are inappropriate to handle the complexity and currently preverified design blocks (subsystem, megacell, virtual chip, and IP) are reused. However, with more blocks from outside the design team included, the interface protocol mismatch and timing errors between blocks are increasing dramatically. A high degree of freedom in design may be an obstacle for the quick and secure design of the system. Although

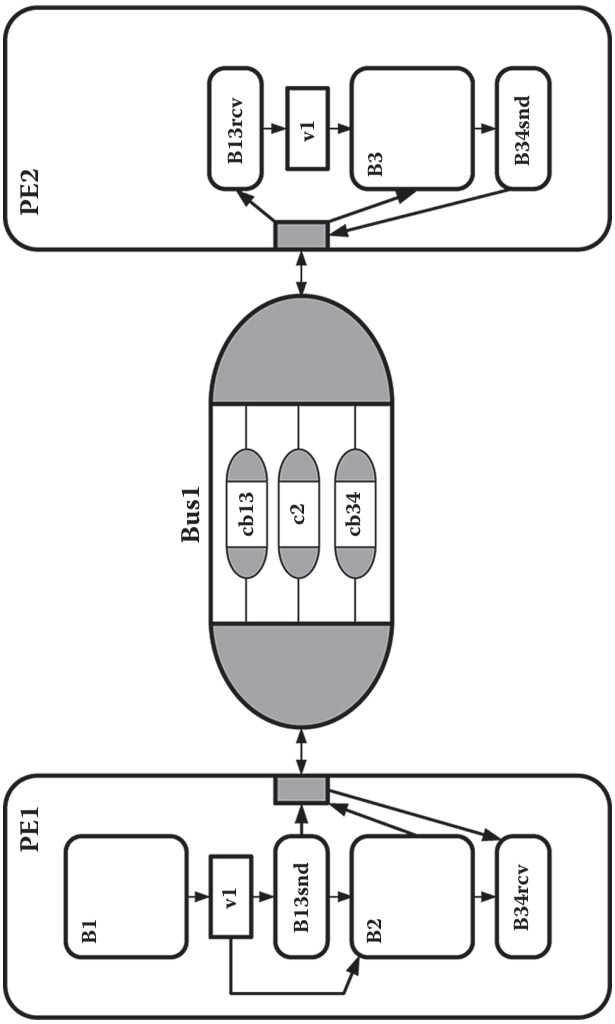


FIGURE 1.19 Channel multiplexed model [12].

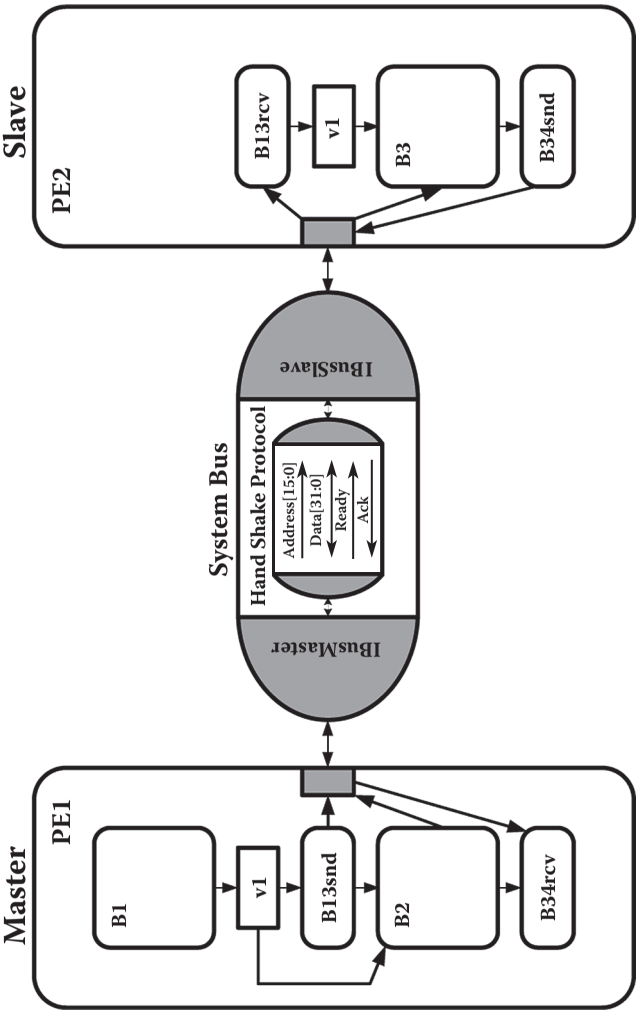


FIGURE 1.20 Communication model after protocol insertion [13].

reducing the degree of freedom in design may enhance design efficiency, the possibility to get a dramatically new design would be lower. This can be realized by fixing most of the hardware blocks, such as microprocessor, bus, and I/O peripherals, and allowing only limited modification, such as adding new hardware blocks as slaves to the main processor, and minor or no modification of the system bus. This preverified architecture with a limited degree of freedom is called a *platform*. That is, the system architecture itself can be reused similar to IP if it has been verified as a reliable and convenient solution for a certain specific application. It is an "application-oriented architectural template tool kit" for the system designer, convenient to use. Figure 1.21 is an example of a platform for the DECT standard wireless communication system. For the efficient reuse of the system architecture, the main hardware blocks of the system will be maintained without modification, and only the hardware blocks necessary to implement the specific requirements of the new system need be added and modified. It helps the designer to implement the concrete system not only at the architectural level but also the functional model stage, because the information on the key architectural components is available already. Furthermore, platforms typically also contain some software blocks (API, operating system, and middleware solutions), design and application methodologies, and a toolset to support rapid architectural exploration.

In the electronic appliance domain, it is common to develop a new product that shares a substantial fraction of components with the previous one, because it can give a differentiation in the product with less development cost. The commonality shared by these products is nothing but the platform. The platform can be planned before product development, or a successful product can be a platform, as schematically shown in Figure 1.22. The former is more like a chip integration platform; the latter application platform is explained in more detail in Subsection 1.2.2.

For example, in PC products, the concept and use of platforms are widely applied to develop related products. Typically, X86 CPUs determine the platform irrespective of the manufacturers. In addition, buses such as ISA, PCI, and USB are standardized and, regardless of the board manufacturers and specifications of peripheral devices, such as keyboard, mouse, audio and video devices are perfectly matched. Of course, the operating system and other software packages are fully compatible.

But many people attach new devices and install new software to transform their PCs into special-purpose systems. This is the ideal example of the platform, but in SoC design, the design space is too wide to apply the PC concept directly. The IP interface is not standardized, and even the CPUs are of too many types.

1.2.2 Types of Platforms

There are many platforms used in industrial SoC development. Examples are the TI OMAP, the Phillips Nexperia, the ARM PrimeXsys, and the Xilinx Vertex-II Pro. The platforms can be categorized into three different layers, as shown in Figure 1.23—application integration platform, SoC integration platform, and basic (hardware) platform [4]. In addition, they can be divided into four different groups: processor-centric platform (TI, OMAP), application-centric platform (Nexperia, Hera), communication-centric platform (Fulcrum), and FPGA-centric platform, as

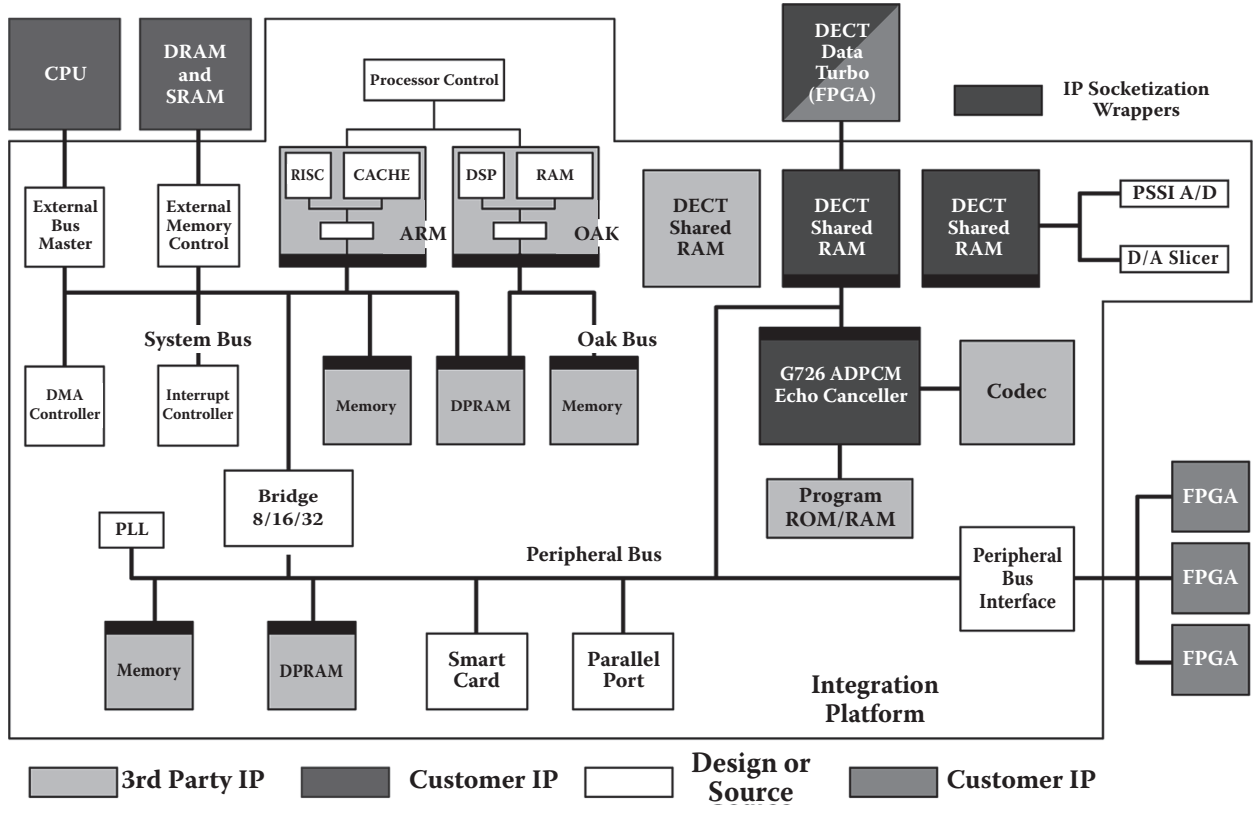


FIGURE 1.21 DECT wireless system integration platform [14].

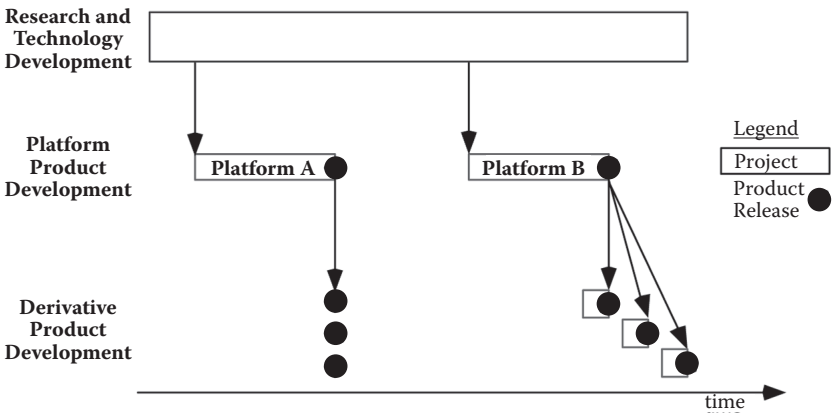


FIGURE 1.22 Concept of the platform-based design.

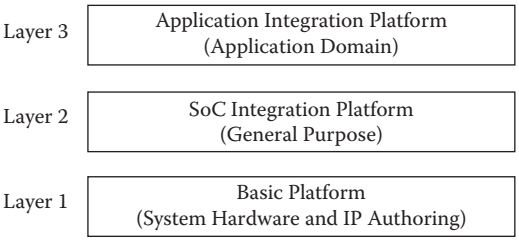


FIGURE 1.23 Three layers of platforms.

shown in Figure 1.24. From now on, we will examine the different platforms based on these classifications.

To implement a platform, availability of many useful IPs is very important; most early efforts in platform research is focused on IP authoring and system integration. The IPs, including processors, buses, and memory architectures, are collected and arranged, usually into a library, and basic system integration methods are provided as design guidelines. This can be called a *basic platform* if the library and guidelines are coupled with specific fabrication technology. Early versions of the platforms fall into this category [4,15]; these have later evolved into other advanced platforms.

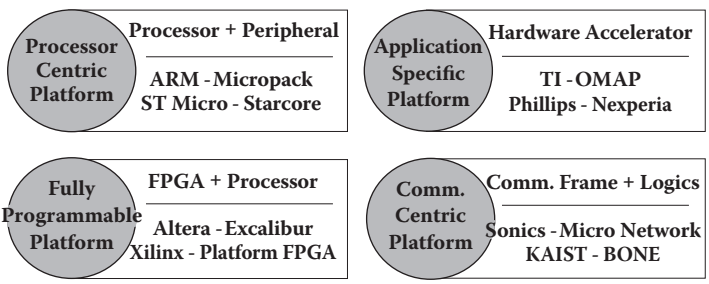


FIGURE 1.24 Four different categories of platforms.

If the basic platforms are more refined and get structured with the hardware kernels, hardware/software IP libraries, and the platform model, the most application-general platform, a chip integration platform, is formed. The platform has the same architecture as the general embedded system, with a CPU, a set of hardware IPs, a system bus, and peripheral blocks for the off-chip interfaces. The CPU plays the most important role in the platform, which can be applied to any application for which the CPU, memories, and bus architectures are appropriate. If the target application is given, an application-specific hardware IP will be selected and adapted to the bus architecture. The power and clock distribution, I/O mapping, test architecture, and related software architectures should be modified with the application-specific IPs and the required application. Its advantage over the basic platform is its flexibility and the reuse of key IP blocks such as CPU, memories, and buses. However, it requires the adaptation to the specific application, verification of the IP blocks being integrated, and a wide range of variability in power consumption, area, speed, and performance of the final chip. Therefore, it requires significant effort to complete the design of the target chip.

The next level of the platform is more application-domain- and process-technology-specific than the other platform types. It pursues more freedom of design and more market-dependent products within a short development time. More than 90% of blocks are from the existing IP library, and value addition comes from unique product-related hardware blocks, software algorithms, and timely market introduction and compatibility. Some of the main characteristics of the application integration platform are as follows: application-domain-specifically, chip integration specification, full IP library, proven interfacing and integration methods, verification environments, and embedded software support architecture.

The four categories of the platforms reflect the main technical backgrounds of the companies or organizations. Communication-centric and fully programmable platforms are more bottom-up approaches and good for layer 1 or 2. Application-specific and processor-centric platforms are good for top-down development and suitable for layer 3. The platform architecture can be analyzed as shown in Figure 1.24 by its processor architecture, memory and peripherals, bus/network architecture, accelerator/coprocessor, reconfigurable blocks, and software. Among the four different platforms, the processor-centric and application-specific platforms are more application oriented and fully programmable and communication-centric platforms are more system-integration-oriented. The communication-centric platform is the theme of this book and will be explained in the following chapters. In this section, we will take a look at the other three platforms.

1.2.2.1 Processor-Centric Platform

The processor centric platform has the most general architecture among the platforms. Similar to the X86-based PC platform, it is composed of a well-known processor, its bus, and other peripheral blocks to easily add coprocessors or accelerators and other more specific peripherals. An example is shown in Figure 1.25. Software solutions are emphasized and only if the performance is insufficient, coprocessors such as DSP or accelerators such as H.264 decoders are added. Of course, it is very

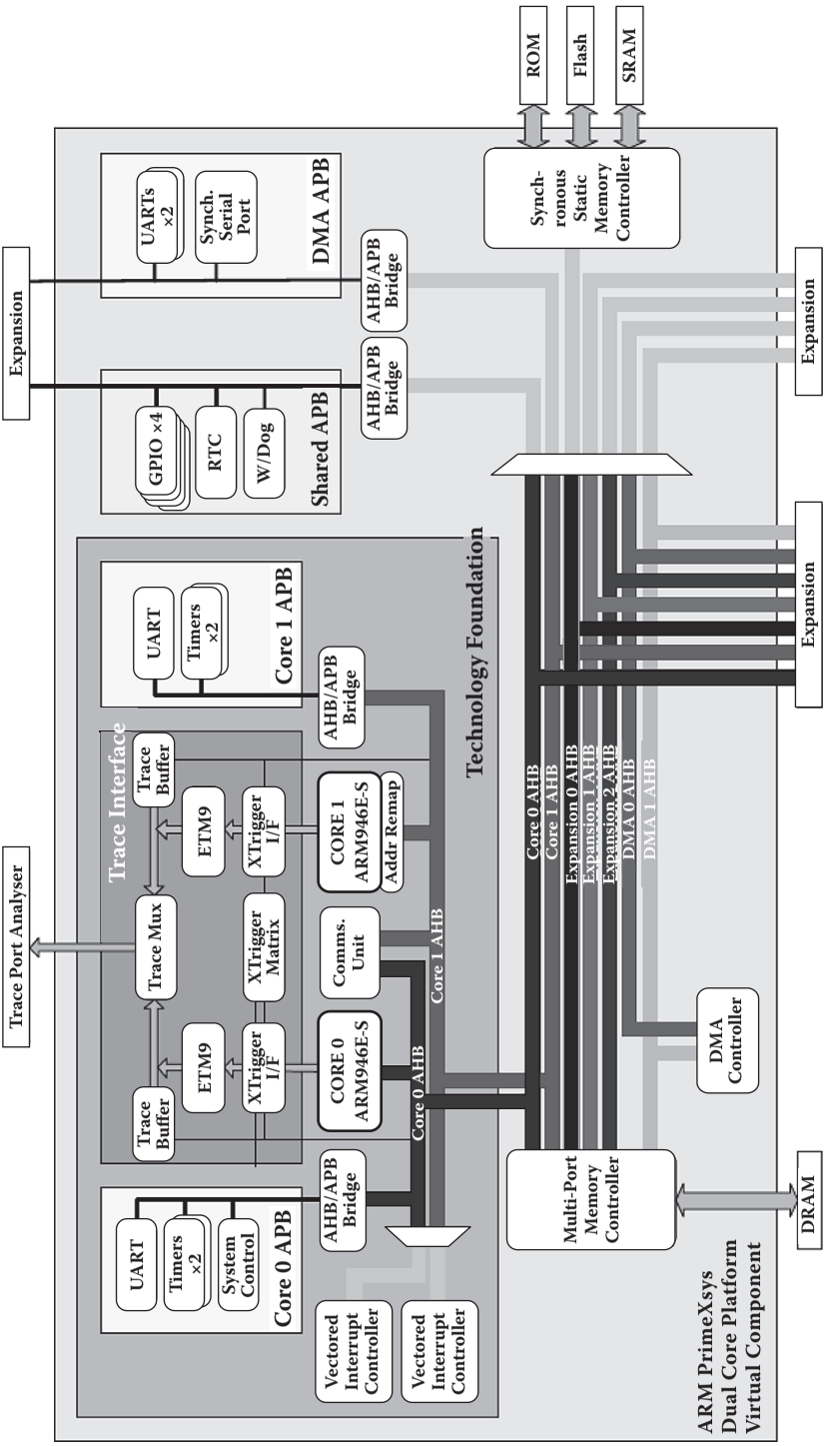


FIGURE 1.25 The PrimeXsys ARM dual core platform [16].

easy to get the software solutions as the processor is well known and many examples already exist. Let us take the ARM-based platform as an example. This platform has been widely used in low-power portable and handheld devices such as cellular phones and PDAs.

The ARM-based platform supports platform layers 1, 2, and 3 of the previous explanation. Layer 1 of the platform has an open-standard, on-chip bus specification—AMBA bus interface standards. The AMBA specification has three bus protocols: high-speed Advanced eXtensible Interface (AXI), general advanced high-performance bus (AHB), and register-based advanced peripheral bus (APB). It also supports the memory architecture, including local and global memory allocation, memory controller policies, caching policies, and memory-space mapping.

At platform layer 2, the SoC integration layer, the main focus is on the hardware blocks directly interacting with OS. For example, device driver blocks, which are configured to the OS and system memory map, are integrated into hardware blocks. For this purpose, ARM provides the AMBA design kit (ADK) and the AMBA compliance test bench (ACT). ADK contains the IP library, example integration, and test components. ACT is an interface-protocol checker for validating compatible IP designs. In addition, the SoC resource allocation tool PMAP (peripheral map) defines the local register descriptions for each peripheral, including register bit maps and reset values. A key feature of the ARM platform is energy management to support dynamic voltage scaling for low power consumption; layer 2 includes energy management. A hardware prototyping using FPGA or FPGA-based multiboard prototyping tools can be implemented and application software can be developed on this platform capable of multimegahertz speeds.

The ARM solutions are so widely used in portable systems that software requirements are very stringent. The software development layer enables quick development of complex stacks involved in wireless communication and multimedia applications. Earlier, the software was developed on the PC and later retargeted to the developed SoC. However, this approach has many drawbacks. For example, the computing power of the PC, including the number of threads, is different from those of the ARM processor. FPGA-based hardware prototypes can be a solution but have a mismatch in the detailed hardware configuration; only a limited number of platforms are available, which discourages the mass proliferation of a platform among thousands of developers. Instruction Set Simulator (ISS) provides a low-cost emulation of the target software. However, it is not fast enough to execute the complex application software and OS needed by modern embedded devices. A new emulation technique is used in the name of the virtual platform in which an image or "skin" of the target device is updated by the software emulator in real time.

1.2.2.2 Application-Specific Platform

The application-centric platform is more close to the target application in terms of hardware architectures and software applications. For example, the Philips NEXPERIA (NEXt EXPERIence) platform [17] is for digital video appliances such as DTV, DVD players, digital video recorders, and setup boxes. The TI OMAP (open mobile application processor) allows multimedia application in wireless handsets

and PDAs [18]; its hardware and software platforms are shown in Figure 1.26. For the application-specific platform, the three layers of the processor-centric platform are applied in the same way.

But it has its own unique features such as heterogeneous multiprocessor architecture and OMAPI standards. It has two masters, ARM 9 processor and TI C55 DSP. ARM plays the role of general-purpose controller, such as dynamic task creation and destruction on the DSP, and resource management of memory and processing power on the DSP. Message exchange between ARM and DSP is through the DSP bridge with two mailboxes. Of course, ARM and DSP have separate memory architecture, device drivers, and OSs.

The OMAPI standard provides software interfaces to OS and hardware interfaces to common peripherals. For example, USB and synchronous serial interconnect are for the connection to wireless modem.

1.2.2.3 Fully Programmable Platform

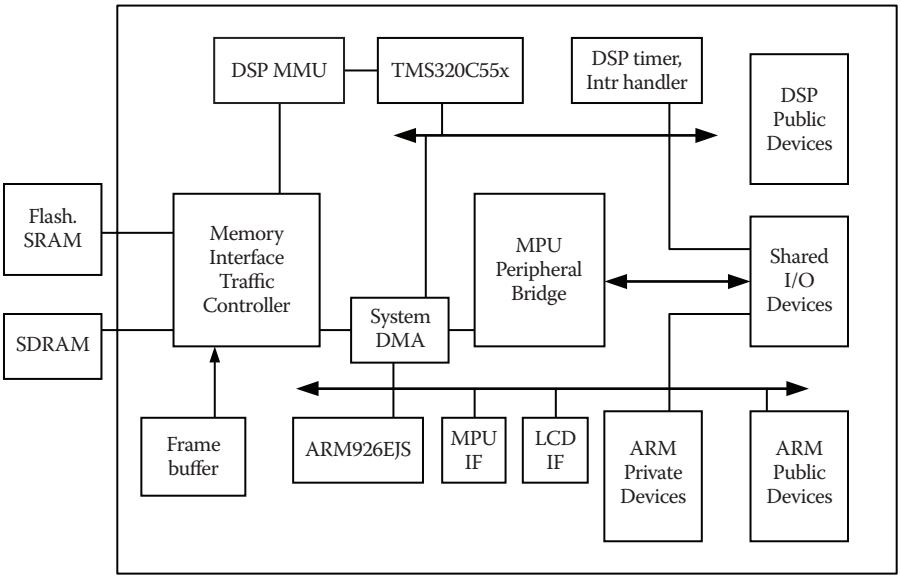
Programmable chips from Xilinx and Altera have been frequently used for fast prototyping and hardware emulation. Especially for systems people such as communication engineers, programmable components are used to meet communication standards as soon as possible. Their relatively large chip sizes can be tolerated in large wire-line systems in which reference designs are provided as a part of the operations of standards, although they are not optimized for SoC implementation. IP blocks, which are frequently required by communication engineers, are integrated as hardware blocks inside the programmable logic arrays.

Let us take Vertex-II from Xilinx as an example (see Figure 1.27). It integrates a 32-bit PowerPC RISC core and multigigabit serial transceivers in a two-dimensional array of configurable logic blocks. It has multiples of 18K-bit dual port RAM, block select RAM (BRAM), 18 bit × 18 bit signed multipliers, and multiple digital clock managers (DCM) for the internal clock synthesis. Its internal bus adopts IBM's CoreConnect bus, which comprises three separate buses for interconnecting processor blocks, IP blocks, and custom logic. The three buses are the processor local bus (PLB) for high bandwidth and low latency, the on-chip peripheral bus (OPB) for slower peripheral cores, and the device control register (DCR) bus to manage the status and control registers of peripherals.

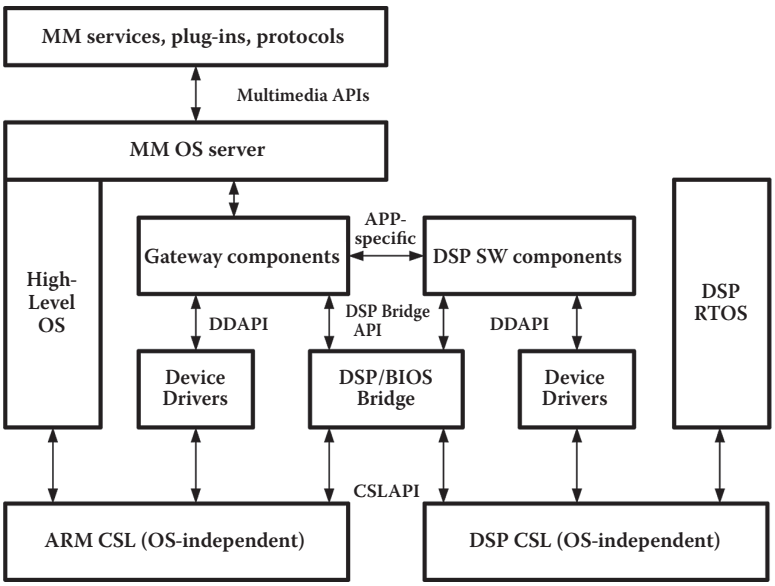
The integration of many IPs inside the programmable chip is contrary to its general-purpose programmable characteristics. In addition, neither can give satisfactory solutions to each area. New research activities such as dynamic reconfiguration and self reconfiguration are underway, but no clear application has been found.

1.2.2.4 Communication-Centric Platform

SoC integration is similar to a Lego play; the interface standards are important to simplify the development of complex systems and reduce the need to design glue logic that potentially degrades the performance of the system. Standard interface architectures provide rapid assembly of IP blocks into SoC as well as guidelines for the design of the IP itself. Bus-based standard practices lead to IBM CoreConnect [20], AMBA, and OMAPI.



(a) Hardware platform



(b) Software platform

FIGURE 1.26 Platform schematics of OMAP 5912 [18].

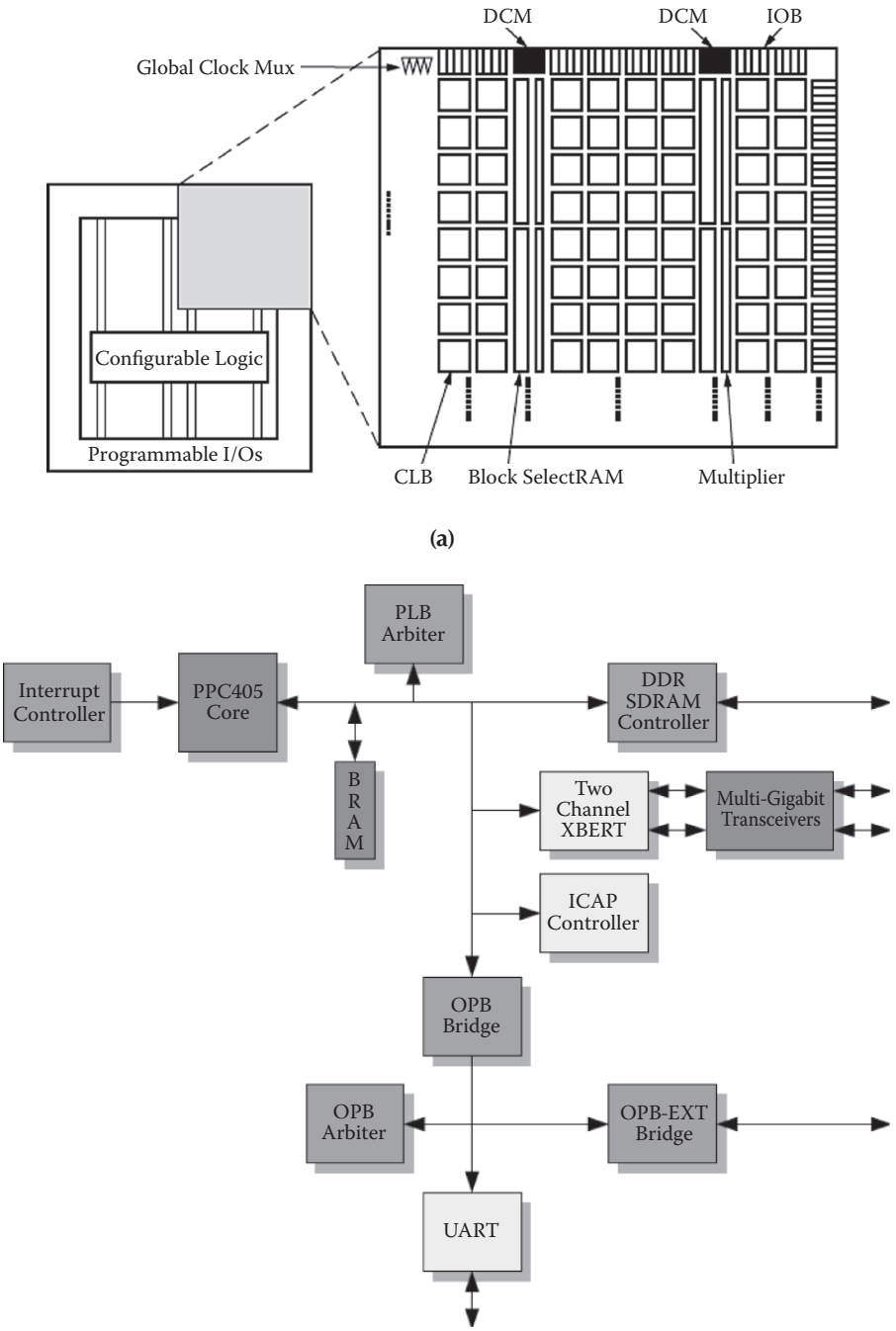


FIGURE 1.27 Vertex-II platform: (a) architecture and (b) its block diagram [19].

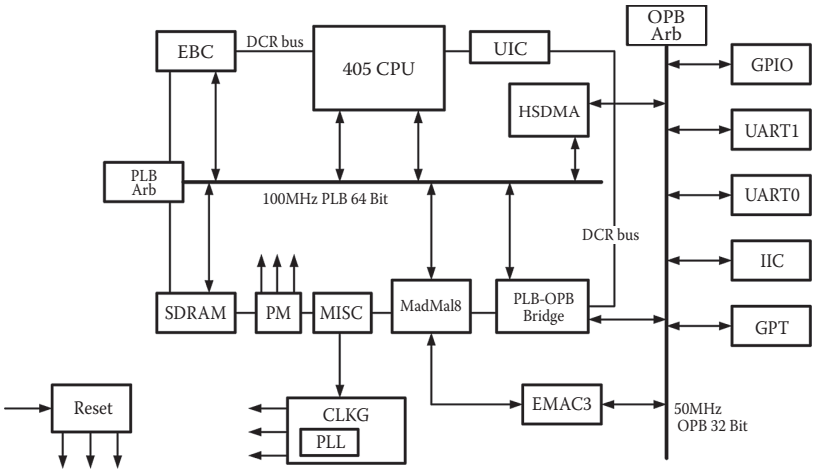


FIGURE 1.28 CoreConnect platform [20].

The CoreConnect of Figure 1.28 comprises three bus types, as briefly mentioned in the previous section; other bus standards also have similar architectures, as shown in Figure 1.28. It is very natural for the SoC design to borrow board-level design practices such as bus-based system designs. However, the bus has critical disadvantages for the complicated SoC. As the number of IPs connected to the bus increases with the integration scale, the bus performance degrades because of increase in parasitic capacitances and resistances. In addition, the bus contention and related arbitration such as round robin and TDMA get complicated, degrading the access time.

To overcome these drawbacks, a temporal solution is proposed by Sonics [21]. They use separate optimization of the communication channel, such as an IP dedicated to communication to optimize the target SoC, rather than fixed interface standards. Sonics decouples communication from computing and introduces *"silicon backplane,"* a communication subsystem that can be tuned to the required bandwidth (Figure 1.29). They provide flexible interface sockets, decoupled agents offering dataflow service, and the internal communication fabric. They adopt the Sonics Module Interface to configure the interface of IP and provide a design software

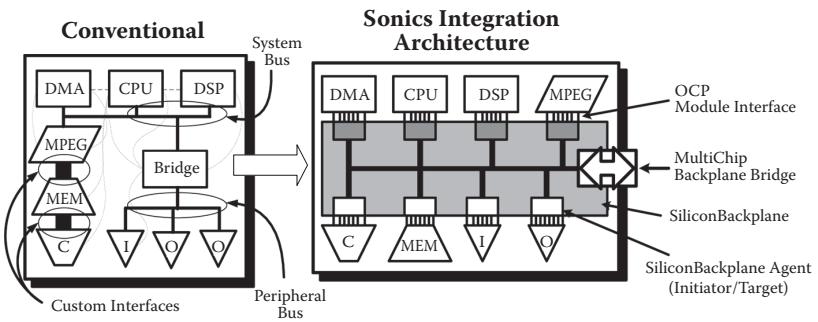


FIGURE 1.29 Sonics silicon backplane [21].

named "*backplane compiler*." To decouple communication from computation, they introduce the FIFO buffer and burst transfer of data by grouping the related transfers into *bursts*. The sender buffers a burst's length of data before beginning the transfer into an equivalent FIFO buffer at the receiving device. This is effective to decouple the computation-intensive data transfer, which is maximized for best-case performance but show poor performance in satisfying real-time deadlines. In contrast, communication-intensive buses are designed to satisfy real-time constraints—in other words, worst-case performance. In this case, TDMA is used to transfer the data across a higher-bandwidth channel with minimal buffering, and higher-level protocols are adopted to select the receiving device [21]. The result is highly efficient interleaved transfers. However, this higher-level processing introduces delay and inefficiencies that are unacceptable for latency-critical operations such as CPU cache line refill. They try to match the data rates of different IP cores by adopting data transfer and arbitration at a single, fully pipelined bus cycle. They use two phases for arbitration: a distributed TDMA method for the predictable data transaction and a fair round-robin method for unpredictable data transaction.

However, these approaches have failed to overcome the drawbacks of the bus in providing modular and scalable communication channels to the SoC. The theme of this book, NoC, is the natural solution to the internal communication issues in SoC. It is because NoC can provide solutions with scalability, modularity and complete isolation between communication and computation.

1.3 MULTIPROCESSOR SOC AND NETWORK ON CHIP

1.3.1 Concept of MPSoC

The SoC platform has recently evolved into Multiprocessor SoC (MPSoC). The SoC or the embedded system follows the architecture of the desktop computer because most system engineers are familiar with the PC and its software solutions. The current PC contains multiple processors, multicore CPU, DSP, and other application-specific processors—to support high computing power with less power consumption for advanced applications such as multimedia or 3D games [22]. So far, the multiprocessor or multicore has been studied in computing discipline as a part of parallel computing. The purpose of developing the multiprocessor system is to improve throughput, scalability, and reliability of the computing system. They have established a well-developed theory and set of practice on the parallel computer. They can be split into two categories, centralized computing and distributed computing, according to the locations of the hardware and software resources. Centralized computing, which has been studied to construct the super computer, is more appropriate for MPSoC. However, there is a clear difference between the multicore or multiprocessor and MPSoC. MPSoC provides a system solution like the Emotion Engine (Figure 1.30), giving full video and graphics solutions, whereas multicore or multiprocessor is just a processing block.

Ideally, the system with n processors shows n times faster performance than a single processor, but in reality, its speed-up ranges from a lower-bound $\log_2 n$ to an upper bound $n/\ln n$ due to conflicts over memory access, IO limitations, and inter-

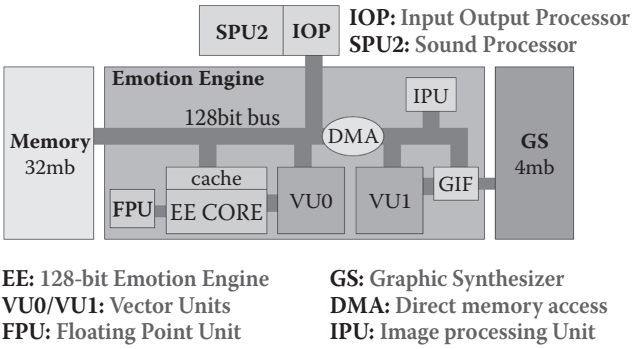


FIGURE 1.30 Architecture of Sony PS2; Emotion Engine has multicore [22].

processor communications [24]. The most famous classification method is Flynn's, SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), and MIMD (Multiple Instruction Multiple Data). Other classification methods are distinguished by having a shared common memory or unshared distributed memories. Among the four different architectures, MIMD architecture is most appropriate to MPSoC.

The MIMD architecture can be regarded as an extension of the uniprocessor single memory + single processor architecture. There are two alternatives for assembling the multiple processors and memory modules. One simple way is to make the processors and memories as pairs and then connect them via an interconnection network, as in Figure 1.31. The processor + memory pair, or a PE, works rather independently of each other, and the memory inside one PE is hardly accessible directly by the other. This class of MIMD may be called as *distributed-memory MIMD* or *message-passing MIMD* architecture. The other way is to group the processors and memories into separate modules, every processor can access any memory through the interconnection network. The set of memories makes up a global address space, which is shared by the processors. This type of MIMD is called *shared-memory MIMD*. Wang summarized previous parallel computer research into six different

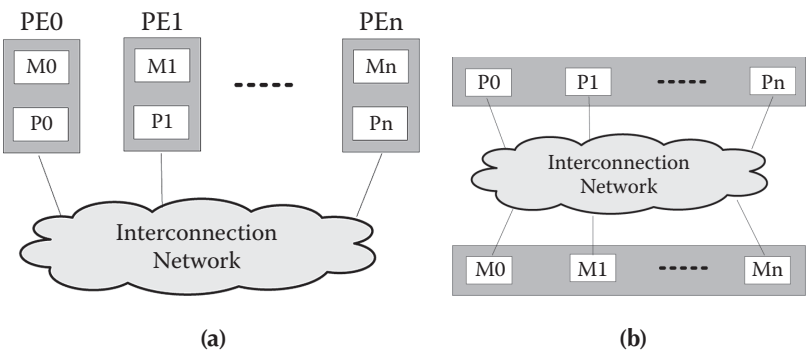


FIGURE 1.31 (a) Distributed Memory and (b) Shared Memory Multiprocessor Architecture.

tracks in more detail—shared-memory track, message-passing track, multivector track, SIMD track, multithread track, and dataflow track [6].

Most of the parallel computing does not specifically mention the heterogeneity of the architecture. Usually it is assumed that all the processors are identical and only the programs running on them may be different. However, if the architectures of the processors are different from one another, the matter is more complicated, and a more detailed approach, such as MPSoC, is necessary. Its processors may be of different types, its memories may be different from one another and distributed heterogeneously on the chip, and the interconnection network between the PEs may be heterogeneous.

1.3.2 MPSoC and NoC

Previous sections showed that the types of computing elements determine the multiprocessor architectures. Here, we will look at it differently that the interconnection structure among the memories and processors can determine the multiprocessor architectures. Three different interconnection methods have been proposed:

Shared bus
Crossbar switch network
Shared (multiport) memories

The shared-bus system shown in Figure 1.32 is very simple to use. In addition, it is compatible with legacy buses. However, at any time, only one processor can access a particular one memory; otherwise, bus contention occurs. To handle this, a bus controller with an arbiter switch limits bus access to one processor at a time. Because it is not scalable and the system efficiency is low, this system is not regarded as an NoC.

In other two-network schemes, switching network and shared memories, NoC concepts are included. The crossbar switch is the ultimate interconnecting architecture for high performance. In Figure 1.33, m vertical processors are connected to m horizontal links, whereas n horizontal memories are connected to m vertical links. At each cross section, a switch connects the junctions with control signals. In this network, every processor can access a free memory independent of other processors. Also, several processors can have access to the memory at the same time. If more than one processor tries to access the same memory, the scheduler in the crossbar should determine which one to connect to. The drawback of the crossbar switch is the number of switches, in this case, $m \times n$. Various multistage networks can be used to reduce this drawback, but it results in switch latency, the delay

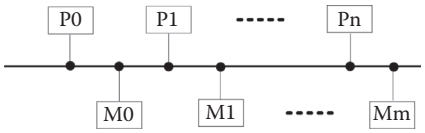


FIGURE 1.32 Shared bus-based multiprocessor system.

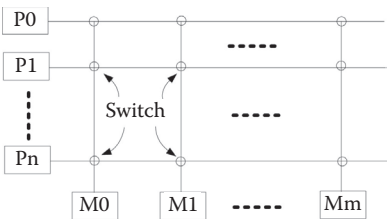


FIGURE 1.33 Crossbar switch network.

incurred by a transaction passing through multiple switch stages.

The multiport memory can be used as an interconnection network, as shown in Figure 1.34. All processors have a direct access path to every memory, and the controller inside the memory determines which processor to connect to. The complexity, previously in the crossbar, is now shifted inside the memory. The realization of memory with such complex logic and multiport is very expensive, even impractical.

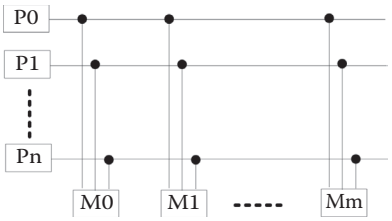


FIGURE 1.34 Multiport memory-based multiprocessor system.

As we have discussed, the interconnection network plays a critical role in MPSoC and is of paramount importance if high-performance MPSoC is to be realized. With advanced submicron silicon processes of the MPSoC, performance of the processors can be improved to more than gigahertz clock operations. However, nanometer-size technology may increase the total length of the interconnection wire on a chip, resulting in long transmission delay and higher power consumption. In addition, the distance between wires shrinks with technology, increasing coupling capacitance, and the height of the wire material increases resulting in greater fringe capacitance; the performance bottleneck comes from the interconnection rather than the processors. On top of this, the design time increases and the complexity worsens as the number of processors increases. The bus system definitely cannot provide the same performance and the NoC or point-to-point interconnection networks should be used.

So, what makes the NoC different from conventional interconnection methods? Not only the interconnection technology but two more technologies (networking and packet switching fabric technologies) also are required for NoC. Of course, more advanced interconnection technologies should be employed on the NoC; e.g., high-speed and low-power signaling, and on-chip serializer/deserializer. Switching fabric requires buffer and scheduler technologies. Networking technology includes network topology, routing algorithm, and network performance analysis. These topics will be extensively explained in Part II of this book.

1.4 LOW-POWER SOC DESIGN

As the integration scale increases, as in MPSoC, power efficiency and performance/watt become more critical metrics along with absolute performance, million instructions per second (MIPS). The low-power SoC is essential to achieve high power efficiency and performance/watt. Low-power design methodologies have been well developed and are actively employed in the design of SoC for cell phones [25,26]. Before we study the low-power NoC in detail, in this section we will review the general low-power SoC design techniques and their trends.

1.4.1 CMOS Circuit-Level Low-Power Design

The first step to low-power design is to know the sources of power dissipation. The design can then be achieved by reducing the contribution of each source by all means.

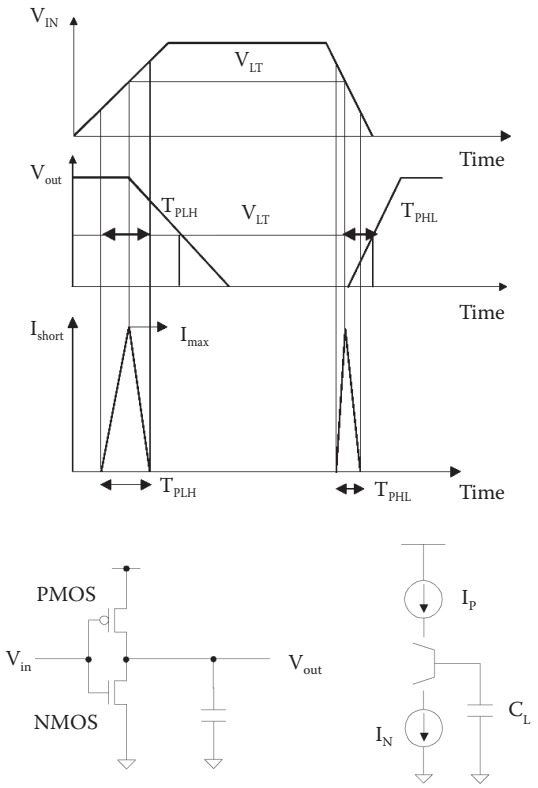


FIGURE 1.35 CMOS inverter and short-circuit current.

CMOS logic devices consume power when they are operating. There are two major sources of active power dissipation in the digital CMOS: dynamic switching power and short-circuit power (Figure 1.35). Another power dissipation source is the leakage of power that results from subthreshold current or current flowing through MOSFET when $V_{gs} = 0V$. The total power is given by the following equation:

$$P_{total} = P_{switching} + P_{short\text{-}circuit} + P_{leak} = \alpha_{0\to1} C_L V_{dd}^2 f_{CLK} + I_{sc} V_{dd} + I_{leak} V_{dd}$$

The first term denotes dynamic switching power, C_L is the loading capacitance, $f_{CLK,}$ the clock frequency, and $\alpha_{0\to1}$ is the probability that a power-consuming transition occurs (the node transition activity factor). The second term is due to the direct-path short-circuit current, I_{sc}, which arises when both the NMOS and PMOS transistors are ON at the same time. The third term is due to leakage current, I_{leak}.

Usually, the active power consumption, especially switching power $P_{switching}$, is dominant, but for 90 nm and 65 nm CMOS fabrication processes, the leakage current is almost 30% of the total power consumption. Low-power design methods are ways to decrease power dissipation by reducing the values of $\alpha_{0\to1}$, C_L, V_{dd}^2, and f_{CLK}. The available low-power techniques and their effects on the terms of the power equation are summarized in Table 1.1. The node transition activity factor is a function of

TABLE 1.1
Summary of the Low Power Techniques for SoC

Toggle count	Logic style, transition reduction coding
Load capacitance	Wire length minimization, partial activation
Voltage scaling (VS)	Small swing
	multi Vdd, dynamic VS, adaptive VS
	Multi-Vth, variable Vth (substrate bias), negative Vgs
	power shutoff, power gating
Frequency scaling (FS)	Clock gating, dynamic FS

the Boolean logic function being implemented, the logic style, circuit topologies, signal statistics, signal correlations, and the sequence of operations. It is effective in reducing the load capacitance of the gate with a high activity factor. However, most factors affecting the transition activity are determined by logic synthesis EDA tools available in modern design practices. Clever algorithms for the correlation of the data can be applied before logic synthesis to reduce toggle counts. Some examples of ways to reduce the number of toggles are shown in Figure 1.36. By rearranging the interconnection of logic gates or input-pins, the number of unnecessary transitions can be reduced.

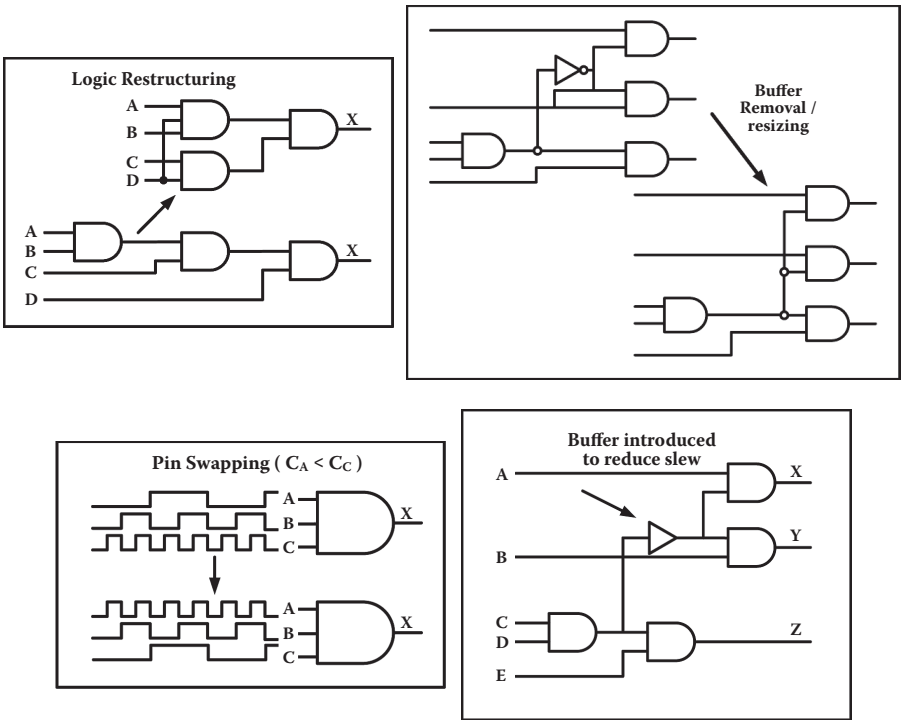


FIGURE 1.36 Circuit examples to reduce the toggles.

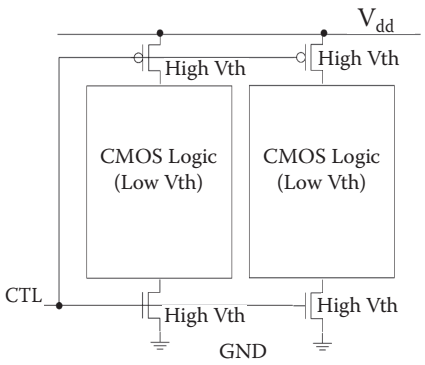


FIGURE 1.37 Multi Vth CMOS logic.

The capacitance in CMOS circuits results from two major sources: devices and interconnects. The value of the switched capacitance can be reduced by various methods in different levels: algorithm, architecture, logic, circuit, and physical layout. Low physical layout and placement, and circuit/logic levels are automatically determined once CAD tools for the design are selected. However, if we use less logic and smaller gates, we can keep the capacitance small. Partial activation of a small part of a large logic array is also effective. The large array of logic blocks is divided into multiple small blocks connected by buffers and is partially activated to reduce the value of the capacitance. For the capacitance of long interconnection, buffers are commonly used to decouple its large capacitance from the driver to reduce the capacitance load.

Voltage has an effective and direct influence on low-power operation due to its quadratic relationship with power. There are many different low-power design methods to take advantage of the voltage. Even without special circuits or technologies, a decrease in the supply voltage reduces power consumption not only in one subcircuit or block, but also in the entire chip. However, as supply voltage is lowered, circuit delays increase, resulting in reduced system performance. To compensate for the slow speed, the threshold voltage of the MOSFET is lowered to allow more current to flow at low supply voltage, $(V_{gs}–V_t)^2$. This low threshold voltage, in turn, leads to subthreshold leakage current in a standby state. In Multi Threshold Logic, the logic blocks are designed with the low threshold MOSFET and, then, those blocks are connected to the power supply and ground through the high-threshold MOSFET switches as in Figure 1.37. You may use the low-threshold-voltage MOSFET as the power switches, but in the standby state, a negative voltage is applied on the Vgs voltage of the switch transistors to put it into a deep turnoff state. In the variable threshold technique, the substrate voltage is varied from ground for the active mode to negative voltage in standby mode to increase the threshold voltage of MOSFET.

For low-power interconnection, the signal swing is reduced to decrease not only the power consumption but also the charging time of the interconnection capacitance, which will be explained in more detail with real circuits in Part II, Chapters 5 and 6.

1.4.2 Architecture-Level Low-Power Design

There are many low-power schemes that have come up from the level of RTL designs. The most common and widely used method is *clock gating*, which disables unnecessary blocks in the synchronous system. The clock is connected to the internal circuits

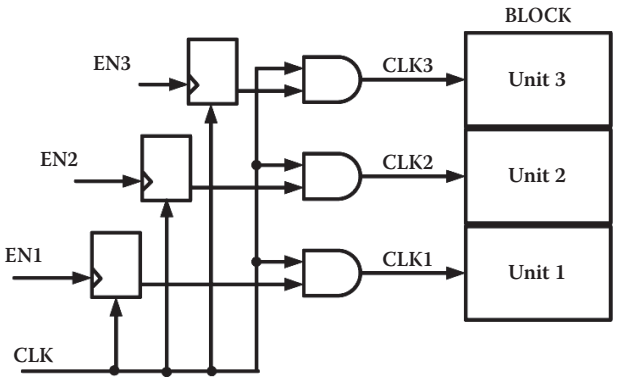


FIGURE 1.38 Clock gating.

through the AND gate, which is controlled by the *gate enable signal*. This scheme can be applied block by block to selectively control power consumption. In clock gating, when the clock is high, the *clock enable signal* goes high. In this case, the pulse of the first clock signal is not wide enough, which may lead to difficulties in circuit operation. This can be avoided by using a latch to make the control signal high only when the clock is low. Figure 1.38 shows an example of clock-gating circuits.

In the architecture level, parallelism can be utilized to reduce power consumption. For example, if you put the same functional module in parallel with the original one, you can double the throughput of the functional operation and, as a result, lower the clock frequency by half, if the throughput is to be the same as the original one. Precomputation can remove unnecessary toggles, too. Before the entire operation of the main circuit, a part of the circuit is precomputed. The internal switching activities of the main circuit are controlled by using the pre-computed results to reduce the number of toggles.

1.4.3 System-Level Low-Power Design

An SoC or subsystem has one or more major functional modes, such as operational mode, idle mode, sleep mode, and power-down mode. In the operational mode SoC operates the normal functions, and in the idle mode, the clock buffer is ON but no signal is being switched. In the sleep mode, the clock part as well as the main blocks is OFF. When the SoC is turned off with the power supply connected, it is in the power-down mode. In the system level, the low-power solutions are multisupply voltage or voltage scaling, power shut-off, adaptive voltage scaling (AVS), and dynamic voltage and frequency scaling (DVFS).

For the system-level low-power schemes, the SoC is first divided into multiple voltage and frequency domains; it then adopts DVFS, AVS, and power shut-off or power gating to control power dissipation in each domain. Figure 1.39 shows an example of a cell-phone application processor with 20 different power domains, which and reduces the leakage as well as operation currents [27]. Separate control of the power dissipation in different power/voltage domains requires a *level shifter and isolation cell*, or *micro I/O*, and a *microswitch* to supply the controlled power to

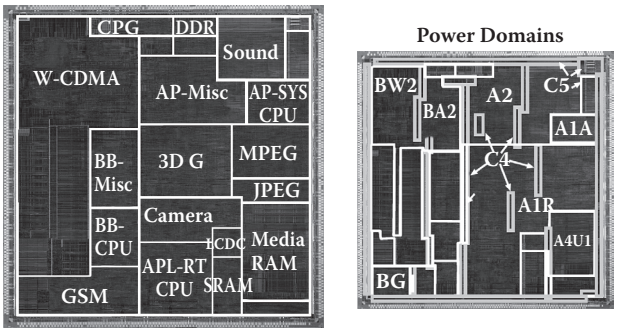


FIGURE 1.39 Cell phone processor with 20 different power domains [27].

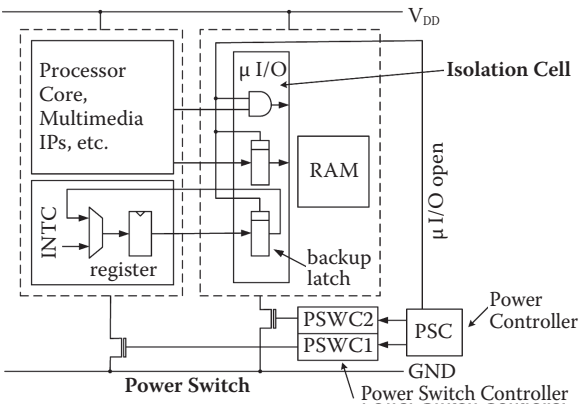


FIGURE 1.40 Power domain interface.

the different power domains, as shown in Figure 1.40. In addition, attention should be paid to *cross-domain timing closure*, which represents the variation of the signal delay time and its related troubles in circuit operation when the signal traverses over the boundary of the voltage domains, as illustrated in Figure 1.41. "Rush Current," which is the current peak when the power switch is ON, should be reduced.

Figure 1.42 shows the schematics of the isolation cell and level shifters to connect the different power domains [28]. The μ switch can be buffered to reduce the loading effect of the control circuit and has a regular physical layout to fit in the layout rows. When a large logic block is turned on, a current spike flows through the switch and may cause damage to the circuits. To alleviate the current spike, multiple switches, instead of a single large switch, are connected and turned on one by one

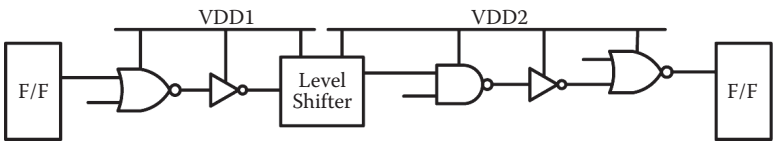


FIGURE 1.41 Delay variation due to multiple voltage.

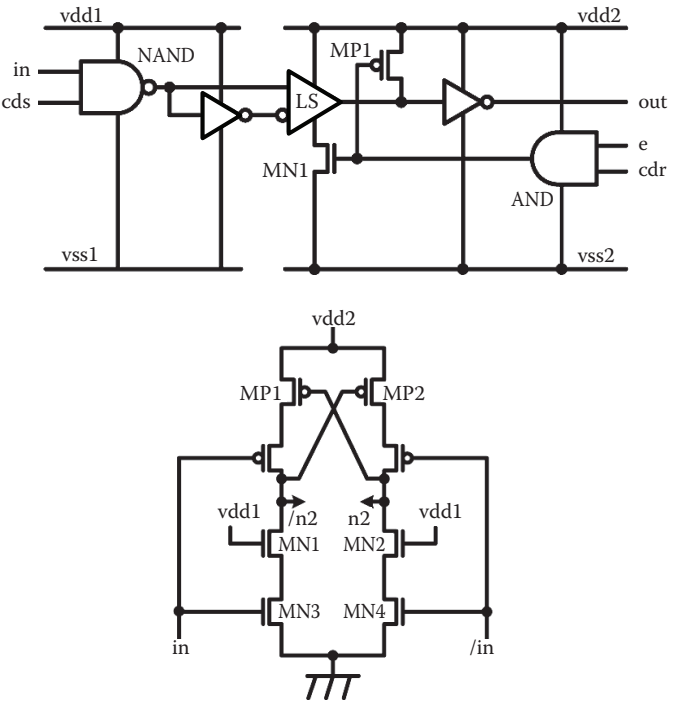


FIGURE 1.42 (Top) μI/O and (bottom) level shifter circuits for power domain interface [28].

to supply current with gradual increase to the power domain. Figure 1.43 shows an example of how to reduce peak current with multiple parallel switches.

The DVFS scheme controls the voltage and frequency together to check power dissipation, as shown in Figure 1.44; it has three independent domains [29]. It is common to scale the values of voltage and frequency in accordance with the software. The OS controls the scheduling according to the workloads of different domains. When the voltage and frequency decrease, first, the clock frequency and, then, Vdd are scaled down. In contrast, for the scale-up, Vdd is up first followed by clock-frequency.

1.4.4 Trends in Low-Power Design

There are four basic themes for low-power techniques: (i) trading off area and performance for power, (ii) adapting designs to environmental conditions or data statistics, (iii) avoiding waste, and (iv) exploiting locality [9]. It is clear that power can be traded with performance and the area can be used to recover the performance. As we have examined earlier, low voltage leads to low power consumption with slower operational speed. Parallel processing can maintain the performance at a lower voltage. Because parallel processing brings up area penalty, this can be regarded as trading area for low power. According to the variation in the environment or statistics of the input data, the operation of circuits can be dynamically changed to save power. DVFS is an example. Avoiding wastes is a very effective technique. Clock gating and power shutoff are good examples of this. In addition, charge recycling reuses

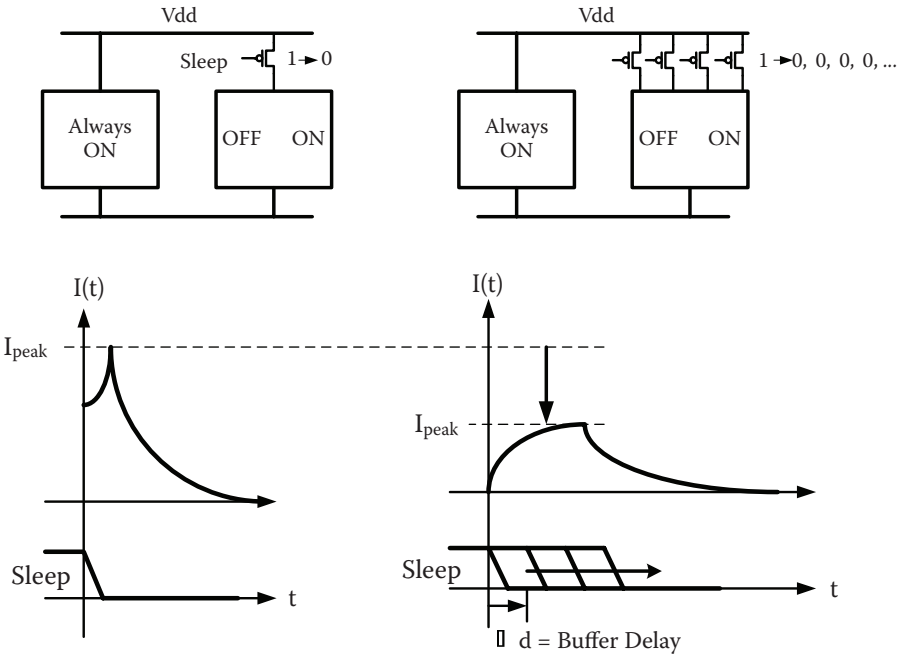


FIGURE 1.43 Rush current control with buffer chaining.

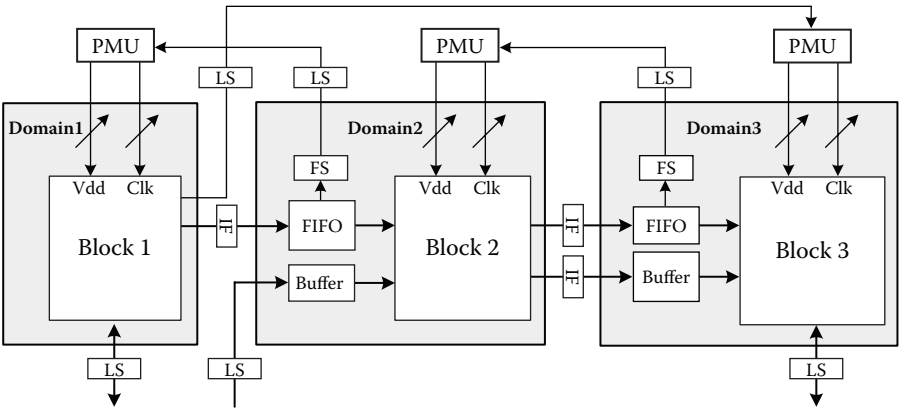


FIGURE 1.44 Three independent domains with DVFS.

the power without waste. Exploiting the locality is another technique of low-power design. A design partitioned to take advantage of the locality can minimize the cost of expensive global communications and exploit partial activation of the chip.

In this book, especially in Part II, we will explain how to apply these general guidelines of low-power techniques to the design of NoC. Of course, software contribution to low power should be taken into account, but this is left to other related books.

REFERENCES

1. Sriram Vangai, et al, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS", ISSCC Dig. Tech. Papers, pp.98-99, Feb. 2007.
2. D. Pham, et al, "The Design and Implementation of a First-Generation CELL Processor," ISSCC Dig. Tech. Papers, pp. 184-185, Feb. 2005.
3. Jorgen Stunstrup and Wayne Wolf, Hardware/Software Co-Design: Principles and Practice, Kluwer Academic Publishers, Boston, 1997.
4. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, Lee Todd, Surviving the SOC Revolution, p. 71, Kluwer Academic Publishers, Boston, 1999.
5. M. Fujita, System LSI Design Engineering, Ohmsha, Tokyo, 2006.
6. http://www.samsung.com/global/business/semiconductor/products/flash/downloads/ssd_datasheet_200710.pdf
7. Gajski, et al, Spec C: Specification Language and Methodology, Kluwer Academic Publishers, Boston, 200.
8. A. Gerstlauer, et al, SYSTEM DESIGN, A Practical Guide with SpecC, Kluwer Academic Publishers, Boston, 2001.
9. A. Gerstlauer, et al, op. cit., p. 73.
10. A. Gerstlauer, et al, op. cit, p. 82.
11. A. Gerstlauer, et al, op. cit., p. 85.
12. A. Gerstlauer, et al, op. cit., p. 109.
13. A. Gerstlauer, et al, op. cit., p. 132.
14. Henry Chang, op. cit., p. 58.
15. Grant Martin and Henry Chang, Winning the SoC Revolution, Kluwer Academic Publishers, Boston, 2003.
16. http://www.arm.com/pdfs/Networking_Solutions.pdf
17. http://www.nxp.com/acrobat_download/literature/9397/75010486.pdf
18. http://focus.ti.com/lit/ds/symlink/omap5912.pdf
19. http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf
20. http://www.idt.mdh.se/kurser/ct3410/ibm_cc_2_9/published/corecon/PlbToolkit.pdf
21. D. Wingard and A. Kurosawa, "Integration Architecture for System-on-a-Chip Design," IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, pp. 85-88, 1998.
22. David Carter, "Introducing PS2 to PC programmers," Australian Game Developers Conference, December 2002.
23. Ahmed A. Jerraya and Wayne Wolf, Multiprocessor Systems-on-Chips, Morgan Kaufmann Publishers, San Francisco, 2005.
24. David E. Culler and Jaswinder P. Singh, Parallel Computer Architecture, Morgan Kaufmann Publishers, San Francisco, 1999.
25. A. P. Chndrakasan and R. W. Broderson, Low Power Digital CMOS Design, Kluwer Academic Publishers, Boston, 1996.
26. J. M. Rabaey and M. Pedram, Low Power Design Methodologies, Kluwer Academic Publishers, Boston, 1996.
27. A. Gupta and T. Hattori, Low Power CMOS Design, Asia and South Pacific Design Automation Conference 2007 Tutorials.
28. Y. Kanno, et al, "μI/O Architecture: A Power-Aware Interconnect Circuit Design for SoC and SiP," IEICE Trans. Electron., VOLE87-C, No.4, pp.589, Apr. 2004
29. Byeong-Gyu Nam, et al, "A 52.4mW 3D Graphics Processor with 141Mvertices/s Vertex Shader and 3 Power Domains of Dynamic Voltage and Frequency Scaling," ISSCC Dig. Tech. Papers, pp.278-279, Feb. 2007.

2 System Design with Model of Computation

In this chapter, we will briefly introduce the methodology of the system-level design from the viewpoint of the designers of Network on Chip (NoC). High-level abstraction is essential for the construction of the architecture of the System on Chip (SoC) and, later, its correlation with the related software. Here, we will explain the practical methods necessary for NoC-based design rather than more rigorous and theoretical study.

2.1 SYSTEM MODELS

The design of a system is a sequence of activities to implement its functional requirements by assembling the physical components and algorithmic software. The entire design process of a system can be split into many stages; each stage has a design input and a design output. Every intermediate design activity requires inputs for simulation, analysis, or synthesis. In the top-down design, the design process starts from the system function requirements; these are the design inputs to the first design stage. Its design output is more detailed and more complicated than the system requirement and one step closer to the physical implementation. Each stage needs a model of system or model of computation (MoC) to map its design input to the design output (see Figure 2.1). The model is composed of a set of simpler subsystems and a method, or the rules for integrating them, to implement the system functions. Of course, each stage has a different model with a different level of abstraction. On the one hand, it should contain all relevant characteristics, but on the other hand it should be as simple as possible.

The topmost model on which the remaining models are based is given by the behavioral specification of the system. Usually, the behavioral specification is described using either the natural language or general system-level languages such as UML [1,2] and SDL, programming languages such as C/C++ and JAVA, specific system languages such as SpecC [3] and SystemC [4], and hardware description languages according to the abstraction levels. The model should fully define the components and their composition rules, and outer behaviors, as well as its characteristics. Different models are chosen in different phases of the design process to emphasize only those aspects of the system necessary for the design at a particular time. In the specification stage, a more abstract model is used because detailed information on the real implementation is not available, whereas at the implementation stage a more detailed model reflecting the system's structure is used. Once a model is chosen based on the characteristics of the system, it has to be refined to show the exact behavior of the system. Even if the characteristics and behavior are fully implemented, we still need to specify how that system is to be manufactured. The system

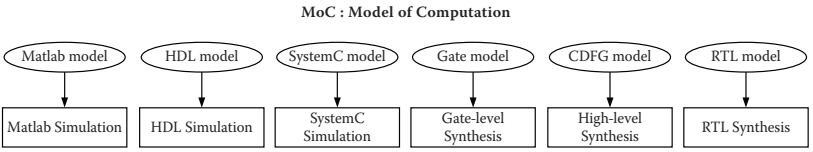


FIGURE 2.1 Model of computation.

should be defined by specifying the number and types of components as well as their connections, which is called *architecture*. The architecture is the dual of the model; the model describes how a system works, whereas the architecture describes how it can be manufactured. The design process is nothing but to find ways to transform a model into architecture.

2.1.1 Types of Models

The SoC is a complicated system and has many heterogeneous components. Managing its complexity and heterogeneity is the key issue in SoC design. The designer at the initial stage should determine how to view the target system and describe its behaviors in his own terms. That is, they have to choose the model of the target system, or MoC. The use of well-developed models with high levels of abstraction and detail can help the SoC designer understand and manage the chip with ease, and less error.

Models or systems can be categorized based on various criteria [5]. Here, we will classify models based on two aspects: communication and behavior. Figure 2.2 shows the relationship. The interface is just a passage to and from an external entity or system. More significantly, it is important to determine how models react to external systems. We will focus on this in Subsection 2.1.1.1. Internal behavior can be implemented in various ways. For example, we can design a state machine for the behavior of a system. Alternatively, we can use different semantics, such as continuous functions, discrete-time models, or combinational logic.

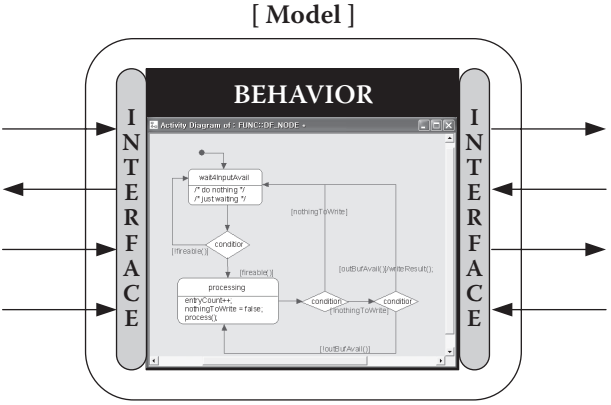


FIGURE 2.2 Models—communication and behavior.

TABLE 2.1
Synchronous/Asynchronous Communication

Synchronous communication	Event sending time = event receiving time	No buffering mechanism
Asynchronous communication	Event sending time (event receiving time)	Buffering mechanism

2.1.1.1 Communication

Firstly, we can divide communication into two types: synchronous and asynchronous [5]. With the synchronous type, the communication between the sender and receiver occurs at the same time. That is, the process that receives a message should get it at the same time it is dispatched by the sending process. Therefore, zero propagation delay and no buffering mechanism are assumed. Compared to the synchronous communication, the sending time and receiving time of a message can be different in the case of asynchronous communication. That is, communication actions can occur at different times through a buffering mechanism. See Table 2.1.

Synchronous communication consists of a sequence of macrosteps, and every step contains processes from each of the components as shown in Figure 2.3a. It is also commonly called blocking communication or rendezvous. In blocking communication, the sending and receiving processes wait for each other until both are ready to exchange the message. The message is then exchanged as a common step, after which the processes are again independent. The asynchronous computation does not have any timing information coordinating all the participating components. It is called nonblocking communication or buffered communication. In a nonblocking communication, the sending process delivers the message when it is ready and goes on, irrespective of when the receiving process chooses to accept the message. If it tries to receive the message before it is sent, the receiving process waits. The nonblocking message passing through requires a buffering mechanism for holding messages sent, but not yet received.

Table 2.2 shows the read/write–blocking/nonblocking relationship. For the read operation, nonblocking is not feasible. *Nonblocking read* implies that the receiving actor can fetch the required data not ready at the sending actor. That is, *read* is always blocking. For the write operation, *blocked write* implies that the sending actor should wait for the receiving actor to be ready. *Nonblocked write* allows one to write and forget. That is, there is a buffer between the sender and the receiver.

Table 2.3 summarizes write/read–synchronous/asynchronous communication. Synchronous communication utilizes blocking write and blocking read. On the contrary, asynchronous communication uses nonblocking write and blocking read for infinite buffer.

2.1.1.2 Behavior: Time and State Space

Most natural phenomenon, such as diffusion of electron, falling object, Brownian motion, and so on, is in the form of continuous state/continuous state, although some of them have too complex mechanisms to model. Therefore, it will be desirable if every dynamic behavior can be modeled in continuous time/continuous state without

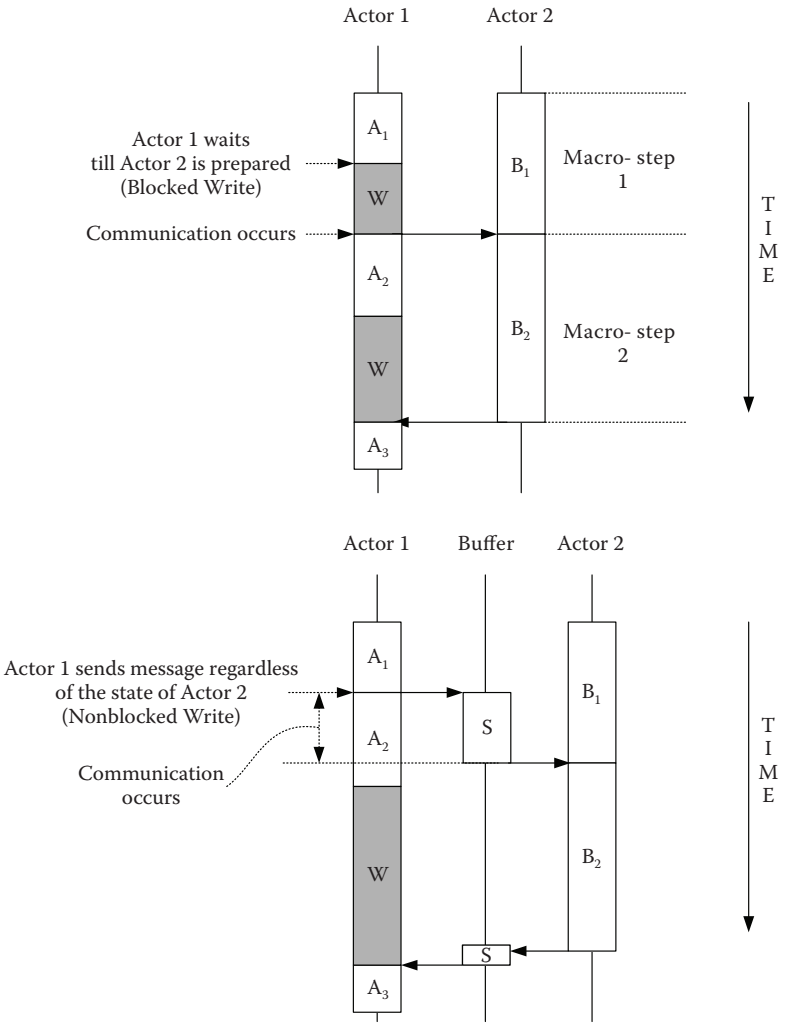


FIGURE 2.3 Timing relationship between synchronous/asynchronous communication.

TABLE 2.2
Relationship between Read/Write and Blocking/Nonblocking

	Blocking	Nonblocking
Read	Wait for data available	N.A.
Write	Wait for recipient to be ready	Write and forget

TABLE 2.3
Blocking/Nonblocking Write versus Blocking Read

	Blocking Read
Blocking write	Synchronous communication
Nonblocking write	Asynchronous communication

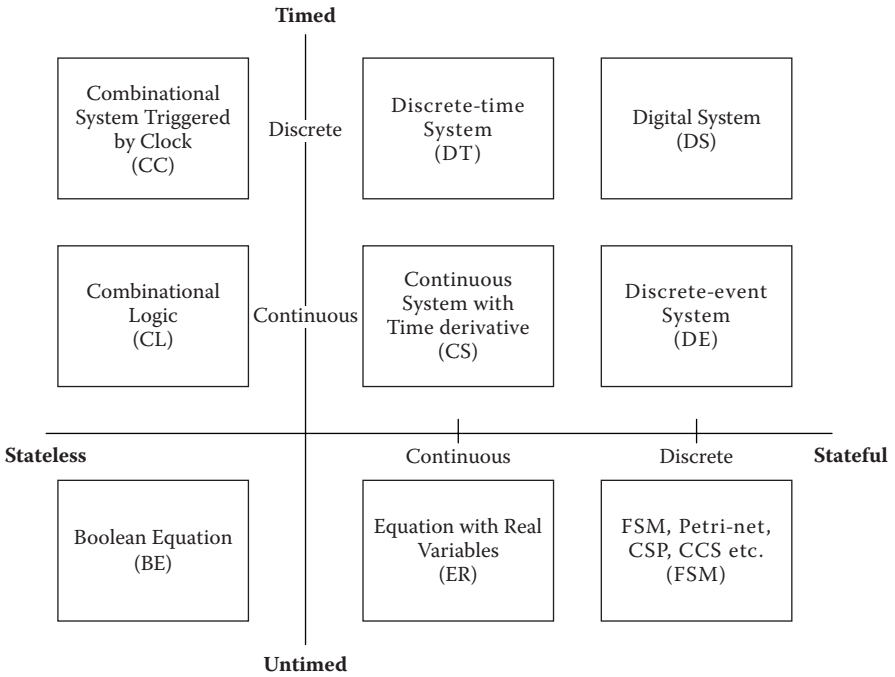


FIGURE 2.4 Time–state space representation.

error. However, it is often utterly impossible to construct continuous-state/time models for certain real systems. Besides, it may be unnecessary to model every behavior in detail. It is here that the abstraction level comes into play, when different domains of state/time are aggressively utilized.

A system consists of behavior and structure. The iteration of, first, defining the structure, second, decomposing it, and, then, modeling behavior of each component is the top-down approach. There are two important ingredients for behavior description: time and state. Figure 2.4 extends the time–state classification [5,6] to cover the untimed system and stateless system.

First, for the time aspect, there are three possible spaces: continuous-time space, discrete-time space, and untimed space. The first two can be called timed space. An untimed system has, of course, no timing information specified for the operation or task. Therefore, an untimed system is adequate to represent an algorithm, or to investigate an untimed performance index such as an execution count or the traffic between models. As mentioned before, there are two types of time for timed systems: continuous (CS) and discrete (DE). Although the DE and CS systems share the common property of time, they have different views. In the case of CS, the state in every time instance is important because it is the base to compute the states after the instance. On the contrary, DE specifies only the important instances of time, such as time-out. That the time domain is continuous for DE means that an event can happen at any instance of time. That the time domain is discrete implies that there is a synchronizer such as a clock or tick event, and every activity behaves based on it. As

will be explained in Subsection 2.2.1, the instance of evaluating every model is also fixed, and in turn it gets easy to model or simulate the discrete-time model.

Secondly, we can divide a behavior model based on the state aspect. Basically, a stateless model is combinational, i.e., output is determined by a combination of inputs only. It can be functional if no timing is related with wire or logic. It can also be used to analyze the timing property with a continuous-timed combinational system. Stateful models possess special media to store data, which last up to some specific point of time, such as buffer, latch, or flip-flop. The state also can be continuous or discrete. A system with discrete-state needs some mechanism to change its own state. It is a combination of input/state/clock for DS, and combination of input/state/event for DE.

2.1.2 Models of Computation

2.1.2.1 Finite State Machine and Its Variants

The finite state machine (FSM) is useful in describing control mechanisms. Basically, FSM can be defined by five parameters: current state, input, output, next-state determination function, and output determination function (see Figure 2.5). Output determination functions are divided into Moore-type (state based) and Mealy-type (input based). The typical example of the Moore-type is the counter logic, in which state itself is used as its output.

There is no coupling scheme for original FSM models. Composition is one way of coupling more than one FSM, and is useful for analysis, but has serious drawbacks such as state space explosion. A **communicating FSM** [7] uses the channel between traditional FSM semantics. Thus, two models in a communicating FSM can synchronize their operation using common events shared by a channel. So it is possible to test interoperability between two models.

Although FSM is simple and straightforward, it is very difficult to model real-life control behavior because of the complexity [8]. D. Harel introduced STATECHART, which enables one to manage the complexity of the model with features such as hierarchy, concurrency, history, etc. First, the complexity of the state machine can be managed using **hierarchical FSM**. Hierarchical states have another state machine inside it. This enables users to group state machines, which can be thought of as a unit of control.

The other way of managing complexity in STATECHART is the concept of **concurrent state machine**. The FSM specifies only OR-state, that is, the machine can only reside in one of the states specified in the machine. However, concurrent state machine can specify AND-state. In addition to complexity management, it also specifies the communication methods by event synchronization. Figure 2.6 shows the concurrent state machine. It uses AND-state concept. The two state machines in AND-states are concurrent; that is, they should be combined to determine the total state of the system. State1 generates the event with transition to state4, whereas state2 transits to state3 due to the event e. This is a course of synchronization. The FSM can be hierarchical and concurrent at the same time, and therefore, can also be categorized as hierarchical, concurrent FSM (HCFSM).

FIGURE 2.5 Moore type FSM versus Mealy type FSM.

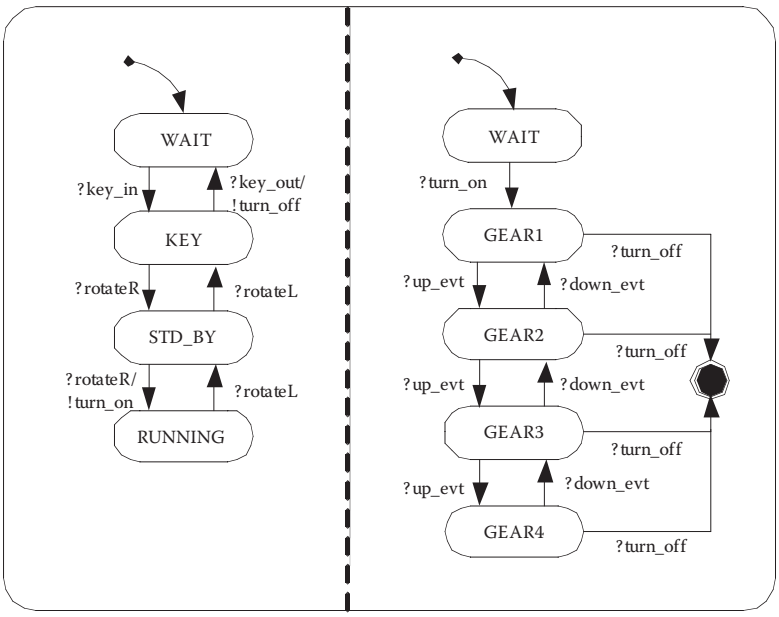


FIGURE 2.6 Concurrent state machine.

The last variant of FSM to be investigated here is the finite state machine with datapath, or FSMD [9]. This type of FSM declares global data that all the states can treat in the machine, so that these data can be seen and modified.

Although timing information can be coupled with state and transition, FSM is basically an untimed model. There are extensions of FSM to hold timed characteristics. However, these kinds of semantics belong to the scope of discrete event systems.

2.1.2.2 Petri Net

A Petri net is also useful in describing the control structure of a system. However, the Petri net focuses on events, especially firing conditions. It is similar to a graph, the place like a node, and the transition, an edge. For a transition to fire, its input condition should be met. For example, t1 transition can fire if there are tokens in places p1 and p2. There is the notion of fork and join with a Petri net. In other words, multiple places can be active due to the transition with input of 1 and output of more than 1. This is a useful notion, but a designer using this should be careful lest the model should fall into deadlock due to the fork-join condition. Figure 2.7 shows a simple Petri-net model for the dining philosopher problem, with only two philosophers. Figure 2.7a shows the deadlock-free model and Figure 2.7b shows the model in which a deadlock can happen. In Figure 2.7a, places p3 and p4 are forks; p1 and p2 represent one philosopher, and p5 and p6 the other. Transitions t2 and t3 are fireable. The modeler does not know which transition will fire; i.e., they are non-deterministic. Figure 2.7b, which refines Figure 2.7a, is more complicated. Here, p5 and p6 are forks. P1, p2, p3, and p4 represent one philosopher; p7, p8, p9, and p10 represent the

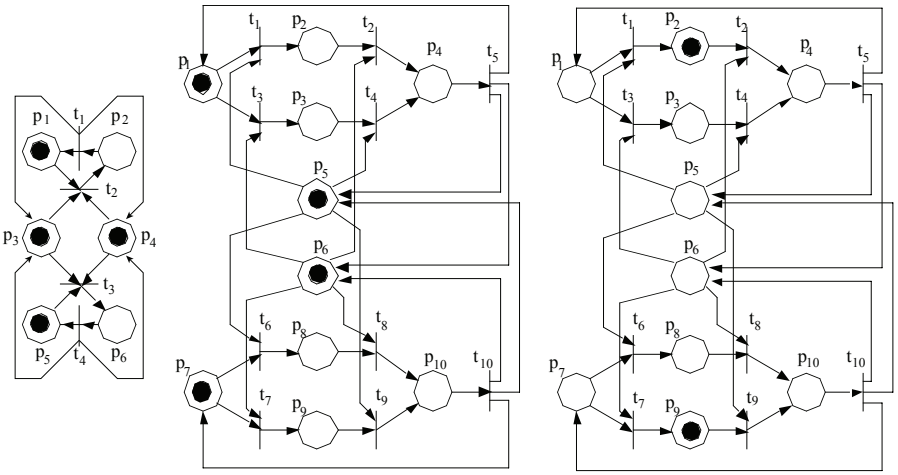


FIGURE 2.7 Petri-net modeling of dining philosophers.

other. t1, t3, t6 and t7 can fire. However, t1 and t3 cannot fire simultaneously because one transition will remove the token at p1; therefore, they are exclusive. This applies to t6 and t7, too. We can simulate the model. Each of the following transition strings is legitimate.

$$(t1 \rightarrow t2 \rightarrow t5)\ (t3 \rightarrow t4 \rightarrow t5)\ (t6 \rightarrow t8 \rightarrow t10)\ (t7 \rightarrow t9 \rightarrow t10)$$

However, this model can fall into a deadlock, as shown in Figure 2.7c, a situation in which each philosopher has put one fork. Because t1 and t7 are fireable and not exclusive, they have fired. However, the next transitions t2 and t9 cannot fire forever because the required token is taken by the other one.

Petri-net simulation can be performed with matrix calculation. Furthermore, Petri-net and FSM models are equivalent. That is, a model in FSM can be translated to one in Petri net. This can be verified by proving that there exist two procedures, one to translate the FSM model to the Petri-net model and the other vice versa.

As with FSM, Petri net is also untimed. There are variants of Petri net, so that it can cover timed behavior—timed Petri net. This relates the timing property to transition in such a way that a specified time should elapse when the transition fires. In most cases, the unit of time is an integer, but it can be extended to a real number. A marked Petri net allows one to model multiplicity for transition from the same source using a token.

Another important variant of Petri net is the Signal Transition Graph (STG). STG is a Petri net whose transition is interpreted as a signal transition, i.e., one transition to turn on or turn off a specific signal. This formalism has been used widely for asynchronous circuit design/verification/synthesis, or an interface synthesis algorithm.

2.1.2.3 Transaction-Level Modeling

Transaction-level modeling (TLM) is a relatively recent concept in the domain of system design, which is devised for easy modeling and fast simulation. Actually,

the definition of TLM is not agreed upon by academics and companies. Open SystemC Initiative (OSCI) and Open Core Protocol International Partnership (OCP-IP) are collaborating in defining TLM and designing a library for TLM modeling and simulation. As a result of such efforts, a standard for TLM 1.0 was adopted in June 2005, with a library, and documents can be found at SystemC homepage. Further, a draft for TLM 2.0 has been submitted, was released in November 2006, and is under review with a proposal. Here in this book, we will focus on TLM based on SystemC. The following summarized characteristics of TLM [10] are the key to its fast simulation:

- Communication is per transaction, and transaction is implemented with function call.
- High-level data types are used instead of low level, in streams of four values, such as 1/0/X/Z.
- Data transfer can be done by passing the pointer to the data; it is efficient for burst transactions, in particular.
- Dynamic sensitivity is utilized lest idle process should be activated.

There have been efforts to define TLM formally. However, users tend to emphasize different advantages of TLM, and it is therefore defined in various ways. TLM is useful because we can construct models *per transaction* using *function calls*. As indicated in a study [11], three key concepts of TLM 1.0 are interface, blocking versus non-blocking, and bidirectional versus unidirectional. The basic definitions are as follows:

- Interface: A contract between sending process and receiving process.
- Blocking: The process can wait for some events to happen with this interface.
- Nonblocking: The process should run the entire body with each execution.
- Unidirectional: The data are transferred in only one direction.
- Bidirectional: The data can be transferred in both directions.

Table 2.4 shows the blocking/nonblocking and uni-/bidirectional combination.

Core interfaces of TLM 1.0 are described with class diagrams in Figures 2.8 and 2.9.

Now we will see how transaction-level models can actually be incorporated with system design. First, let us classify the models into two types with respective roles, initiators, and targets connected through channels. For example, Initiator and channel in Figure 2.10 have the contract *read_if* in common, which is the interface. Then, Initiator can access the channel through functions declared at the *read_if* interface. Of course, the channel should have implemented the functions declared at the *read_if* interface. Then, the function called from Initiator will actually have been implemented at the channel. In other words, the interface is implemented at the channel. Therefore, it is often said that the channel implements the interface.

Every SystemC channel should inherit either from *sc_prim_channel* or *sc_channel*. A channel that inherits from *sc_prim_channel* is called a primitive channel, and one that inherits from *sc_channel* is called a hierarchical channel. The biggest differ-

TABLE 2.4
Classification of TLM Interfaces

	Unidirectional		Bidirectional
Blocking	tlm_blocking_get_if		
	tlm_blocking_put_if		tlm_transport_if
	tlm_blocking_peek_if	tlm_get_if	
Nonblocking	tlm_nonblocking_get_if	tlm_put_if	
	tlm_nonblockin_put_if	tlm_peek_if	
	tlm_nonblocking_peek_if		

ence between them is that a hierarchical channel can have its own processes such as sc_module (actually, sc_channel is type-defined with sc_module; therefore, the SystemC module and hierarchical channel are identical). On the contrary, the primitive channel does not have a process; it only depends on request_update/update semantics.

The scheme presented in Figure 2.10 is one way of constructing a system, but it is not the only way to do the job. Instead of constructing a channel class independently, the target can act as a channel. That is, if only the target class inherits from the interface, it can act as a channel. The advantage of this method is that we can obtain much faster simulation speed, because Initiator can call the functions directly implemented at the target, not the channel. This method relieves the target from building another thread, which, in turn, eases the simulation kernel from thread/context switching.

Basically, the overhead of scheduling events are heavy for discrete-event simulation or cycle-based simulation. By eliminating every detail of event scheduling and events buffer management, TLM can obtain fast simulation. Early development and fast simulation provide the software engineer a base platform for software development. From the modeling viewpoint, it gets easier because a modeler has only to identify all the transactions for communication design, and model each of them one by one.

Although TLM itself is a high-level modeling technique, there are abstraction layers within it as defined by OCP-IP [12,13]. TLM 2.5 describes the TLM abstraction layers along with Register Transfer Level (RTL), which we call TLM0 in this chapter. The TL3 layer is the most abstract level, which covers untimed and packet-level messages. By refining the timing domain to an estimated timing and packets to burst of words, TL2 model is obtained. With this model, software development can start. By further refining the timing domain to cycle accurate, we get the TL1 model. This gives the real cycle-accurate model, which enables exact timing analysis. The last one shown in the table, although not TLM, is RTL, which indicates the relationship between TLM and RTL. Usually, RTL is the most abstract level for implementation, because we can connect the model to the synthesis tools.

2.1.2.4 Dataflow Graph and Its Variants

Since the invention of dataflow graph (DFG), it has been used extensively in the design representation/simulation/analysis field [12–15]. The DFG is especially appropriate

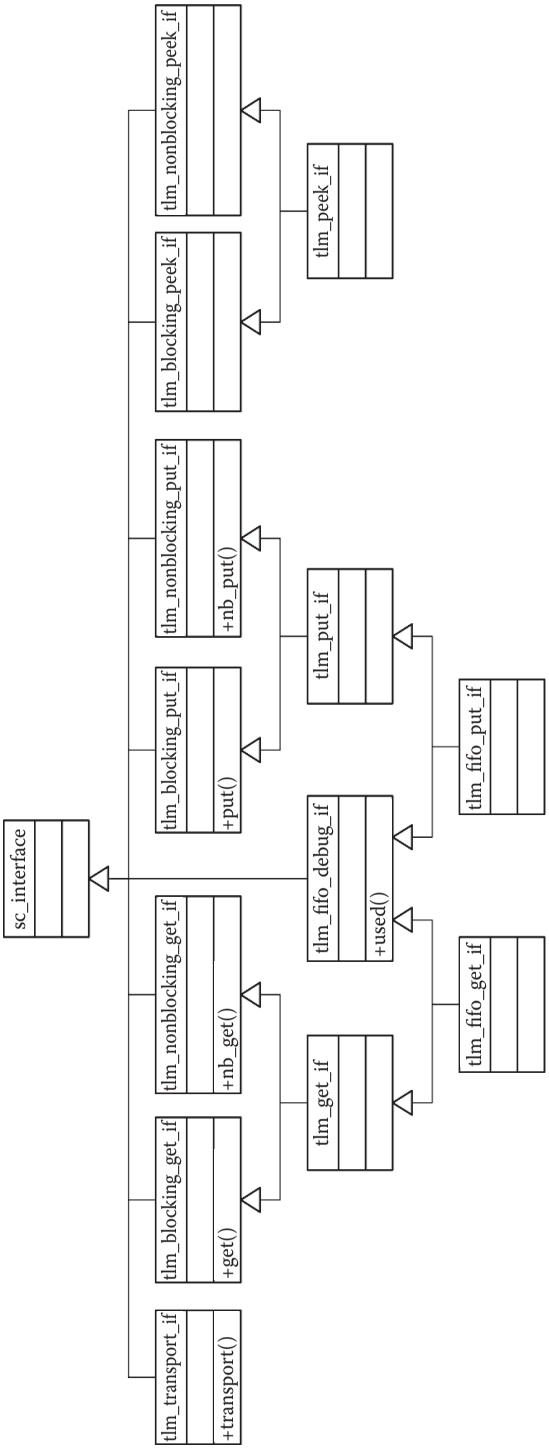


FIGURE 2.8 Class diagram of TLM 1.0 interfaces (I).

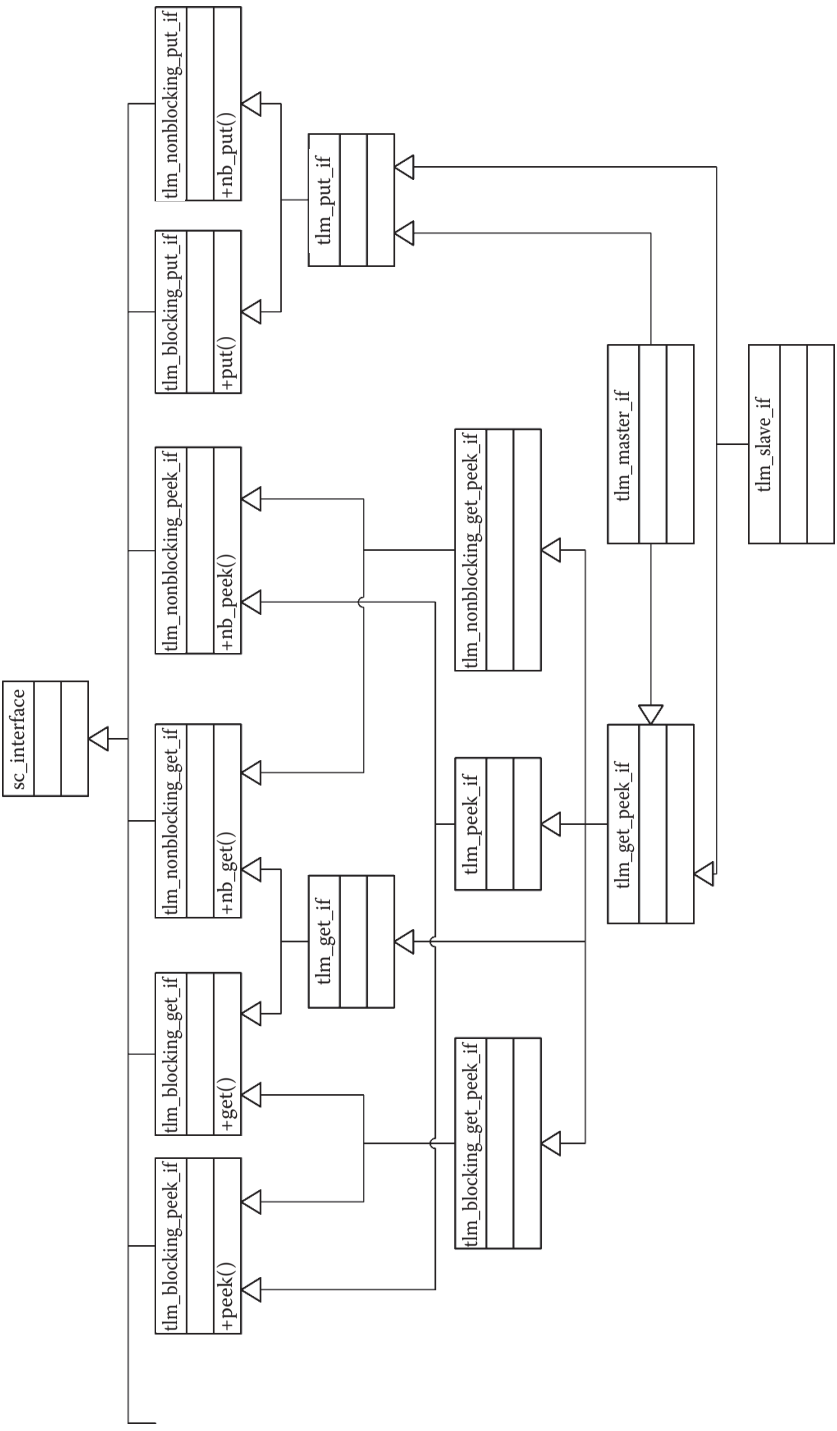


FIGURE 2.9 Class diagram of TLM 1.0 interfaces (II).

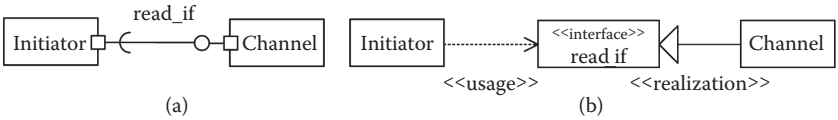


FIGURE 2.10 Role of interface for TLM.

TABLE 2.5
Abstraction Layer of TLM

		Timing Data	Description
TLM	TL3	Untimed	Functional specification (better if executable)
	Message Layer	Packets	Untimed architecture exploration
	TL2	Estimated Timing	To analyze/model SoC architecture
	Transaction Layer	Burst of Words	Early SW development
	TL1	Cycle-accurate	Cycle-true analysis of target model
	Transfer Layer	Word	Detailed SW development
RTL	TL0	Cycle-accurate	Synthesis
	Implementation	Bit/Bitvector	

for the description of a pipelined system in which any two enabled operations can be executed sequentially or concurrently. DFG is a directed graph whose vertex set is in 1:1 correspondence with the set of tasks (or operations). The directed edge set is in correspondence with the transfer of data from one operation to another.

DFG is generally formalized as a graph, as in Figure 2.11. We can classify the semantics of DFG based on those with (i) capacity located at each edge and (ii) synchronization methods. When the capacity between edges is infinite, it is called Kahn Process Network (KPN) [16]. The introduction of infinite FIFO at the edge relieves the modeler or tool developer from worry about schedule problems. Its scheduling strategy is so simple. It stores data to the edge (nonblocking write) and fetches data, if available, from the input edge (blocked read). That is, reading is blocking, which implies that the receiving process should wait till there are enough data at the input buffer. In case of general dataflow with finite buffer, writing will also be blocking. That is, the writing process should wait till there is enough space to write at the output buffer.

The other type is the synchronous DFG (SDFG) [17]. It synchronizes the communicating process with their producing and requiring tokens. For example, p1

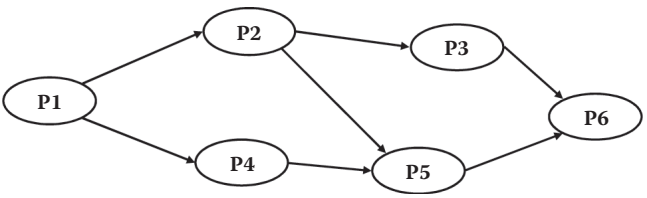


FIGURE 2.11 Dataflow Graph: simple example.

in Figure 2.12 produces two tokens, whereas p2 requires four tokens to operate. So p1 should execute the operation twice for p2 to do it once. To invoke execution, the initial token can be located in any of the edges, as is required to make the graph run.

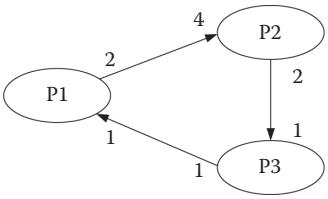


FIGURE 2.12 Synchronous Dataflow Graph.

DFGs are excellent in representing computations described by complex algorithms, but not suitable for depicting the control part, as is the case with the C language. They, as well as their variations, are commonly used to describe DSP components and systems. The control flow information related to branching (or conditional) and iteration (or loop) can be represented by Control DFG (CDFG) by introducing special vertices, branching vertices. CDFG has acted as good input specification for high-level synthesis [18]. Figure 2.13 shows the CDFG representation corresponding to the C code.

2.1.2.5 Process Algebra-Based Semantics

Process algebra is the theoretical background for a family of semantics that specially aims to specify concurrent processes [19,20]. Often, it refers to the family of concurrent semantics. Semantics that corresponds to this family include CCS (Calculus of Communicating Systems) [21], CSP (Communicating Sequential Processes) [22], LOTOS (Language Of Temporal Ordering Specification) [23], although there are subtle difference in semantics and syntax. There are many texts that compare these semantics [24]. Therefore, we will not consider them here. Instead, we will briefly introduce semantics, which is useful for designing systems, especially asynchronous circuit.

There are several kinds of semantics based on process algebra, such as CCS and CSP. The common word for semantics is "communication," possibly considering concurrency. All these semantics are based on atomic events (actions), which are inseparable. With the event (action), a process (agent) is modeled using various operators such as prefix, choice, restriction, composition, renaming, etc. The communication happens through composition, which is basically the synchronization of

```
if ( x >= c)
  x = x  * 1;
else
  x = x – 1;
```

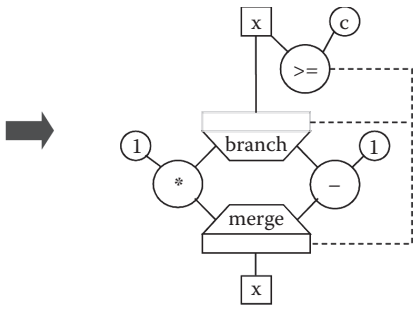


FIGURE 2.13 Example of Control Dataflow Graph (CDFG).

events between two or more processes (agents). This synchronization happens in a rendezvous in a possibly guarded fashion, which implies all the following:

- There is no buffer in the communication lines.
- Sending and receiving actions happen simultaneously.
- Blocked message read.

The dining philosopher example is shown in Figure 2.14. FORK, LFORK, RFORK, DINER, and EATING_DINER are process names and rightGet, rightPut, leftGet, and leftPut are names of events (or actions). The fork can be used by either the left or the right dining philosopher. If it is used by the left diner, it should also be released by the left one, and so, too, by the right diner. The diner should get both forks to eat; therefore, the DINER process specifies this. The person can raise the forks in any order, left after right or right after left. After eating, the diner should put back both the forks. This example relates to CSP. This kind of semantics is very useful in specifying clockless behavior, because asynchronous circuit should respond to the input immediately.

2.1.3 Summary

In practical applications, only a few types of MoCs, summarized in the following text, are commonly used [25]:

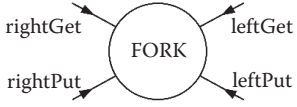
Synchronous/reactive: In a synchronous MoC the components are processes. They communicate in atomic instantaneous actions called *rendezvous*. The process reaching the point of communication has to first wait until the other side has also reached its point of communication. There is no risk of overflows, but the performance may suffer.

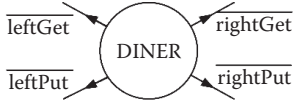
Dataflow process network: One of the asynchronous message-passing models, which communicate processes by sending messages through message-buffering channels. The nodes represent computations, and the arcs represent totally ordered sequences of events (or tokens).

A. Jerraya et al. have proposed a more simple classification based on concurrency and synchronization concepts [26]. Concurrency is based on the granularity of computation and sequencing of operations. Parallel computation can be performed at the bit (e.g., bit slicing), operation (e.g., multiple datapaths with multiple functional units), and process or processor (e.g., distributed multiprocessor model) levels. Sequencing of operations can be expressed using control-oriented and data-oriented models. In control-oriented concurrency, a specification is defined by the execution order of the element of specification (e.g., concurrent FSM). In the data-oriented concurrency, the execution order of the operations is determined by the data dependency (e.g., dataflow graph).

Based on the concepts of concurrency and synchronization, four categories of models can be obtained: synchronous (control- or data-driven) and asynchronous (again, control- or data-driven).

Other more practical representations have appeared in the literature as introduced previously: control, data, and control dataflow systems. Control systems are





FORK = (leftGet.LFORK + rightGet.RFORK)

LFORK = leftPut.FORK

RFORK = rightPut.FORK

DINER = (leftGet.rightGet + rightGet.leftGet).EATING_DINER

EATING_DINER = (leftPut.rightPut + rightPut.leftPut).DINER

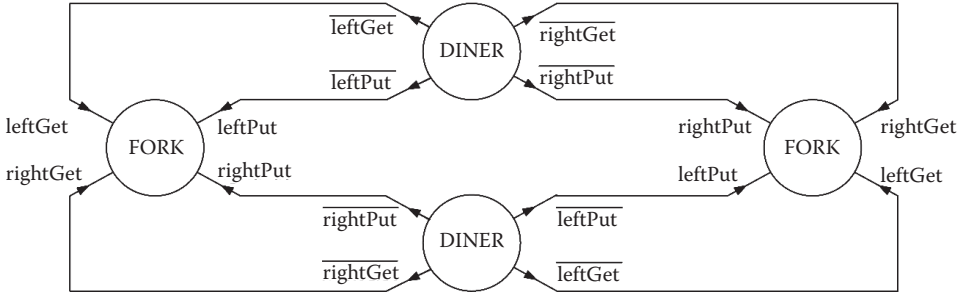


FIGURE 2.14 Dining philosophers: Example of Communicating Sequential Processes (CSP).

usually depicted with FSM because their states and transitions between states can describe the behaviors of systems such as state diagrams. The other is for data-intensive applications such as mathematical computation in which DFG is frequently used. D. Gajski [27] has summarized FSM, DFG, and their variations very clearly and concisely.

Since most real systems require both control and computation, the features of FSM and DFG should be combined. This is realized by FSM with Datapath (FSMD). It divides time into equal time intervals, called states, and allocates one or more states to each node in the DFG. That is, the FSM part of the FSMD defines the control steps or clock cycles of FSMD and, in each control step, the DFG part performs its operations under the control of FSM. The FSMD is a typical model used in behavioral synthesis. However, FSM and FSMD do not support explicit concurrency and hierarchy, which may bring up an explosion in the number of states, making it incomprehensible.

Describing more complicated control machines requires means for the expression of hierarchy and concurrency. The HCFSM is an extension of the FSM, in which each state can be decomposed into a set of substates to model hierarchy and those substates can have concurrency by parallel execution and communication through global variables. HCFSM transitions can be classified into two types, structured or unstructured. The structured form allows transitions only between two states on the same level of hierarchy, whereas the unstructured type enables cross-hierarchy transitions. The statecharts and state diagrams of UML are examples of HCFSM.

2.2 VALIDATION AND VERIFICATION

Designing embedded systems requires important activities such as verification, validation, and synthesis. The relationship between the three activities is shown in Figure 2.15, centering around models. We will focus on validation and verification here and treat synthesis in Chapter 3. Validation and verification problems are very important issues.

In short, validation is the question "did we build the right model?" and verification, "did we build the model right?" A valid model implies that it has neither semantic error such as deadlock nor mismatch with validated model at a higher

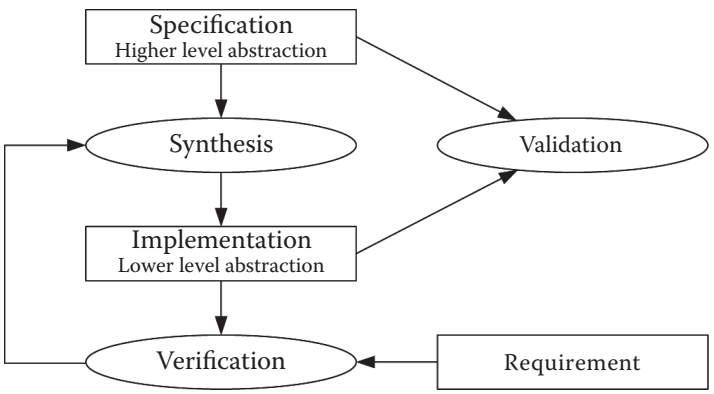


FIGURE 2.15 Verification and validation.

abstraction level. A verified model implies that the model yields expected/required behavior and performance.

There are many ways to verify and validate models. The most popular way of verification and validation is simulation, in which we mimic the system under design in software languages such as C/C++, JAVA, or Matlab and compile a model to get an executable specification. By running the resulting simulator, we can check the validity of the model, obtain the performance index, and get the ability to strobe any place at any time. Simulation enables us to fully access the internal structure of the system under design. However, there is a severe drawback: simulation speed. To tackle this problem, emulation is introduced—which exploits reconfigurable logic such as FPGA for speedup. However, it has a limitation in accessibility to internal logic. We will investigate this in the following text.

2.2.1 Simulation

There are two important responsibilities that every simulator has to perform: message routing and time management. Message routing implies that a message (or event) generated by one component should be relayed to the relevant one. Time management is the rule to adjust simulation time. How to deal with time is likely to determine the simulation kernel. There can be three possible ways of dealing with time:

- Discrete event simulation:
  - Discrete event simulator or classical HDL simulator manages event buffers, in which events are sorted in increasing order for triggering time.
  - Whenever any event happens, the kernel should insert it to the events buffer and sort with the time stamp.
  - This overhead of managing events buffer can be heavy.
- Cycle-based simulation:
  - This method scans every time instance in which there can be events, whether there is actually an event or not.
  - If relatively many components should be scheduled in each time instance, it can be more efficient than general event-driven simulation.
- Calculation method:
  - Continuous time system calculates each state value based on time derivative.
- Transaction-level simulation:
  - Communication is separated from computation, and simulation model is made per transaction.
  - Communication reduces to function call and wait for delay specified/computed.

Besides the simulation method listed previously, active research is under way on simulation speedup for a specific domain, such as compiled simulation or simulation reuse for processor evaluation. However, these techniques are efforts for simulation speedup, which is already performed in the simulation method previously described. Therefore, we will not cover such methods here.

Each simulation/analysis is well defined, and research on matters such as speedup is actively pursued. However, a system may consist of different models of computation, or even for same model of computation different languages can be assembled. Therefore, coverification will be explained. Transaction-level simulation has same semantics with discrete-event simulation in that it uses events to estimate the delay or wait for a specified event, using dynamic sensitivity.

Although transaction-level simulation is being studied hard nowadays, it does not have semantics different from discrete-event simulation.

2.2.1.1 Discrete-Event Simulation

As briefly explained previously, discrete-event simulation manages the events buffer for time-stamped events, to know what event should be processed next. An HDL (Hardware Description Language) simulator uses the term "events wheel" for exactly the same mechanism, which is depicted in Figure 2.16. The event wheel has an initial point, which indicates the head of the wheel. From the head of the wheel, events are stored at each slot sorted with next schedule time. Then, simulation kernel keeps on watching the head. If there is no event, simulation stops. Otherwise, it processes the event at the head and advances the simulation time by rotating the wheel. If new events are generated from the previous event processing, it should be inserted into the appropriate slot at the event wheel. Thus, discrete-event simulation is an iteration of execution and rotation of the wheel.

A more general and formal algorithm for discrete-event simulation can be found in the literature cited [6].

2.2.1.2 Cycle-Based Simulation

Discrete-event simulation is general enough to cover any system with discrete events, with time resolution being real. However, there are two major drawbacks. One is the memory inefficiency resulting from event-buffer management, and the other, simulation speed resulting from buffer management and event routing. To reduce these overheads, cycle-based simulation is devised. Cycle-based simulation does not treat time as real. Instead, it treats time as an integer, or tick event.

This method is available for a system that operates on a global synchronization event, such as a clock. Instead of dealing with every event, it is interested only in the clock event.

```
while !stop begin

      do_model1();

      do_model2();

      do_model3();

      eval_delayed_assign();
```

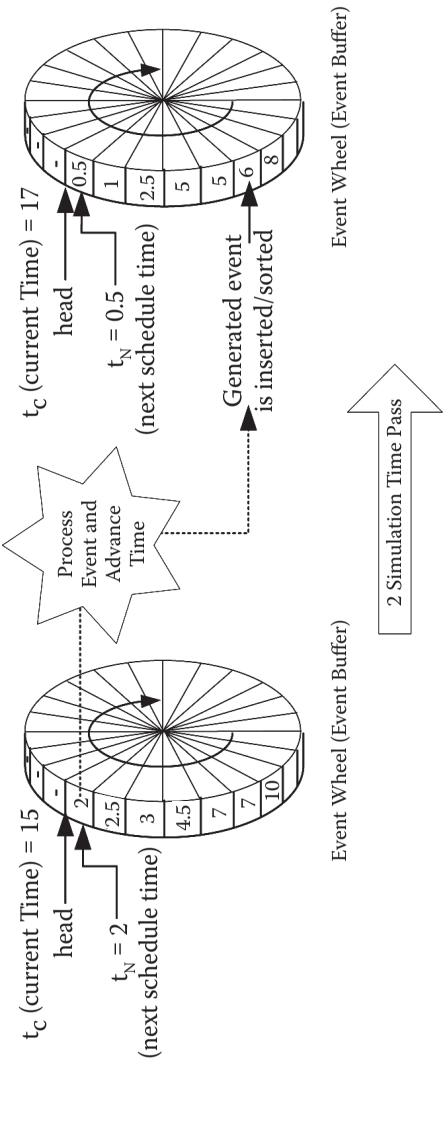


FIGURE 2.16 Time-wheel semantics for discrete-event simulation.

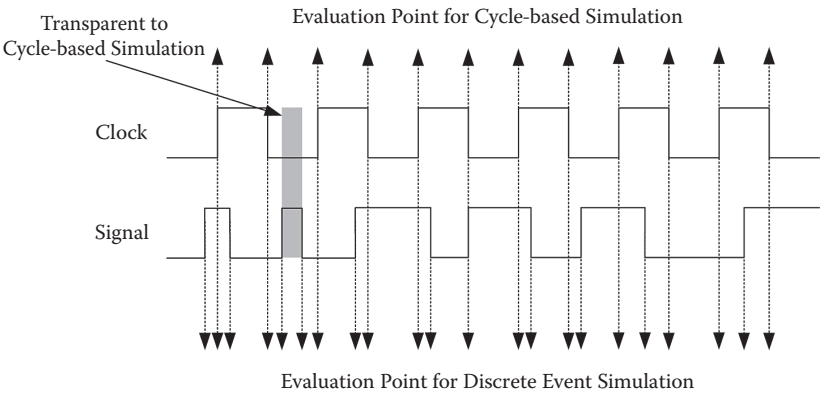


FIGURE 2.17 Evaluation point: discrete-event simulation versus cycle-based simulation.

```
        propagate_event();

        clock++;

end while
```

The previous example is the simplest form of cycle-based simulation. One iteration corresponds to one clock cycle. It iterates a loop while there is any event. Within the loop, it just executes every model not knowing whether it has any event to process. Therefore, the cycle-based simulation should first detect the causal relationship between models.

Generally, it is said that cycle-based simulation is faster than event-driven simulation by about 10 to 50 times. However, the efficiency of cycle-based simulation is directly proportional to the number of models that have events to process at each cycle time. Figure 2.17 compares the cycle-based and discrete-event simulations in regard to the number of evaluations for events. Discrete-event simulation reacts to every type of event. On the contrary, cycle-based simulation only treats the activity occurring at the clock edge. However, as we can see at the gray area in the figure, cycle-based simulation is not capable of detecting interclock activity such as glitch. Therefore, the purpose of cycle-based simulation is to verify the correctness per system cycle by investigating the input and output value.

2.2.1.3 Transaction-Level Simulation

Basically, transaction-level simulation has no mechanism that is different from that of cycle-based or discrete-event simulation. In other words, the transaction-level simulator can be constructed based on the cycle-based or discrete-event simulation algorithm. Figure 2.18 contrasts transaction-level simulation with the discrete-event type. Discrete-event simulation manages events queues that store control and data information. In other words, a time-stamped message is routed to the destination module when the time of the message is imminent. On the contrary, control and data are separated in transaction-level simulation. Control (i.e., when will the message be

transferred) is determined by counting the number of tick events that are generated by constantly moving the tick event generator.* Data are transferred by function call implemented at the destination model, possibly using C/C++ *memcpy* function or the assignment operator. This simulation eliminates the necessity of managing event queues, and event routing, which results in fast simulation.

2.2.2 Formal Method

Although simulation is a useful technique to validate or verify a system, it has a coverage problem. Generally, simulation is performed with test vectors. Verifying or validating a model against a small set of test vectors does not guarantee that the model is free of bugs. As a counterpart of simulation, there is formal method. If a model passes a formal verification successfully, we can say that the model is free of bugs. Therefore, the formal method can be very useful for verifying/validating safety-critical system designs such as automotive control systems. The formal method uses every possible mathematical tool to verify/validate a model. Unfortunately, it is true that the formal method is not utilized extensively due to its algorithmic complexity. However, it is also true that formal verification tools, such as Formality from Synopsys [28], Incisive Formal Analysis from Cadence [29], FormalPro from Mentor Graphics [30], are gaining interest in electronic system design because guaranteeing that a system is bug-free is a great advantage. As stressed by Johnson [31], it may be that education is the limiting factor in the prevalence of formal methods. In the following text, we will briefly introduce representative formal methods instead of enumerating complex mathematical formulas, because the purpose of this book is the introduction of possible verification techniques:

- Model checking—to checks if the system described in finite-state-based formalism satisfies specification in temporal logic:
  - Temporal logic is a logic to express the temporal relationship between states.
  - Finite-state-based formalism includes FSM, automata, CSP, and Petri net.
- Theorem proving—proves a theorem using a set of axioms and inference rules:
  - It carries out formal proof on specification or implementation models.
  - How to sequence formulas is the key to deduction.
- Equivalence checking—checks if two systems in automata-based formalism is equivalent:
  - Language containment—checks if one model is contained in another (homomorphism):
    - For FSM, $M_1 = <S_1, X_1, Y_1, \delta_1, \lambda_1>$ and $M_2 = <S_2, X_2, Y_2, \delta_2, \lambda_2>$, M2 is contained to M1 iff there exists two functions h and k such that

$$\forall s \in S1,\ \delta_1(h(s)) = h(\delta_2(s)) \text{ and } \lambda_1(k(s)) = k(\lambda_2(s))$$

  - Bi-simulation relation: similar to language containment, except that h and k are not functions but relations.
    - Therefore, it is different from language containment.

* Here, TLM is assumed to be an extension of cycle-based modeling/simulation.

FIGURE 2.18 Transaction-level simulation versus discrete-event simulation.

The barrier that blocks the formal method from industry is the complexity of algorithms or state space explosion problems. Therefore, current research on formal verification mainly focuses on abstraction of the specification or implementation model that is under verification.

REFERENCES

1. http://www.uml.org.
2. Martin Fowler, *UML Distilled*, 3rd ed, Addison-Wesley.
3. Daniel D. Gajski, Zhu, J., Rainer Doemer, Gerstlauer, A., and Zhao, S., *SpecC: Specification Language and Methodology,* Kluwer Academic Publisher, Boston, March 2000.
4. http://www.systemc.org.
5. Axel Jantsch, *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation,* Morgan Kaufmann Publisher.
6. Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer, *Theory of Modeling and Simulation,* Academic Press, 2000.
7. Daniel Brand and Pitro Zafiropulo, On Communicating Finite-State Machine, *J. ACM,* Vol. 30, issue 2, pp. 323–342, April 1983.
8. D. Harel, *Statecharts:* A Visual Formalism for Complex Systems, *Science of Computer Programming 8*, 1987, pp. 231–274.
9. D. Gajski et al., *High Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers.
10. Thorsten Grotker et al., *System Design with SystemC,* Kluwer Academic Publishers.
11. TLM White Paper, http://www.systemc.org/.
12. A.P.W. Bohm, *Dataflow Computation,* CWI Tracts, 1983.
13. Kavi K. M. et al., A Formal Definition of Data Flow Graph Models, *IEEE Transactions of Computer,* November 1986.
14. Edward A. Lee and Thomas M. Parks, Dataflow Process Networks, *Proc. IEEE,* Vol. 83, no. 5, pp. 773–801, 1995.
15. S.S. Bhattacharyya et al., *Software Synthesis from Dataflow Graphs,* Kluwer Academic Press, 1996.
16. Gilles Kahn, The Semantics of a Simple Language for Parallel Programming, in *Proc. IFIP 74,* Rosenfeld, J.L., Ed., North-Holland, Amsterdam, 1974, pp. 471–475.
17. Edward A. Lee and David G. Messerschmitt, Synchronous Data Flow, *Proc. IEEE,* Sept., 1987.
18. Daniel D. Gajski et al., High-level Synthesis: *Introduction to Chip and System Design,* Springer.
19. Matthew Hennessy, *Algebraic Theory of Processes* The MIT Press.
20. Robin Milner, *Communication and Concurrency,* Prentice Hall, 1989.
21. Robin Milner, *A Calculus of Communicating Systems,* Vol. 92, LNCS, Springer-Verlag.
22. Hoare, C. A. R., Communicating Sequential Processes, *Communications of the ACM,* 21(8), pp. 666–676, 1978.
23. van Eijk, P. H. J., Vissers, C. A., and Diaz, M., *The formal description technique LOTOS,* Elsevier Science Publishers B.V., 1989.
24. Colin Fidge, *A Comparative Introduction to CSP, CCS and LOTOS,* Technical Paper No.93–24, Software Verification Research Center, Dept. of CS, University of Queensland, 1994.
25. Stephen Edwards et al., Design of Embedded Systems: Formal Models, Validation and Synthesis, *Proc. IEEE,* Vol. 85, no. 3, March 1997.
26. Ahmed A. Jerraya and Wayne Wolf, *Multiprocessor Systems-on-Chips,* Morgan Kaufmann Publishers, San Francisco, 2005.

27. Daniel D. Gajski, *Principles of Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1997.
28. http://www.synopsys.com/products/verification/verification.html.
29. http://www.cadence.com/products/functional_ver/incisive_formal_verifier/index.aspx.
30. http://www.mentor.com/products/fv/ev/formalpro/.
31. Steven D. Johnson, *Formal Methods in Embedded Design,* Computer, pp. 104–106, November 2003.

3 Hardware/Software Codesign

In this chapter, we will briefly explain codesign methodology from the view point of NoC designers. Codesign is an effort to shorten design time by overlapping hardware and software development.

3.1 CODESIGN

Time-to-market, which can be defined as the period of time taken for an idea to become a product, is very important for a product. Figure 3.1 shows a typical revenue-to-time graph. Two products—P1 and P2—for the same application are on the market at time T1 and T2, respectively. A slight difference between the products' development time can result in gross difference in revenue. Reducing design time is the key to the success of a real product.

A system consists of hardware and software. Therefore, the system design also consists of both hardware and software designs. In the traditional design shown in Figure 3.2, the software designer had to wait for the hardware to be developed. The design activities were performed in a serial manner. What is worse, if hardware error is found, or developed hardware cannot meet system requirement during software design, the hardware design should be done again, and software designers should wait. With codesign methodology, the design activities are parallelized as much as possible. Hardware design is performed as before. Software design is performed simultaneously using a virtual platform instead of the real hardware under design. As a result, design time, or time-to-market, can be reduced significantly.

Figure 3.3 shows the traditional codesign framework. It starts with an analysis of the application and requirements. Then the system specification is generated. Next, hardware and software partitioning is performed. Here, software covers the processor and the codes implemented on it, and hardware refers to special functional units. In many cases, software covers the control part also. The requirements for interfaces between hardware and software, or between hardware and hardware are specified, such as device driver or DMA. Partitioned specifications of hardware and software are also generated. The hardware and software are developed by respective implementing procedures. Hardware designers can (1) design a completely new module, (2) reuse predesigned, preverified, modules, and (3) exploit the conventional synthesis methodology. The software developer can make machine codes by (1) writing down the assembly code and using the assemblers and (2) writing down software in a high-level programming language such as C/C++ and using the compiler. Interface synthesis is a procedure to connect hardware and hardware or hardware and software. Traditional research on hardware–hardware interface synthesis includes the composition of state machines of the two modules that are to be connected. Recently,

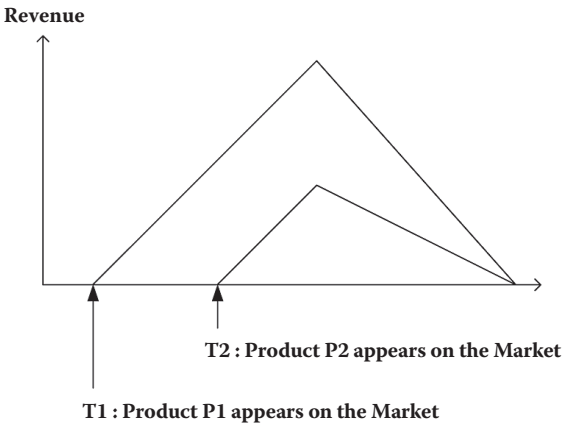


FIGURE 3.1 Time-revenue relationship.

NoC has replaced the interface of the target system. Finally, cosimulation between the hardware and software components are performed to verify the correctness and to measure the performance index of the designed system. The rest of this chapter gives more detailed research results on application analysis, hardware–software partitioning, resource allocation, and scheduling.

The platform-based design, as explained in Chapter 1, Section 1.3, has introduced significant improvement to design methodology. Sangiovanni-Vincentelli, a key originator of the concept, has defined it as a "layer of abstraction with two views." The upper view allows an application to be developed without referring to the lower levels of abstraction. The lower view is a set of rules that classify a set of components belonging to the platform [1].

In addition, the platform-based design is defined in various ways as follows:

- Library of components together with their composition rules [2]
- Abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer in the design flow [3]
- An integration-oriented design approach emphasizing systematic reuse [4]
- The creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and delivered to customers for quick deployment [5]

Figure 3.4 shows a general picture of the platform-based design. There are two important aspects of a system—application and platform. The application is analyzed during the profiling phase. The characteristics of the application are investigated using various techniques, which will be explained in more detail in the next section. The platform consists of the original design (which is chosen as a good architecture for some specific application) and rules to derivate a similar but new design. Along with the application analysis results, one platform, or a set of platforms, is selected at the prune phase. Then, to find the optimal architecture, the model synthesis–evaluate phase is iterated. To construct an executable specification, the model base is incorporated.

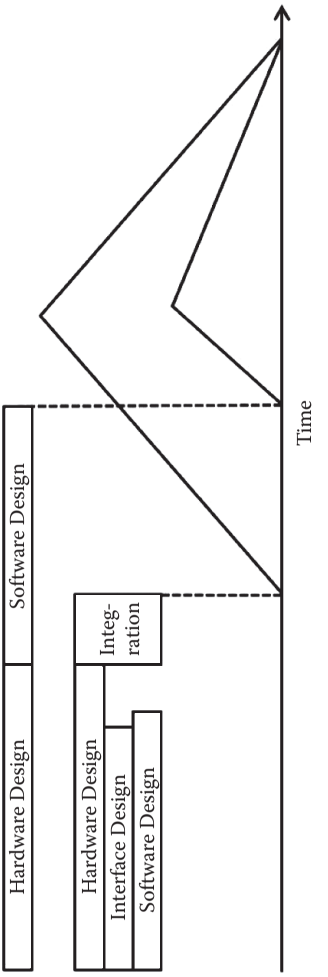


FIGURE 3.2 Effect of codesign in time-revenue relationship.

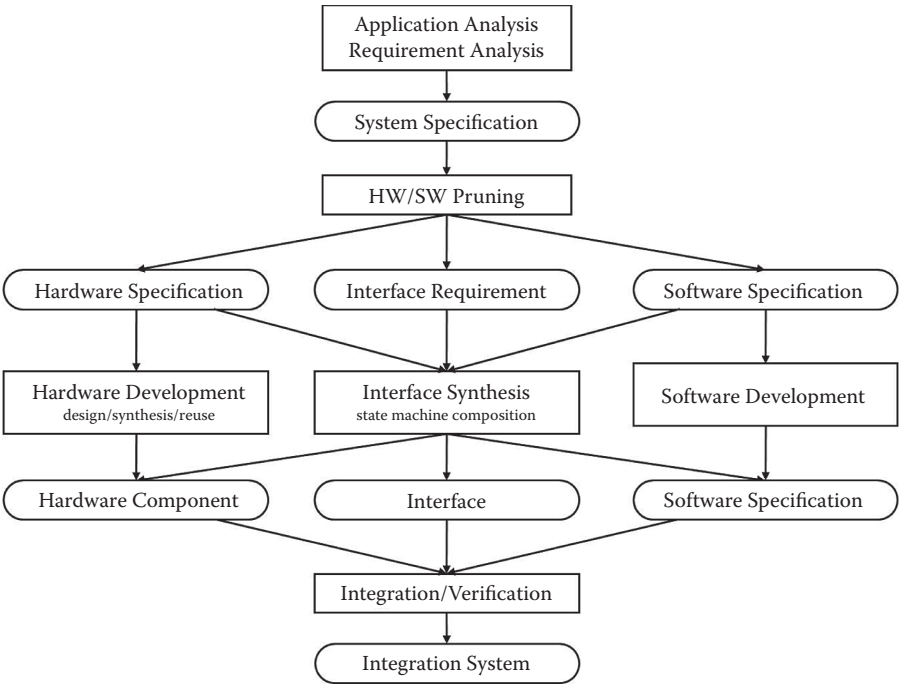


FIGURE 3.3 Traditional codesign framework.

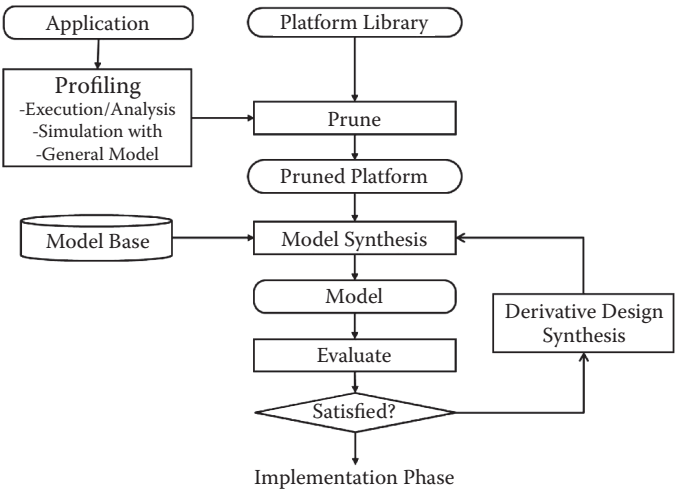


FIGURE 3.4 Platform-based design.

Key components of the methodology, application analysis, partitioning/mapping, and scheduling, will be explained in more detail in Chapter 4 through 5.

3.2 APPLICATION ANALYSIS

3.2.1 Performance Index

Application analysis is an activity to explore various characteristics of application that will be implemented as a system. As a result of application analysis, the designer obtains formal definition of the target application. It enables designers to perform various verification/validation activities. In this chapter, general comments on application analysis will be given first; the UML-based dataflow modeling example will be given with the vision recognition algorithm. The first point for consideration in application analysis is what designers want to investigate as performance indexes. The formal representation of applications consists of tasks connected by links, which can be abstracted to communication. Three aspects for application can be considered: overall performance, task-level, and link- (communication-) related performance.

- Overall performance
  - Speed
  - High-level power consumption
- Task level
  - Execution count (hotspot)
  - Required buffer size
- Communication performance index
  - Traffic
  - Congestion

Table 3.1 shows each performance index and what we can do by investigating the respective indexes. First, if the execution time of application is longer than required, or the current application implementation cannot satisfy the real-time requirement, the first thing to do is to optimize the application. It may include the optimization of either task or communication, or both. Of course, at a later stage of the system design, hardware can, or must, be incorporated. Application optimization is very important; there should be no memory leakage, redundant code, etc. It is difficult to predict power consumption during application-level analysis. However, the pattern of data to be exchanged between tasks can be a clue for power optimization of the network. Although not shown in the table, parallelism gives designers hints on concurrent execution of multiple tasks that constitute the target application.

The main interest in performance indexes related to communication is for the design of efficient network systems. Analysis on tasks can be hints for PE or ASIC design. Hotspots can be distributed to various PEs by considering data-parallel features.

Figure 3.5 describes various application analysis methods. Manual analysis implies modeling applications with tasks connected by links and calculating the various performance indexes from data source to sink. The domain expert's intuition

TABLE 3.1
Performance Index for Application Analysis

Category	Performance Index	Effect on System Design
System	Speed	Optimize application: divide a single task to many, or merge different tasks to one
	Power	Optimize application
Communication	Traffic	Determine buffer size for communication system
		Optimize ordering of data to be transferred
	Congestion	Distribute tasks into PE efficiently
		Design congestion control block
		Partition and schedule application
Task	Execution count	Distribute each execution by considering data dependency
		Develop more efficient PE
	Buffer size	Determine buffer size for PE to be mapped

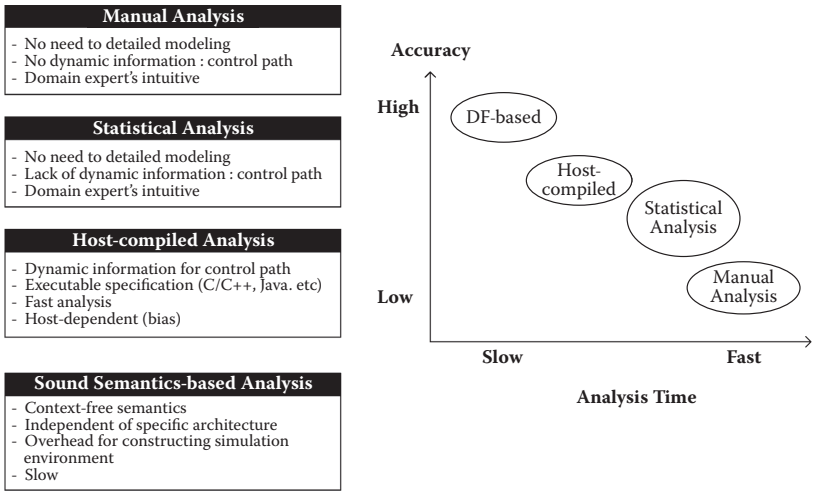


FIGURE 3.5 Application analysis.

is inevitable, because there is no information on control characteristics. That is, there is no means to know how many times each task will be visited during execution.

Host-compiled analysis is done by just running the binary targeted to the host machine. This method is fast, and we can come by some performance indexes such as hot spot, memory usage, etc. However, the property and characteristic of the host CPU affect analysis results. In addition, the compiler used for compiling the application can also alter the structure of the application, which will affect the profiling results. That is, we can get incorrect results, biased by the host machine and compiler.

The last task is to introduce a formalism that does not reflect a specific platform. Figure 3.6 depicts the outline of such a method, which adopts dataflow-based

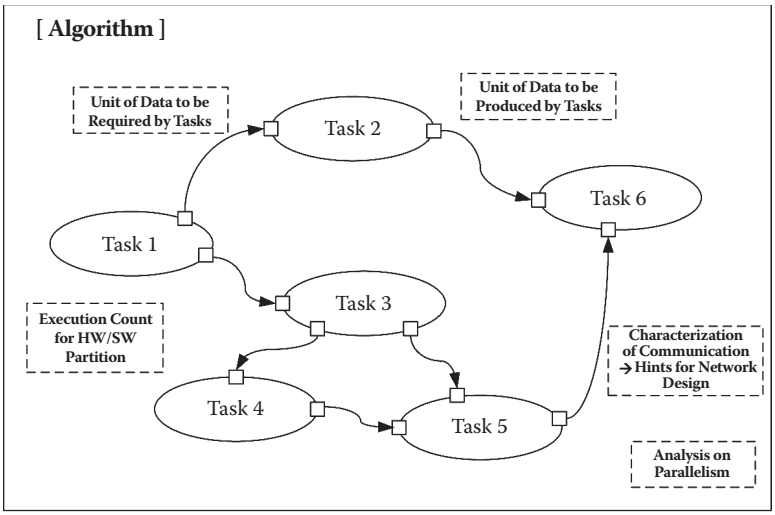


FIGURE 3.6 Algorithm representation with task-level model.

approach. Basically, an application is a directed call graph between tasks. From the source task, or outside the application, data are generated to the rest of the task nodes. Only if the condition for a task to fire is satisfied, the task is invoked. Then, the task consumes a fixed amount of input data, and also generates a fixed amount of output data. The edge between task nodes implies the buffer between tasks. The capacity of this buffer determines the characteristics of this semantics. This is exactly the same with the dataflow formalism explained in Chapter 2.

Because the semantics is so simple, the simulation/execution semantics is also simple, as will be explained later in this chapter. By running the algorithm based on this semantics, we can obtain the following unbiased characteristics of application and performance indexes as in Figure 3.6.

- Parallelism
  - We can investigate what tasks can be computed in a parallel manner.
- Execution count (hot spot)
  - This information will be used for hardware/software partitioning.
- Amount of data exchange between tasks

Based on this information, partitioning of tasks, mapping from function to PE, and, finally, scheduling can be determined. Then, the next section will show how we can actually construct application analysis frameworks with UML.

3.2.2 Task Graph: Sound Semantics for Application Analysis

The basic task graph model, which is a sound semantics for application representation, can be formalized as follows:

TG is a graph G = <V, E>, where

- *V: a set of tasks*
  - *v = <P_{in}, P_{out}, in, out, process>, where*
  - *P_{in} is a set of input ports*
  - *P_{out} is a set of output ports*
  - *in ⊆ P_{in} × Z, where Z is a set of nonnegative integer*
    - *Required number of inputs for process function to be invoked*
  - *If there is (p_1, n) ∈ in, then there is no (p_1, m) ∈ in such that n ≠ m*
  - *out ⊆ P_{out} × Z, where Z is a set of nonnegative integer*
    - *Number of outputs generated as a result of function execution*
    - *If there is (p_2, n) ∈ out, then there is no (p_2, m) ∈ out such that n ≠ m*
  - *Process*
    - *A function that translates data at input buffer and generates data to output buffer*
  - *E ⊆ V.P_{out} × Z × Z × V.$P_{in,}$ where Z is a set of nonnegative integer*
  - *There is a buffer at each edge*
  - *(v_1.p_1, z_1, z_2,v_2.p_2) ∈ E implies*
    - *There is a path from p_1 port of v_1 to p_2 port of v_2 through buffer with size z_1*
    - *z_2 is the number of items in the buffer*
  - *z_2 cannot be greater than z1*

Figure 3.7 shows the basic edge relationship between two functional nodes. These nodes are connected through one buffer. The buffer size is configurable, i.e., it can be an infinite first in first out (FIFO) queue or a buffer with 1, 2, 4, … capacity. The semantics is very simple; v_1 can execute its function if, and only if, there are enough data at e_1 in front of the node $v_{1,}$ as well as enough capacity at e_2. When there is more than one input edge for a vertex, all the input conditions should be met for the vertex to be executed. This semantics can simply be translated to other semantics, such as Petri net or dataflow graph.

The read/write mechanism for this semantics is based on the capacity of the buffer located at the edge. In any case, read is blocking, which implies that a task will be pending in case of insufficient data at the input buffer. If the size of the buffer is infinite, write is nonblocking, as with the Kahn Process Network. Figure 3.8 describes this network. If the size of the buffer is finite, write can be blocking.

By configuring the buffer size, the number of required input data, and the number of generated output data at each node, the task graph can also be converted to a

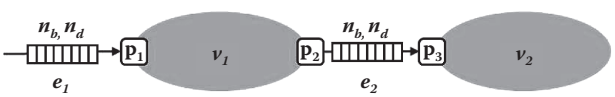


FIGURE 3.7 Tasks and links.

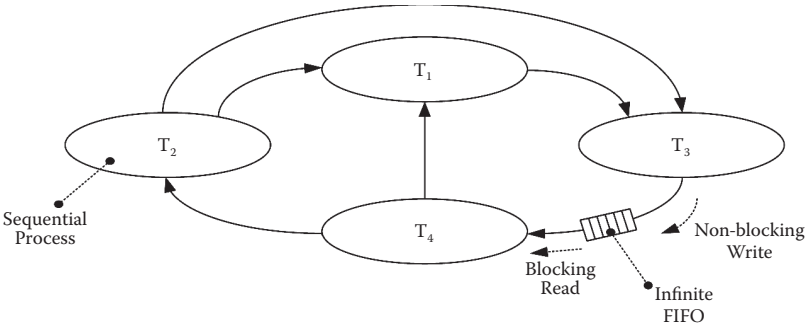


FIGURE 3.8 Kahn Process Network.

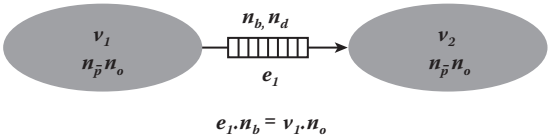


FIGURE 3.9 Synchronous dataflow.

synchronous dataflow graph, as shown in Figure 3.9. Then, some functions can be defined for the task graph (TG).

For task graph G = <V, E>

- *req:* $V \times V.P_{in} \rightarrow Z$
  - $req(v_1, p_1) = n$, *where* $(p_1, n) \in v_1.in$
  - *returns the required number of inputs for input port*
- *gen:* $V \times V.P_{out} \rightarrow Z$
  - $gen(v_1, p_1) = n$, *where* $(p_1, n) \in v_1.out$
  - *returns the number of generated outputs for the output port*
- *input_edge:* $V \rightarrow E^n$
  - *return a set of edges whose destination vertex is the parameter of the function*
- *output_edge:* $V \rightarrow E^n$
  - *returns a set of edges whose source vertex is the parameter of the function*
- *vfireable:* $V \times input\_edge(V) \rightarrow \{ 0, 1 \}$
  - $vfireable(v_1, E_{in}) = 1$ *iff* $\forall p \in v_1.p_{in}$, *there exists* $(*, z_1, z_2, v_1.p) \in E_{in}$ *such that* $z_2 \geq req(v_1, v_1.p)$
  - $vfireable(v_1, E_{in}) = 0$ *iff* $\exists p \in v_1.p_{in}$, *there exists* $(*, z_1, z_2, v_1.p) \in E_{in}$ *such that* $z_2 < req(v_1, v_1.p)$

Based on these functions, more complicated ones can be defined. First, *fetch_data* function is defined in such a way that it fetches a fixed amount of data, which is determined at the vertex definition.

```
fetch_data ( v of type V, g of type TG)
for ∀p ∈ v.P_in begin
    construct a set of edge E' = { (*, *, *, v.p) }
    for ∀e = (*, z1, z2, v.p) ∈ E' begin
        e = (*, z1, z2 – req(v, v.p), v.p)
    end for
end for
```

Secondly, *write_data* is defined in such a way that the fixed amount of data generated is written to the buffer connected as output. The amount of data is also defined at vertex semantics.

```
write_data ( v of type V, g of type TG)
    for ∀p ∈ v.P_out begin
    construct a set of edge E' = { (v.p, *, *, *) }
    for ∀e = (*, z1, z2, v.p) ∈ E' begin
        e = (*, z1, z2 + gen(v, v.p), v.p)
    end for
end for
```

Before defining the simulation function, let us investigate the execution semantics of the vertex. Figure 3.10 shows an example of an actual dataflow graph. There is a buffer with size 4 between nodes 1 and 2. Node 2 can fire when data at the input buffer are more than one. Once it fires, it will put one unit of data to the output buffer. At the initial state, the vertex stays at the wait state. Periodically, it checks if there are more data at the input buffer than required. If not, it gets back to the wait state. Otherwise, it runs the process function. After process, it checks if there is enough room at the output buffer. If so, it writes the generated data to the output buffer. Otherwise, it waits until there is enough room for the output buffer.

This procedure is described in the following *simulate* algorithm. It iterates until the *stop* condition is satisfied. The stop condition depends on the application.

```
simulate(g of type TG)
while true
    for ∀v ∈ g.V begin
        if(vfireable(v, input_edge(v) == 1) begin
            fetch_data(v, g);
        end if
    end for
    process();
    for ∀e ∈ g.E begin
        if(efireable(e, output_edge(v) == 1) begin
            write_data(v, g);
        end if
    end for
end while
```

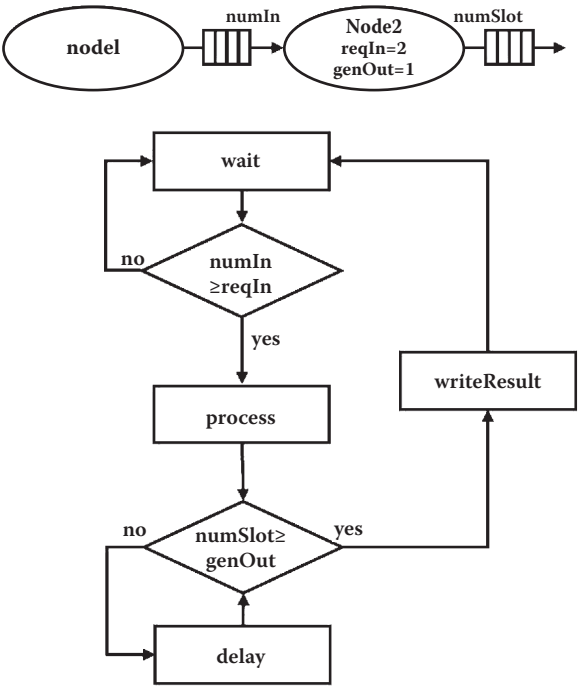


FIGURE 3.10 Action semantics for task graph.

3.2.3 Implementing Task Graph in Unified Modeling Language (UML)

We will show how UML can be incorporated to support application analysis based on the task graph by exemplifying vision algorithm.

Figure 3.11 shows the class hierarchy for dataflow modeling. CPROFILE is a class that defines performance indexes and print functions. DF_NODE and DF_BUFFER inherits from it; therefore, it shares the list of performance indexes. More detailed explanation on DF_NODE and DF_BUFFER will be given later.

Next, interface is defined. Interface is a contract between two actors, which can be used to send/receive events or data. As explained earlier, there are two kinds of interfaces—required and provided—and they can be represented by a lollipop notation as in Figure 3.12.

In Figure 3.12, *my_class* has one provided interface, *writeBuffer*, and one required interface, *readBuffer* (*req_ifc* and *prv_ifc* are the names of ports, respectively). The provided interface is, as its name implies, an interface that should be implemented and provided by the class. Required interface of a class is an interface that the class requires for another class. Of course, the other class should provide the interface. Then, *my_class* in Figure 3.12 should define the function abstractly declared in the *writeBuffer* interface. The other class, which requires the *writeBuffer* interface, can access the *my_class* instance through the function. Conversely, the instance of *my_class* can access the other instance of the class, which provides the *readBuffer* interface.

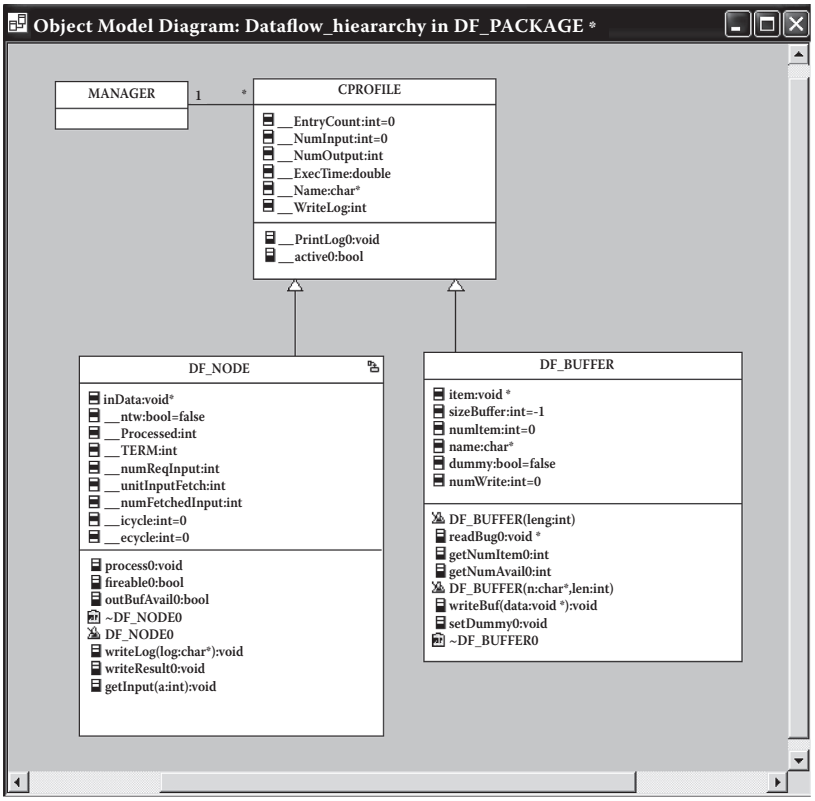


FIGURE 3.11 Class diagram for dataflow.

We have defined the two interfaces in Table 3.2.

The buffer is a passive component. That is, other entities can write to and read from it. Therefore, these two interfaces should be provided by the DF_BUFFER class, as shown in Figure 3.13. That is, DF_BUFFER should implement all the functions listed in Table 3.2.

Every active process should inherit from DF_NODE, which enables the use of every mechanism implemented at DF_NODE. For example, Figure 3.14 exemplifies the class diagram for the vision recognition algorithm. Every active process inherits from DF_NODE. Then, they inherit the state diagram implemented by DF_NODE,

TABLE 3.2
Interface for UML Implementation of Dataflow

Interface	Function	Description
readBuffer	getNumItem	Return number of data contained in DF_BUFFER
	readBuf	Read data contained in DF_BUFFER
writeBuffer	getNumAvail	Return number of available slot in buffer
	writeBuf	Write data to DF_BUFFER

FIGURE 3.12 Interface.

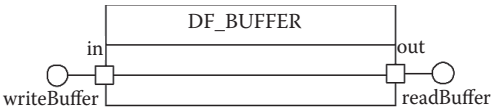


FIGURE 3.13 Buffer class for dataflow.

as in Figure 3.15, and the state diagram implements the action semantics of the dataflow node. There are four characteristic functions: *fireable, process, outBufAvail,* and *writeResult*. These functions are declared abstract in DF_NODE. Therefore, every active process has to implement the four functions.

For example, cGaussianFilter defines the four functions as follows. First, the fireable function checks if it can fire to see the number of available data items at the input edge.

```
Fireable()
    if(FIREABLE(inP, numReqInput)) return true;
else return false;
```

If the fireable function returns true, it can execute its process function. The process function first fetches the required number of data from the input buffer and performs computation. For example, the following process function for a Gaussian filter computes the resulting value with filter masks.

```
process()
/* Firstly, fetch the required data */
int i, j;
double* tInt;
DECLARE_TWO_DIMENSIONAL_ARRAY(double, tmpStore, 5, 5)
for(i = 0; i < 5; i++){
    for(j = 0; j < 5; j++){
        READ_BUFFER(tInt, inP, (double*));
        tmpStore[i][j] = *tInt;
        delete (double*)tInt;
    }
}
/* Secondly, compute */
result = 0.0;
for(i = 0; i < 5; i++){
    for(j = 0; j < 5; j++){
        result += tmpStore[i][j] * mask[ord][i][j];
    }
}
result /= filterWeight[ord];
DELETE_TWO_DIMENSIONAL_ARRAY(tmpStore, 5, 5);
```

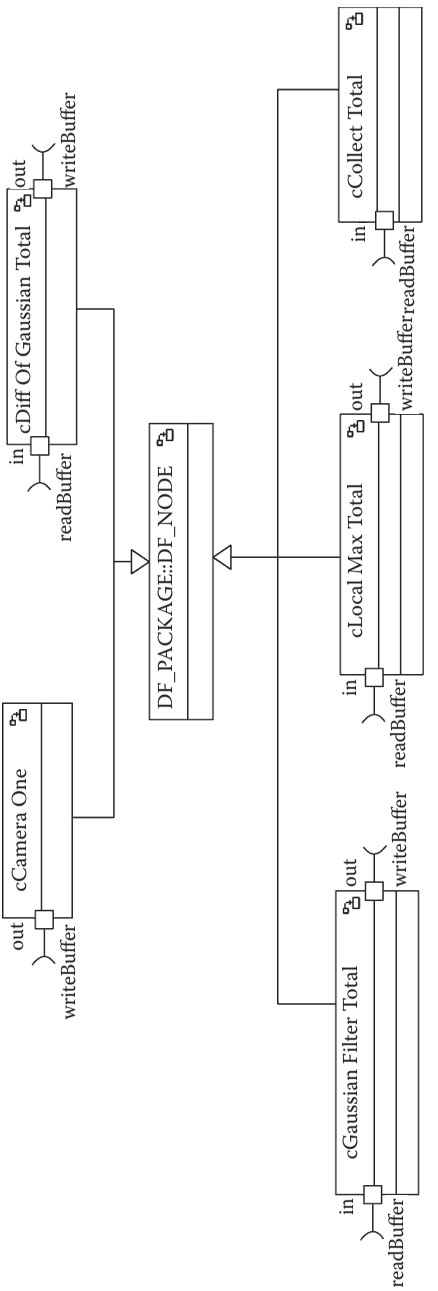


FIGURE 3.14 Tasks for Vision Recognition Algorithm.

FIGURE 3.15 Inheriting action semantics for dataflow node.

After performing the process function, it should check if there is enough room at the output edge. In case of the Gaussian filter example, three output edges are connected. Therefore, it checks all the three output buffers.

```
outBufAvait()
return (WRITEABLE_AT(outP, 0, numGenOutput0) && \
    WRITEABLE_AT(outP, 1, numGenOutput1) && \
    WRITEABLE_AT(outP, 2, numGenOutput2));
```

If there is enough room, it writes data to the output buffer and gets back to the wait state.

```
writeResult()
double* res = new double;
(*res) = result;
WRITE_BUFFER_AT(outP, res, 0);

res = new double;
(*res) = result;
WRITE_BUFFER_AT(outP, res, 1);
```

By connecting the modeled active components and buffers, the application construction is complete. Figure 3.16 shows the overall structure of the application.* By using the automated compilation procedure, we could generate the C/C++ code and compile the application. Consequently, the performance index declared at the CPROFILE class could be collected.

3.3 SYNTHESIS

The result of application analysis should be fed into the next codesign stage—partition, mapping, and scheduling. Each of the terms can be defined as follows:

- Partitioning: Aggregation of a set of primitive tasks into one group such that it can be run on one processing element
- Mapping: Coupling of the partitioned operations into real processing elements
- Scheduling: Conversion of partial order of tasks on each processing element to total order

The three steps are closely related to one another and, therefore, should be iterated to obtain the required performance. Partitioning and mapping are more closely related. Therefore, we will describe the two in one section, and scheduling, in another.

3.3.1 Partitioning and Resource Allocation

We can formulate partitioning/allocation as a problem between application and architecture. Figure 3.17 describes a simple picture of partitioning/allocation. There

* Links that are not shown are due to the limitation of the tool used.

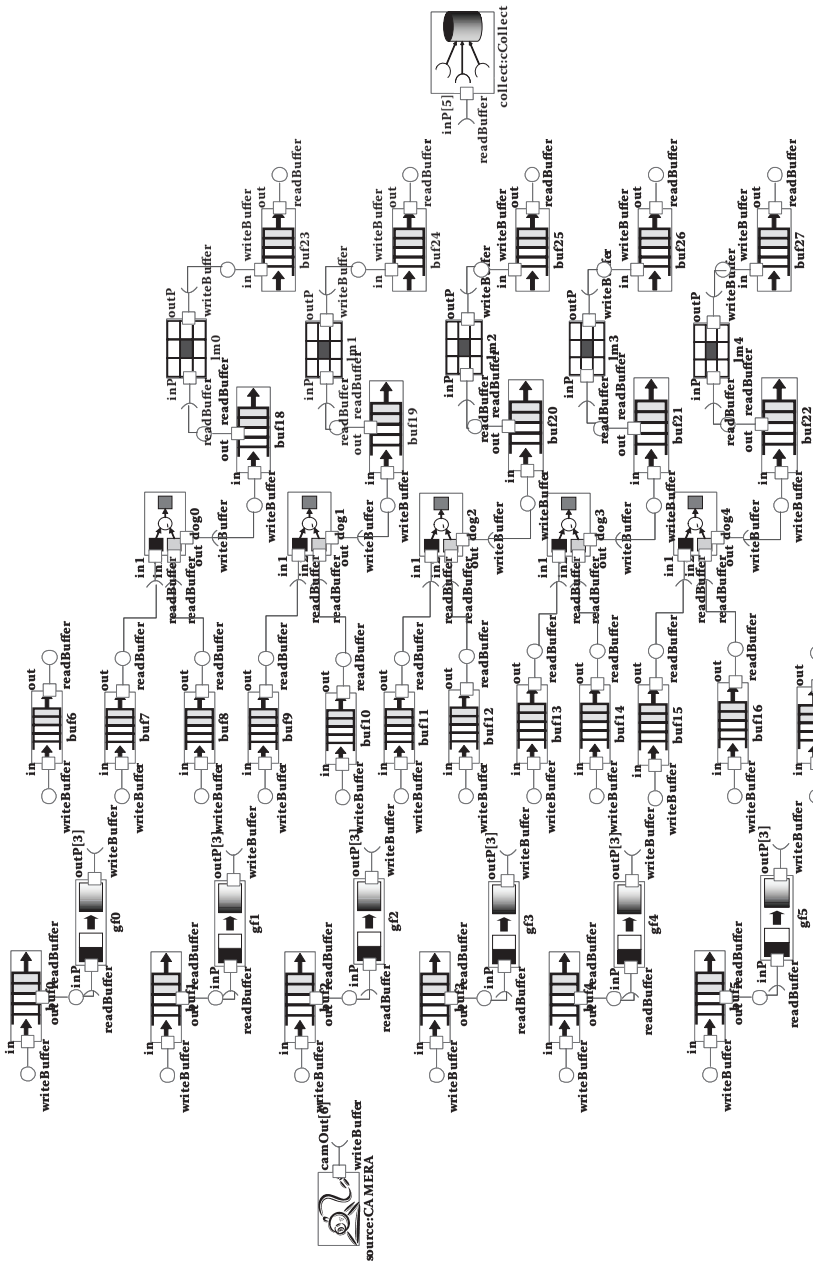


FIGURE 3.16 Vision Recognition Algorithm.

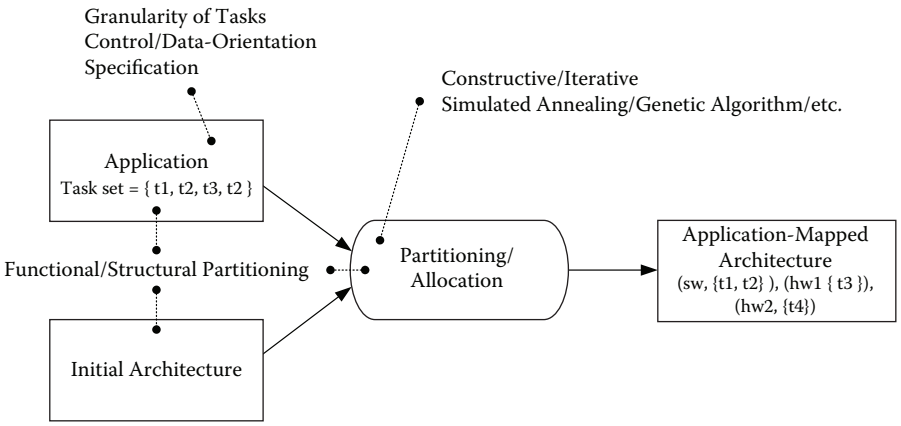


FIGURE 3.17 Partitioning and resource allocation.

are three key components: application, initial architecture, and application-mapped architecture. It is well known that the general partitioning problem is NP-complete, which implies that there is no known good algorithm that can find optimum results within polynomial time. However, partitioning is a crucial step in designing complex systems and has therefore been a major research theme.

Next, let us formulate the partitioning/mapping problem. The task graph G = (V, E) is given such that V is a set of tasks and E is a communication routine between the tasks. For each task, cost functions are given/computed as $cost_h(v)$ for hardware implementation cost and $cost_s(v)$ for software implementation cost. In addition, the cost of communication between nodes can also be given/computed with $cost_{comm}(v_1, v_2)$. Then, the partitioning problem is formulated to divide V into V_h and V_s such that $V_h \cup V_s = V$ and $V_h \cap V_s = \varnothing$, as shown in Figure 3.18.

The most important performance indexes during partitioning/allocation are speed and area. Usually, the partitioning algorithm tries to evaluate a sample partition in terms of speed and area. Figure 3.19 shows the general inclination of speed and area trade-off when tasks are migrated between hardware and software. When a task is migrated to software from hardware, i.e., a new hardware is designed and incorporated, speedup is expected, but additional area due to the new hardware is also to be endured. This applies to migration from software to hardware, too. Therefore, most partitioning algorithms try to (1) optimize area with speed requirements, or (2) optimize speed with area constraints.

We can classify partitioning according to the specification method, whether hardware- or software-oriented, partitioning algorithm, and time of mapping/allocation.

First we can classify the partitioning algorithm based on the initial specification and direction of component migration [6,7].

- Hardware-oriented: Start with hardware configuration for software, and migrate hardware to software till design constraints such as area are satisfied [8].
- Software-oriented: Start with software specification, and migrate the software part to hardware until design constraints such as timing are satisfied [9].

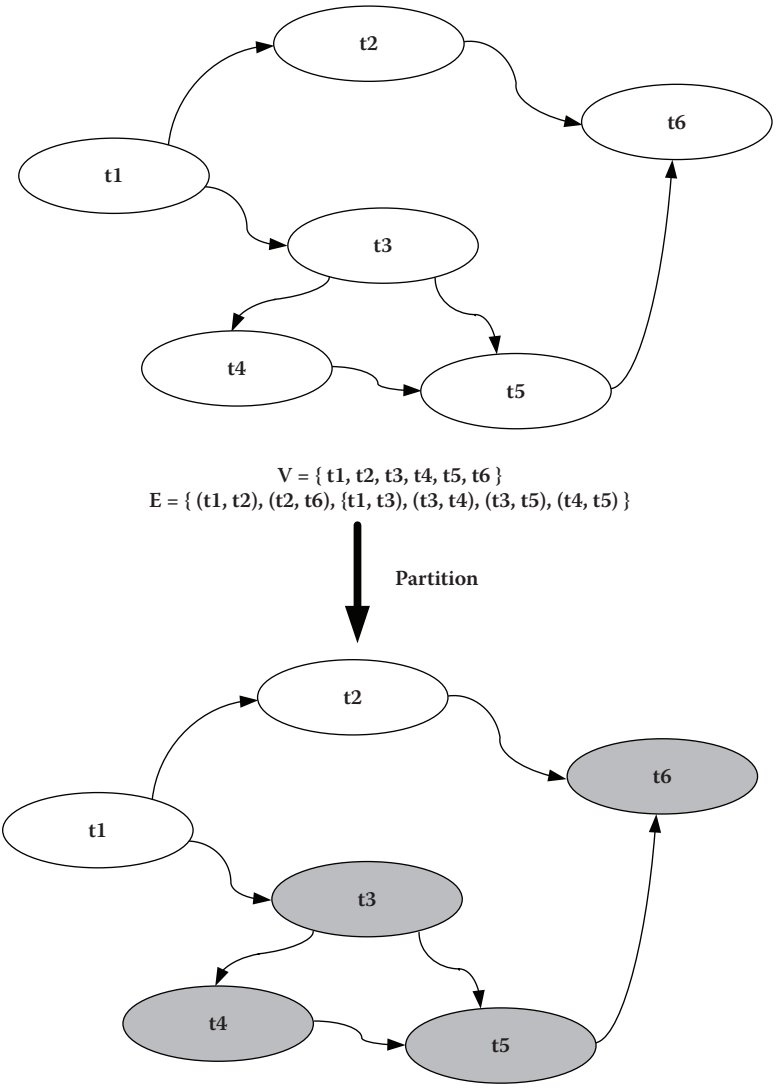


FIGURE 3.18 Partitioning.

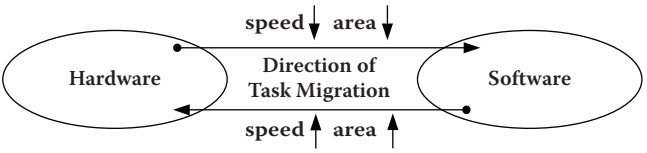


FIGURE 3.19 Inclination of speed/area between hardware and software implementation.

Usually, initial specification for software-oriented partitioning is in the form of a software programming language such as C/C++, and for hardware-oriented partition, in the form of HDL or HardwareC.

As indicated previously in this section, the partitioning problem is NP-complete. Therefore, there have been various heuristics to cope with it, such as dynamic programming, simulated annealing, genetic algorithm, taboo search, etc. Here, we will focus on the types of algorithms in more detail.

- Constructive algorithm
- Iterative algorithm:
  - Greedy algorithm
  - Kernighan-Lin algorithm (Min-Cut)
  - Evolutionary algorithm
    - Simulated annealing
    - Genetic algorithm

Figure 3.20 shows a simple picture of the **constructive partitioning algorithm**. It starts with one or more seeds (or clusters), which consist of tasks. Then, it takes the tasks one by one from the task set and computes the closeness with the current partition. If it is close enough, the task taken will be inserted into the current partition. Otherwise, it iterates with the next task.

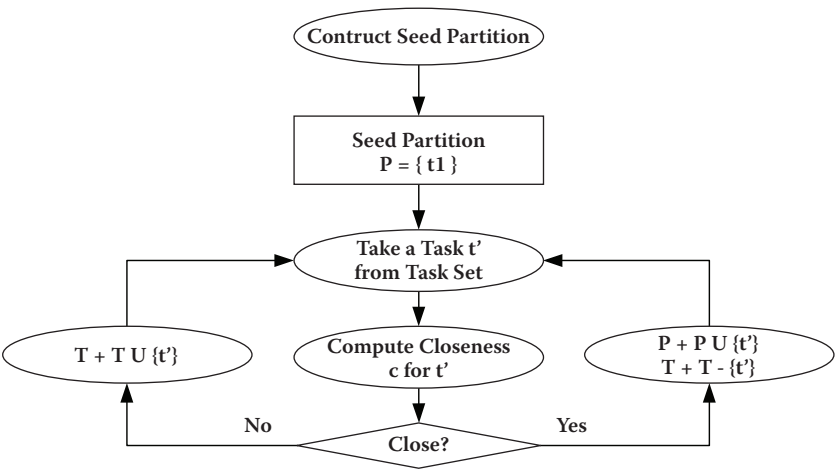


FIGURE 3.20 Constructive partitioning algorithm.

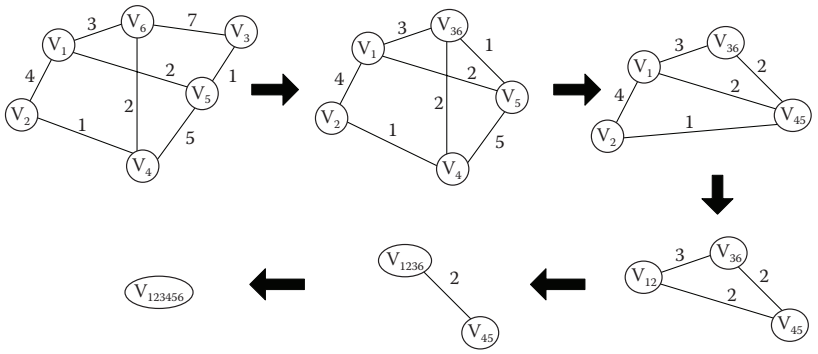


FIGURE 3.21 Hierarchical partitioning algorithm.

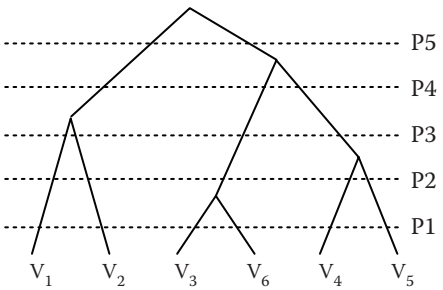


FIGURE 3.22 Partition tree.

Another method of constructive partitioning is the **hierarchical partitioning**, exemplified in Figure 3.21. As shown in the figure, this algorithm aggregates tasks connected by the edges to the biggest weight. It records the order of node merging. If the number of ways of partitioning is set to four, $\{v_1\}$, $\{v_2\}$, $\{v_3, v_6\}$, and $\{v_4, v_5\}$ would be the partitioning candidates.

By iterating this algorithm, we can obtain the partition tree, as shown in Figure 3.22. Then, we can obtain five ways of partitioning, P1, P2, P3, P4, and P5. When there is more than one graph with different weight, for example, two graphs with their own weight, algorithm can be further extended such that the procedure mentioned previously alternates between the two weight graphs.

The iterative algorithm starts with a sample partitioning. Then, this algorithm iterates (1) the modification of the previous partition and (2) evaluation of the modified partition. The greedy algorithm always selects a partition from many that yield better performance. However, this algorithm is apt to fall into a local maximum, as shown in Figure 3.23. To get out of this, various algorithms that exploit evolutionary methods such as simulated annealing (SA) or genetic algorithm (GA) are introduced.

GA is based on the belief that a new chromosome generated from a superior one may have superior performance. Based on this belief, two ways of generating new chromosomes are used: mutation and cross-over. Mutation is to change one part of the chromosome, that is, a gene, and crossover is to mix exclusive genes between two good chromosomes, as shown in Figure 3.24.

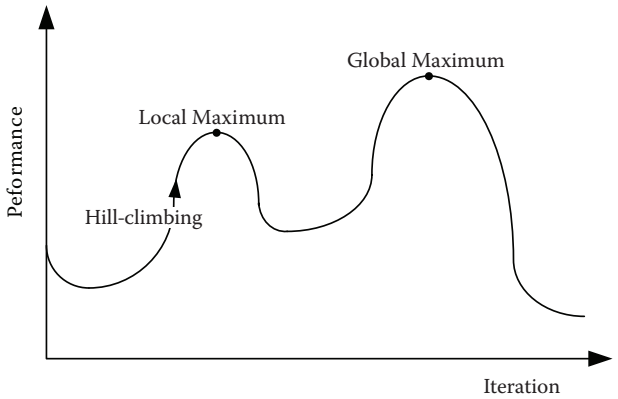


FIGURE 3.23 Hill-climbing algorithm.

Then, GA is applied according to the following procedure:

Step 1. Define the gene for chromosome encoding.
Step 2. Decide the ratio of mutation and crossover.
Step 3. Define the initial population.
Step 4. Calculate the cost of each chromosome with cost function.
Step 5. Stop if stopping criteria are satisfied.
Step 6. Perform mutation and crossover whose ratio is defined in Step 2.
Step 7. Get back to Step 4.

The most important design activity with GA is chromosome encoding. In most cases, each gene represents a function, and it implies whether it will be implemented by software or hardware [10, 11]. For example, when task graph G = (V, E), a set of resources R = {sw, hw}, and cost functions are given, the gene is an element of V × R. That is, (t1, hw) implies that task t1 is partitioned to hardware.

Figure 3.25 shows a simple picture of SA, which introduces the probability of selecting inferior candidates based on the temperature parameter. As shown in the figure, a low temperature makes the probability of accepting an inferior solution higher, so that it can transfer the next envelop. With high temperature, the probability gets low, so that it goes to the peak. The most important design activity for SA is to find the neighbor, which corresponds to mutation or crossover in case of GA, and configure the temperature/cooling ratio. Several researches using SA can be found [7]. There are also approaches using other metaheuristics such as the ant-colony algorithm or tabu search.

The Kernighan/Lin (KL) algorithm has been introduced [12] to divide the access graph into a two-way partition so that edges crossing the cut are minimized. Therefore, it is often called a min-cut algorithm. Figure 3.26 shows an example of the cutting of a graph. By minimizing the number of edges between two partitions, we can obtain minimum communication. However, the original algorithm is not suitable for hardware/software partition problems. Therefore, the KL algorithm has been extended [12] in such a way that it considers the execution time as the cut

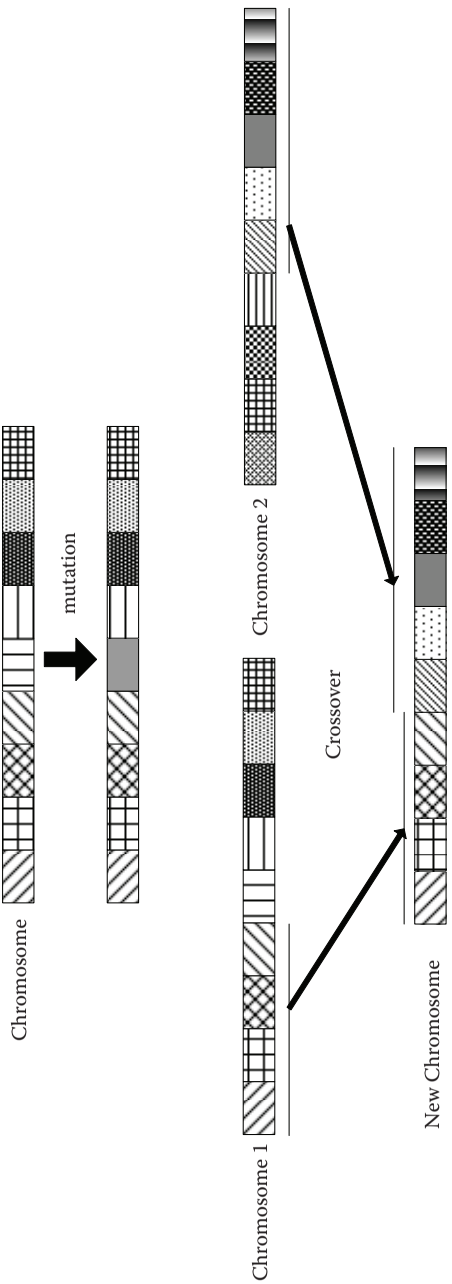


FIGURE 3.24 Genetic algorithm.

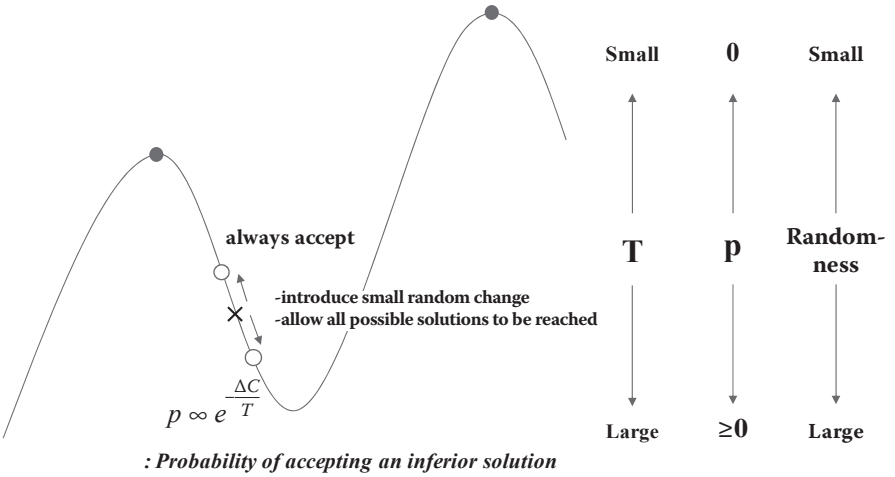


FIGURE 3.25 Simulated annealing.

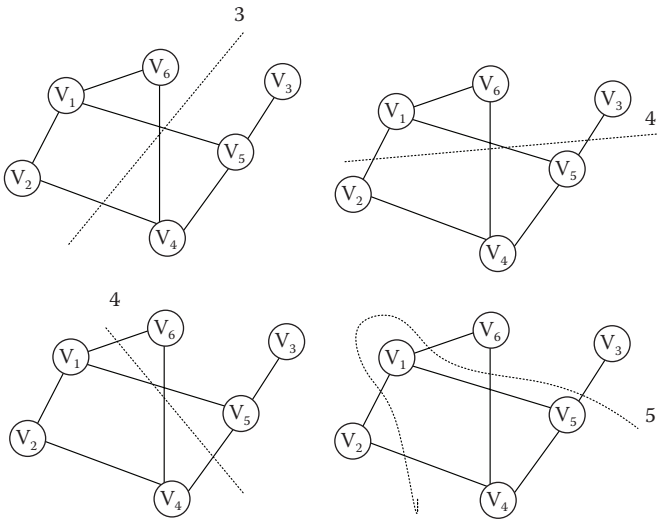


FIGURE 3.26 Cutting of graph.

metric, uses a single move instead of swapping two nodes, and takes a constant time to select the best next move.

Based on the partition target and the order of mapping/allocation, partitioning algorithms can be divided into functional and structural partitioning [13,14,15]. Figure 3.27 shows the procedures of functional and structural partitioning [16]. Functional partitioning first partitions the functional specifications, then obtains system architecture by applying synthesis to the partitioned functions. On the contrary, structural partitioning first obtains structures, possibly from the netlist, and partitions the fine-grained structural elements into hardware and software.

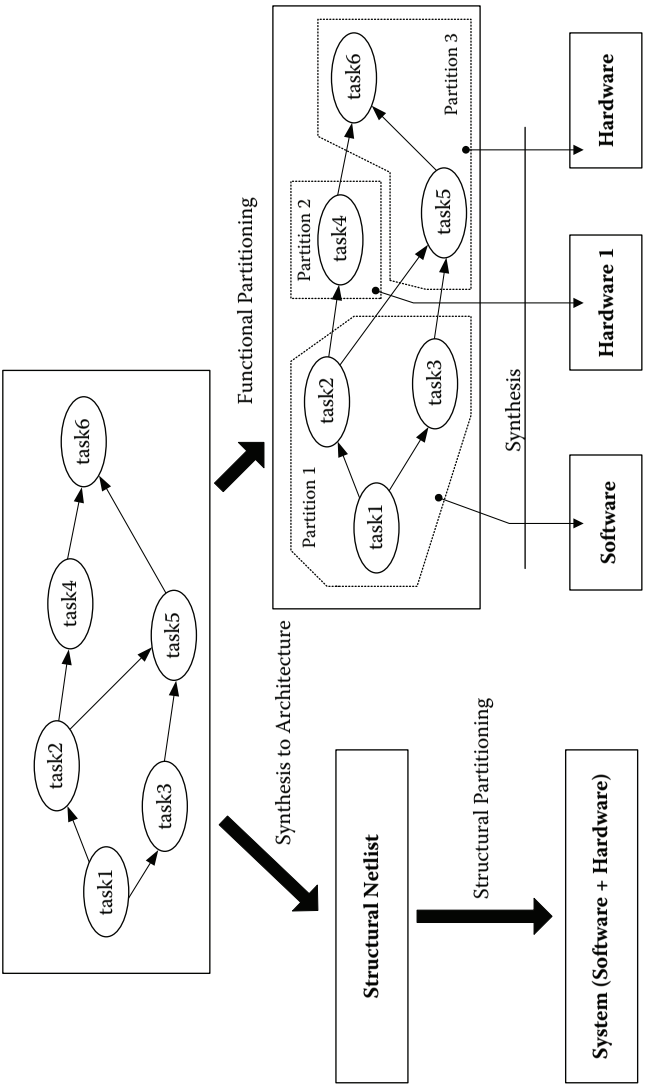


FIGURE 3.27 Structural partitioning versus functional partitioning.

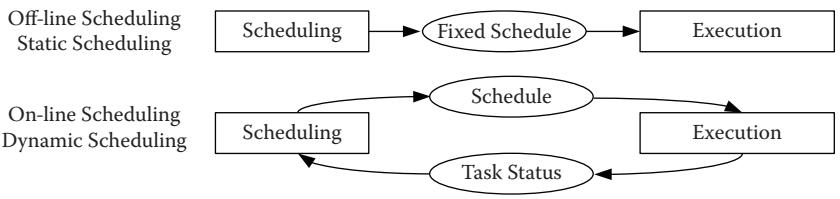


FIGURE 3.28 Static scheduling versus dynamic scheduling.

3.3.2 Scheduling

Scheduling in real-time systems has been a classical problem. Resource-constrained scheduling in codesign is also NP-complete [17]. Scheduling can be classified as on-line and off-line, as shown in Figure 3.28. In case of off-line scheduling, the entire procedure of scheduling is performed before task execution. Therefore, scheduling between tasks does not change during runtime. On the contrary, on-line scheduling uses the runtime status of task execution.

A study [18] classifies various scheduling algorithms into optimal and heuristic ones. Optimal algorithms ensure the quality of output schedules. The optimal algorithm family includes Integer Linear Programming (ILP)-based methods, and the Constraint Logic Programming (CLP), and bipartite graph matching methods. However, it takes too much time to obtain the optimal schedule. In general, the scheduling problem belongs to the NP-hard class, which implies that we have not found whether the problem can be solved in polynomial time. The algorithm to find out the optimal solution does not have scalability against problem space. Therefore, heuristic methods are introduced. The Heuristic algorithm offers a good compromise between the quality of, and time to obtain, the schedule, although it does not guarantee that the schedule found is an optimal one. Heuristic algorithms include the list scheduling algorithm, greedy algorithm, genetic algorithm, etc.

Based on the target of optimization, scheduling algorithms can be classified into resource-constrained scheduling and time-constrained scheduling problems [19]. Resource-constrained scheduling tries to minimize the execution time with the number of available resources fixed. On the contrary, time-constrained scheduling tries to use the minimum number of resources only if the scheduling does not violate timing constraints.

REFERENCES

1. Richard Goering, Platform-based Design: A Choice, not a Panacea, *EE times*, November 09, 2002.
2. Alberto Sangiovanni-Vincentelli et al., Benefits and Challenges for Platform-based Design, *Proc. 41st Annual Conf. on Design Automation*, San Diego, pp. 409–414, 2004.
3. Sangiovanni-Vincentelli, A., Defining Platform-Based Design, *EE Design*, February 2002.
4. Bob Altizer, Platform-Based Design: An Emerging Reality, SoC Online, October 2003.
5. Jean-Marc Chateau, Flexible Platform-Based Design, *EE Design*, February 2001.
6. Wu Jigang, Thambipillai Srikanthan, and Guang Chen, One-dimensional Search Algorithms for Hardware/Software Partitioning, *5th IEEE/ACM Int. Conf. on Formal Methods and Models for Codesign*, pp. 149–158, June 2007.

7. Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli, System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search*, Kluwer J. Design Automation for Embedded Systems,* Vol. 2, No. 1, pp. 5–32, January 1997.
8. Rajesh Gupta and Giovanni De Michelli, Hardware-Software Cosynthesis for Digital Systems*, IEEE Design and Test of Computers*, Vol. 10, No. 3, pp. 29–41, 1993.
9. Frank Vahid, Jie Gong, and Daniel D. Gajki, A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning, *European Design Automation Conf.,* pp. 214–219, September 1994.
10. Hidalgo, J. and Lanchares, J., Functional Partitioning for Hardware-Software Codesign using Genetic Algorithms, *23rd EUROMICRO Conf.*, pp. 631–638, 1997.
11. Pierre-Andr´e Mudry, Guillaume Zufferey, and Gianluca Tempesti, A Hybrid Genetic Algorithm for Constrained Hardware-Software Partitioning, D*esign and Diagnostics of Electronic Circuits and Systems*, pp. 1–6, April 2006.
12. Frank Vahid and Thuy Dm Le, Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning*, Design Automation for Embedded Systems,* 2, pp. 237–261, 1997.
13. Frank Vahid, A Survey of Behavioral-level Partitioning Systems*,* UC Irvine, Technical Report, #91–71, October 30, 1991.
14. Frank Vahid, Thuy Dm Le, and Yu-Chin Hsu, Functional Partitioning Improvements over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance*, ACM Transactions on Design Automation of Electronic Systems*, Vol. 3, issue 2, pp. 181–208, 1998.
15. Elizabeth Dirkes Lagnese and Donald E. Thomas, Architectural Partitioning for System Level Synthesis of Integrated Circuits, *IEEE Transactions on Computer-Aided Design*, Vol. 10, No.7, July 1991.
16. Frank Vahid, Thuy Dm Le, and Yu-Chin Hsu, A Comparison of Functional and Structural Partitioning, *Int. Symp. on System Synthesis*, pp. 121–126, La Jolla, 1996.
17. Garey, M.R. and Johnson, D.S. Complexity Results for Multiprocessor Scheduling under Resource Constraints, *SIAM J. Computing,* Vol. 4, Issue 4, 1975.
18. Peter Arato et al., Time-Constrained Scheduling of Large Pipelined Data Paths*, J. System Architecture,* 51, 2005, pp. 665–687.
19. Heijligers, M.J.M et al., High-level Synthesis Scheduling and Allocation using Genetic Algorithm, *Proc. Asia South Pacific Design Automation Conf.*, 1995, pp. 61–66.

4 Computation–Communication Partitioning

In this chapter, the system is regarded as a combination of several interacting behaviors that need to communicate with one another. Also, the clear segregation of communication from computation is beneficial for optimization of the system because (a) separate optimization is possible, (b) communication functions can be dealt with abstractly, and (c) custom interface can be used for communication implementation.

The basic concepts and tools to help the reader understand this chapter are the separation of computation and communication, abstraction, optimization, and independence, as shown in Figure 4.1. The first step to be considered is the separation of communication from computation. This enables us to focus on each component—computation and communication. Abstraction has been a useful tool not only for the construction of models, but also in top-down design methodology. The system under design should go through optimization for many performance indexes, such as power, area, or speed. By virtue of the separation of computation and communication, the designer has only to focus on communication optimization. Computation acts as a traffic generator for the purpose of network optimization. The last step is to make the communication model independent and modular. Of course, the independent module should undergo the process of verification and validation. Once it is verified and validated, the network model/implementation can be reused in a different project.

This chapter will give a brief description of the current trends in on-chip communication by contrasting end-to-end communication with point-to-point communication and the bus-based system with bus-independent one. Then, the concept of separating computation and communication will be given. OCP-IP is a good concept to do this work by introducing the socket concept, which will be discussed in Section 4.3. With the socket, the system designer can focus on the communication system design. Section 4.4 will describe a few instances of communication refinement. Lastly, network-based design, that is, a kind of platform-based design, will be briefly explained in Section 4.5.

4.1 COMMUNICATION SYSTEM: CURRENT TREND

Communication systems have evolved because of the increase in system traffic and their complexity. In this section, we will compare these systems from two points of view:

- End-to-end communication versus point-to-point communication
- Bus-based communication system versus general interconnection fabric-based communication systems

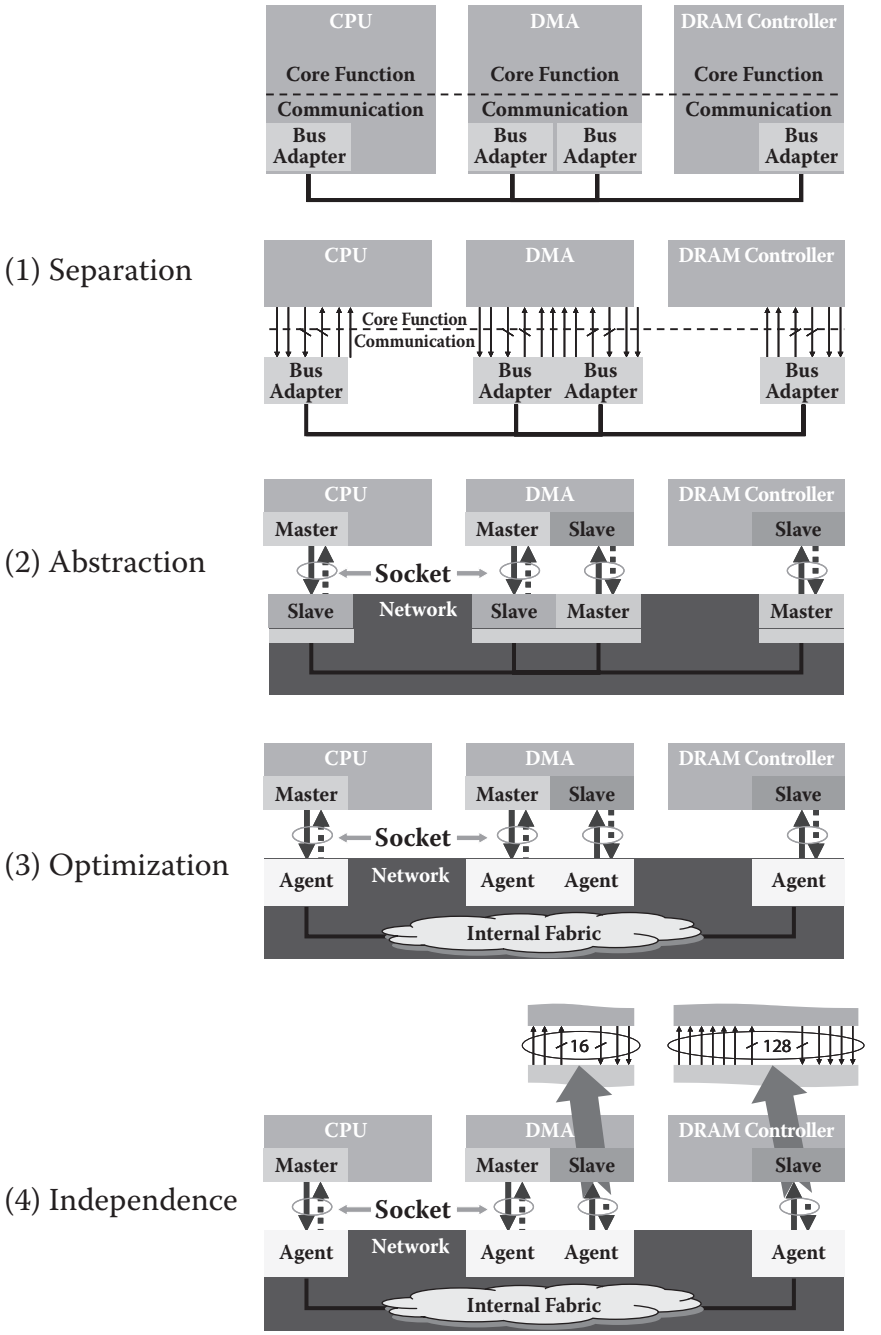


FIGURE 4.1 Tools and methods for system design.

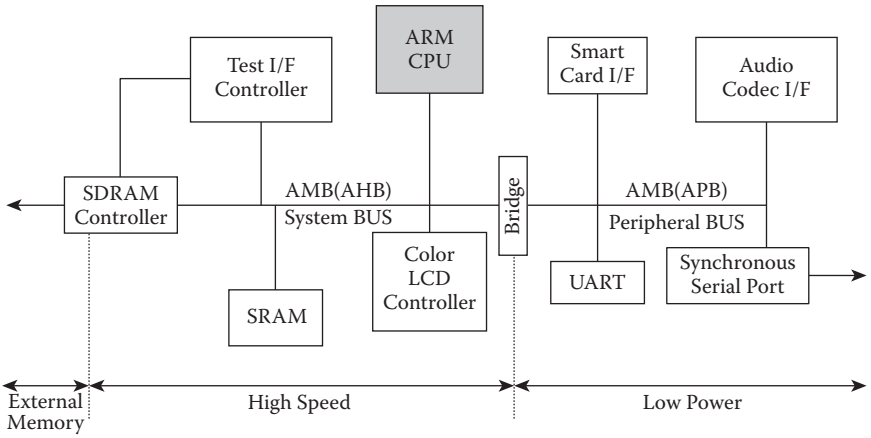


FIGURE 4.2 System with AMBA 2.0 specification.

Figure 4.2 shows an example of a system employing AMBA 2.0 [2], which has been extensively used as a de facto standard. There are two bus systems in the figure: AHB (advanced high-performance bus) and APB (advanced peripheral bus), connected through a bridge. AHB aims to obtain high performance by increasing its operating frequency. This is often used as a backbone high-performance bus. In Figure 4.2, the ARM processor, test interface controller, SRAM controller with SRAM, and the color LCD controller are attached to the AMBA AHB bus. APB is used for a low-power peripheral. It aims at low power with low complexity. In addition, it can cover many peripherals that have diverse speeds and standards. There is another bus named ASB (advanced system bus) that can be used as a system bus, although not shown in the figure. ASB is not equipped with advanced transfer schemes such as burst transfer or split transaction to simplify hardware. Therefore, designers who use the AMBA protocol are free to select AHB or ASB for their system buses according to the requirement and complexity of their systems.

This bus system works fairly well for one master system with a memory controller and several peripherals. However, as the integration scale increases, as in multiprocessor systems, the AHB shows its limitations. In a multiprocessor system, multiple masters can operate concurrently by communicating with slaves or other masters; this is not covered by the AHB specification in AMBA 2.0.

Figure 4.3 is a modification of the AMBA AHB specification for support of multiple masters by introducing a multilayered AHB structure [3]. The key characteristics of a multilayered AHB are for support of multiple buses. It can be constructed by integrating the decoder and mux, as shown in Figure 4.4. Access conflicts with identical slaves at the same time are resolved by arbiters, each of which is connected to its corresponding slave. Besides the configuration shown in Figure 4.4, various structures can be constructed such that the master is connected to a few slaves directly, or a few masters are connected to a few slaves, as shown in the literature cited [3]. This flexibility is the strong point of the multilayered AHB. In addition to this advantage, no modification is required for the modules that were already built for the AHB bus.

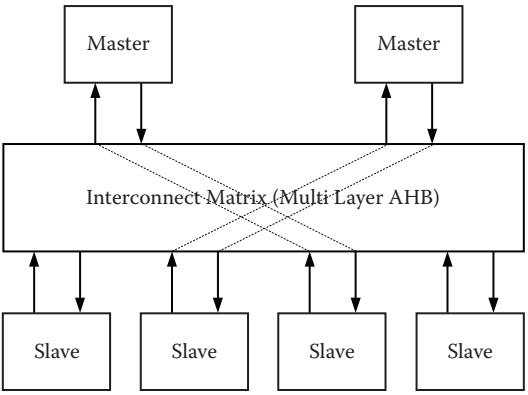


FIGURE 4.3 Multilayered AHB system configuration.

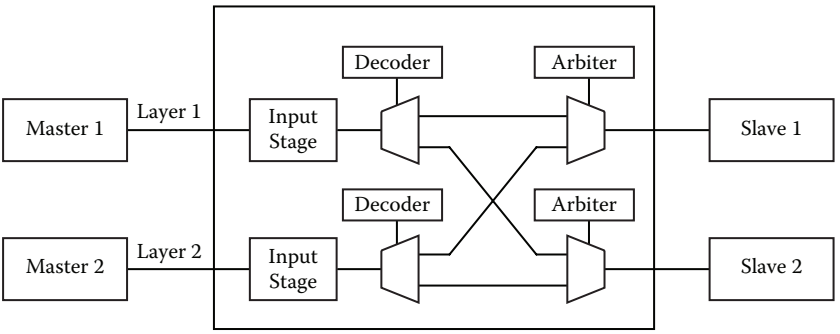


FIGURE 4.4 Implementation example of multilayer AHB.

From the viewpoint of IP designers, developing IPs for specific bus structures implies giving up the chance for the IPs under design to be widely used [4]. The most recent improvement of AMBA specification is bus independent. Figure 4.5 shows a schematic diagram of the AMBA AXI (Advanced eXtensible Interface) [1,5]. The figure shows that communication occurs point-to-point between the master and slave interfaces. Also, it has backward compatibility with the existing AHB and APB, so it can interoperate with the existing AMBA technology [6]. The point-to-point communication shown in Figure 4.5, rather than end-to-end communication, is possible by using five channels according to the AXI specification: read address channel, write address channel, read data channel, write data channel, and write response channel. All the channels behave in handshake manner using VALID and READY signals.

The AXI protocol is burst based. For a READ or WRITE transaction, the master initiates the address along with control information such as burst size, which should not exceed 4KB boundaries. It is not necessary to send every address for burst transfer as in AHB. Instead, AXI specifies only one address of the burst transfer with burst size. In addition, out-of-order transaction is introduced to implement high-performance interconnect, maximize system performance.

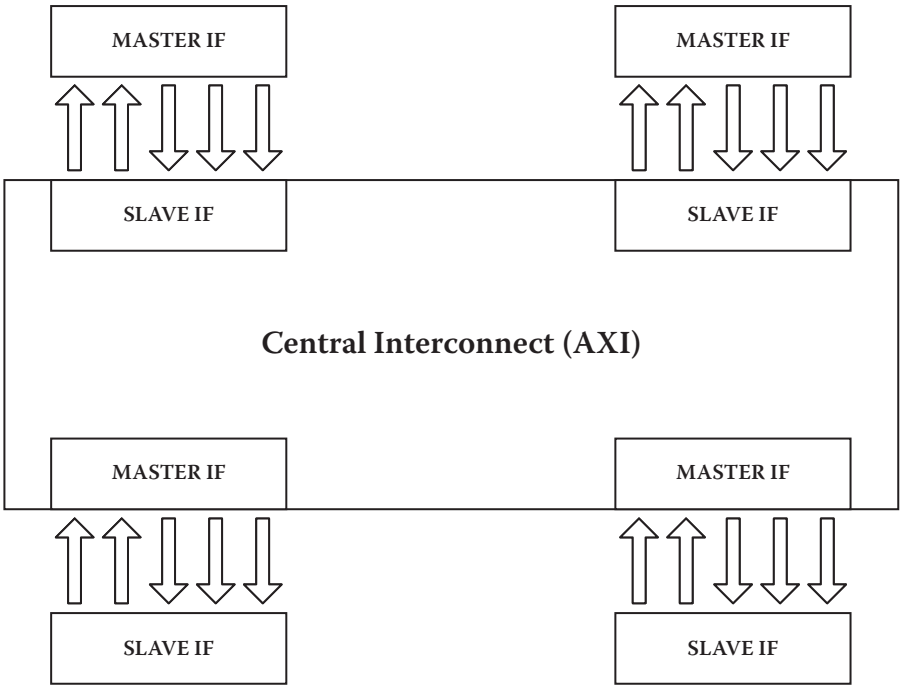


FIGURE 4.5 AMBA AXI interconnection.

There is another important class of interconnection network that employs bus-independent, freely available protocols. It is named Open Core Protocol (OCP) (see Figure 4.6). As indicated in by Smith [4], for an IP or core to be widely reused, the core block should remain unchanged when it is migrated from one system to another. OCP enables the core designer to be free from the actual implementation of the communication system. Instead, they need only care for the OCP interface. From the communication designers' viewpoint, they can freely choose interconnection methods only if they keep the OCP interface. Therefore, OCP acts as a well-defined boundary between core and communication systems.

OCP consists of protocols with signal names, not specifying the internal architecture, so the designer can freely choose any on-chip interconnection that meets the system requirements. Besides the flexibility in selecting interconnection architecture, designers can extend the OCP interface by using sideband signals, which refer to OCP signals used to cover flow control, interrupt, power control, device configuration registers, test modes, and so on. Pipelined transfer can be incorporated to achieve increase of throughput. In addition, a generic flag bus is used to accommodate the signaling needs unique to a specific core [7].

What we can observe from these examples is the separation of the core component design from the interfacing of each module. More system designs follow the communication-centric design and integration rather than the core-centric design. By making each design process independent and focusing on each of them, optimization

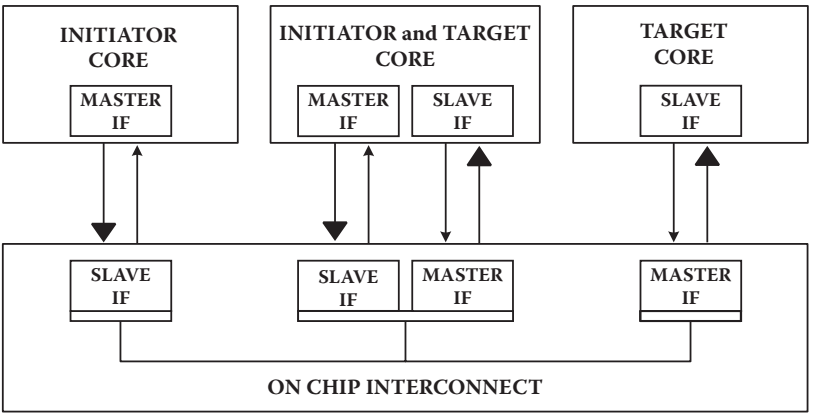


FIGURE 4.6 OCP configuration.

can be easily obtained with well-defined design flow. In this chapter, we will focus on the communication-centric design.

4.2 SEPARATION OF COMMUNICATION AND COMPUTATION

As long as there was enough room for improvement on processors, relatively little interest was bestowed on the communication system. However, the method of sending and receiving messages among processing elements has been a bottleneck in achieving maximum performance, possibly, because of large integration for digital convergence, complexity of the target system, and process technology. No one would deny that the communication fabric of SoC will become more and more important. However, it is not clear how we can separate communication from computation and design the optimum communication architecture.

Before we look into the separation of communication from computation, we should first define computation and communication. The following definitions clearly explain the difference between computation and communication:

- Computation
  - Activity to change the inside states of a module
- Communication
  - Activity to change the outside states of a module

In classical system design, much focus was put on designing modules that satisfied performance requirements, assuming basic communication structures such as bus architecture or using interface synthesis at back end.

Figure 4.7 shows the most abstract picture of a networked system. It describes the situation in which modules can send and receive messages, not knowing the exact implementation of the network. However, each module should be equipped with the relevant wrapper and driver for each specific network instance. Figure 4.8a shows a simple picture of a networked system, focusing on the module structure. The processing element (PE) can be abstracted as having states and functions inside.

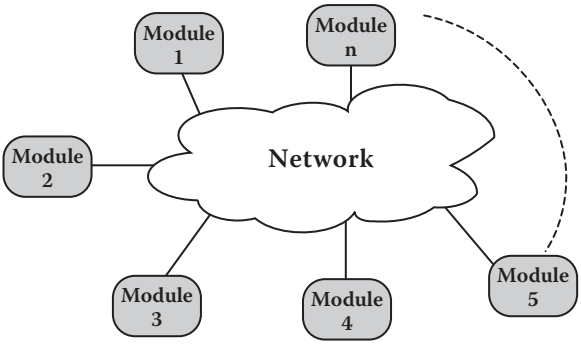


FIGURE 4.7 Networked system.

Functions can be further classified into pure behavior (or computation), input function, and output function, as shown in Figure 4.8b. OFunc (output function) sends data to the network, and IFunc (input function) reacts to transactions from the outside. Then, the IFunc and OFunc functions are defined for the specific target network. By developing the state machine for each transaction for IFunc and OFunc as in Figure 4.8c, we can obtain a networking module, a so-called "plug." Then, the network gets equipped with "sockets" through which PEs can communicate with each other. In the literature, many expressions such as plug, socket, wrapper, and network interface are used together, and more detailed explanations of these terms will be given in the latter part of this chapter with specific examples.

4.3 COMMUNICATION-CENTRIC SOC DESIGN

In this section, we will first discuss the socket, which enables the overall design flows into the core-centric and, communication-centric design and integration.

4.3.1 Overview

Figure 4.9 depicts the concept of the socket. A system consists of a computation component and a communication system, with a connection module between them. The connection can be between the computation block and the computation component, or between the communication system and the communication system, or between the communication system and the computation component. System design is nothing but designing the computation component, the communication system, and the connections. To take all these into consideration together is quite challenging. Therefore, divide and conquer is a good strategy in the design of computation and communication systems. The socket-based design is an example.

In Figure 4.9, the computation engine and the communication system are separated by wrappers and the socket. Different implementations of the computation engine can be integrated with preexisting, preverified communication systems only if the core designers use a standardized socket and modify the wrappers. To describe it differently, the socket itself is isolated from the implementation of the computation component and communication system by the designers. The core designer and the communication designer

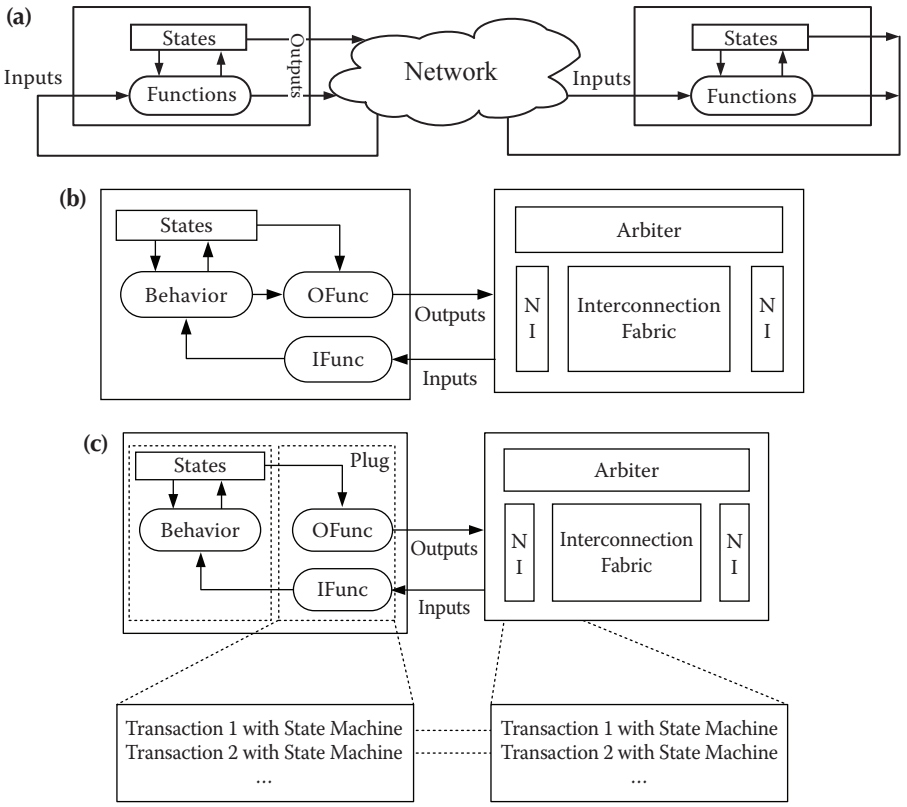


FIGURE 4.8 Separation of communication and computation. (a) General picture of networked system (b) separation of function into behavior, input function, and output function (c) interface modularization.

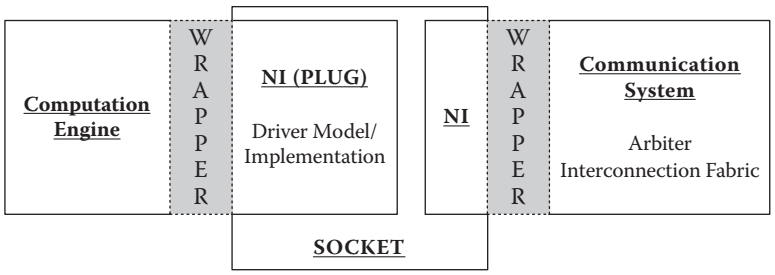


FIGURE 4.9 Socket: Boundary between computation and communication.

can design their parts so that they are compatible with the socket. For this purpose, the signal and protocol should be defined precisely, and can be extended if necessary.

From the designers' viewpoint, the socket enables them to communicate with other components through the communication system under design. However, if the protocols or services provided for the socket are rigid and cannot be altered for optimization, bottlenecks can happen at the socket. Therefore, socket definitions should be able to accommodate flexibility to be tuned for optimization (e.g., address/data width configuration, parameters for protocol, and sideband signaling). In addition, it should provide different abstraction models for the purpose of verification. Usually, core designers start from a functional specification, which is possibly executable. Then, they might want to test their specification to check functional correctness. When the functional communication system is also ready, the overall system is executable with a proper socket model.

4.3.2 OCP-IP: Socket Abstraction

Communication-centric systems gain interest in industry because communication limits the total performance of a chip as the critical bottleneck. OCP-IP have proposed a theoretically well-defined practical framework in cooperation with various companies and universities. At the highest level of abstraction, Transaction-Level Model (TLM) is proposed by OCP and Open SystemC Initiative(OSCI), which consists of three levels, as shown in Chapter 2. We will look into more details of the implementation-friendly layer. As described in the literature cited [8], the following properties are the key to understanding the OCP 2.2 specification:

- Point-to-point synchronous interface
  - Unidirectional signals that work on the rising edge of the clock
  - Simplification of timing analysis, physical design, and general comprehension
- Bus independence
  - Not tied to a specific bus architecture
  - Much room for customization: device selection methods and arbitration schemes
- Commands
  - Two basic commands: Read and Write
  - Five command extensions: WriteNonPost, Broadcast, ReadExclusive, ReadLinked, and WriteConditional
- Address/data
  - Configurable address and data width for area-efficient implementation
  - Word size of the OCP fixed by data width
  - Possible to transfer less than a full word by incorporating byte enable
- Pipelined transfer support
- Burst transfer support
- In-band information support
- Tags
  - Control of ordering of responses: possible for responses to be out of order with tags

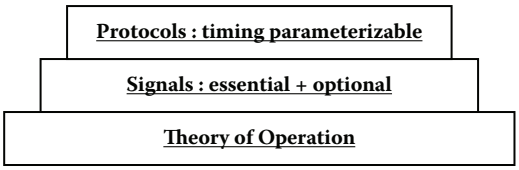


FIGURE 4.10 OCP-IP concept.

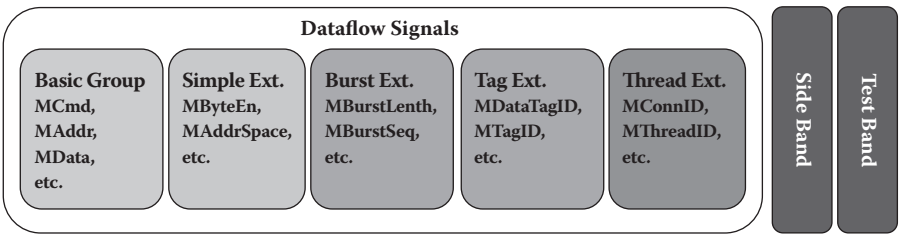


FIGURE 4.11 OCP-IP signals.

- Thread and connections
  - Transactions within different threads with no ordering requirements, and independent flow control from one another
  - Within a single thread of dataflow, all OCP transfers to remain ordered, unless tags are in use
- Interrupts, errors, and other sideband signaling

With the theory of operation, signals and protocols are defined by abstraction of the communication system, as briefly shown in Figure 4.10.

Then, OCP 2.0 specification defines three types of signals: dataflow signals, sideband signals, and test signals. Also, dataflow signals can further be categorized into basic signals, simple extension, burst extension, tag extension, and thread extension, as shown in Figure 4.11.

We can classify signals according to their roles: request signal, response signal, and handshake signal. The relationships among them are shown in Figure 4.12. Signals initiated by the master get to start with *M*, and signals controlled by the slave begin with *S*. Every request signal is initiated by the master, and all the response signals are controlled by the slaves. Readers can find the full description of OCP signals in the OCP specification [8]. Based on these OCP signals, protocols can be precisely

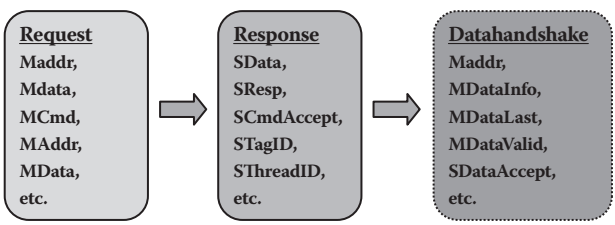


FIGURE 4.12 Signals: category by role.

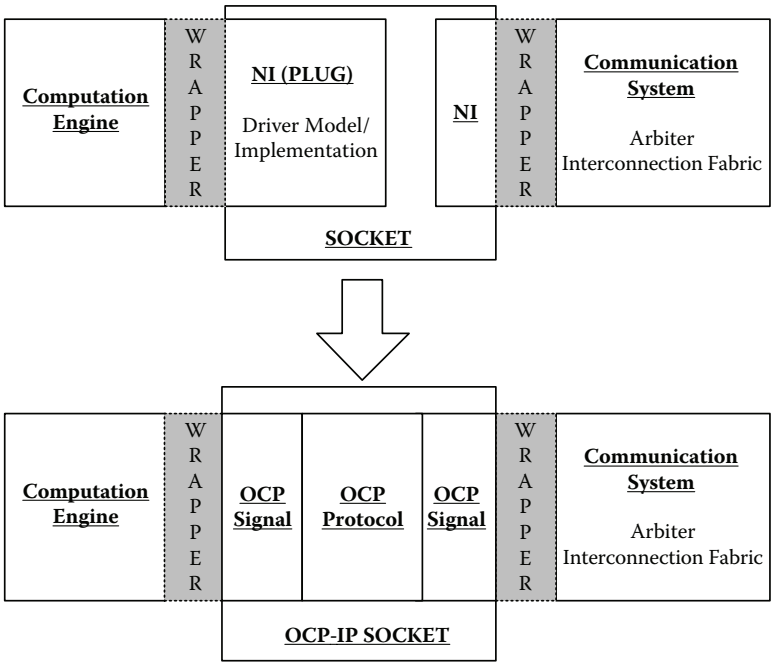


FIGURE 4.13 OCP-IP socket.

defined by OCP specification, and OCP users can design their own modules easily. OCP protocols can include simple read/write, request handshake with/without separate response.

The strong point of the OCP-IP standard is its flexibility; that is, the communication system can be freely selected or designed. If the signals and protocols under design comply with the OCP standard, it can be integrated with any other module that is also compatible with OCP. Figure 4.13 describes the compatibility mechanism. The OCP-IP socket itself is independent of the computation component and communication system, and it provides abstraction of communications with signals and protocols. Then, the computation component and communication system can be designed by making use of the signals and protocols defined by OCP-IP. As a result, IPs and communication systems both compliant with OCP-IP, can be freely combined as shown in Figure 4.14.

Reusability is also enhanced by introducing the OCP-IP protocol as a communication standard. However, it is a by-product of the standardized protocol, because reusability can be obtained also by any other de facto communication standard. Reusability is a matter of the number of supported IPs and the vender's willingness to support the protocol, rather than a technology issue.

4.4 COMMUNICATION SYNTHESIS

By completely separating the computation and the communication, the communication system designer need focus only on the interconnection system that complies

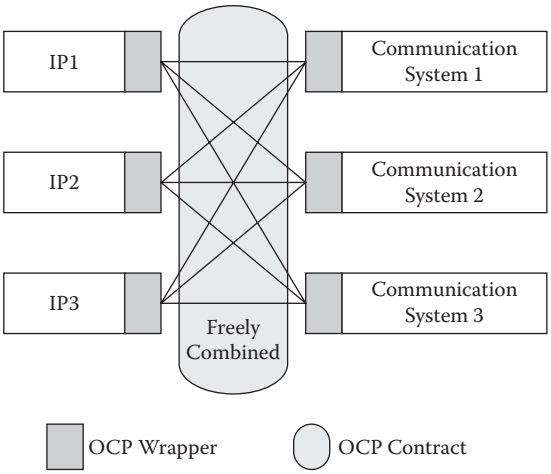


FIGURE 4.14 Integration of IPs and communication systems through OCP socket.

with the socket protocol. In this section, a high-level communication design using Unified Modeling Language (UML) will be introduced in brief, followed by a work on automatic/semiautomatic communication refinement.

4.4.1 High-Level Communication System Design

To decide on a communication system, a baseline model is necessary for analyzing the traffic. Such a model should be equipped with generic architecture. There are many high-level methods to model/evaluate initial design, such as Matlab, SystemC, SpecC, and so on. We have introduced UML to take advantage of many useful diagrams and reusability. Figure 4.15 is an example for the evaluation of a baseline network model. The system consists of one main controller and eight clusters,

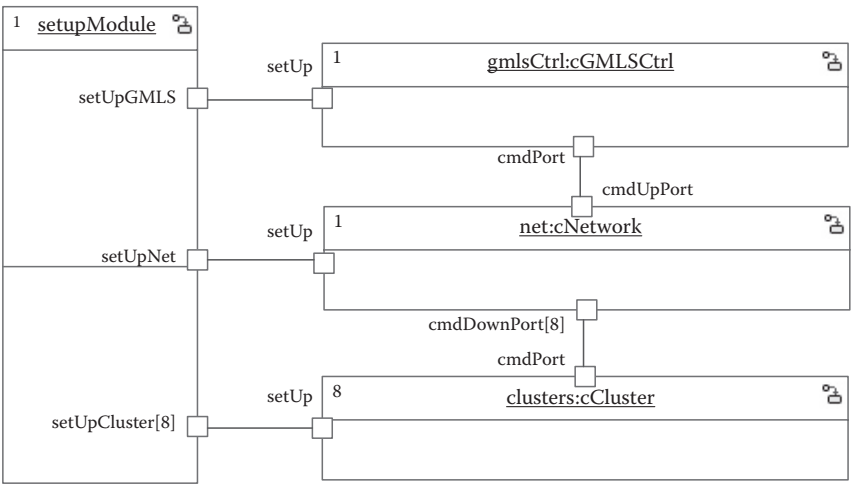


FIGURE 4.15 Overall systems.

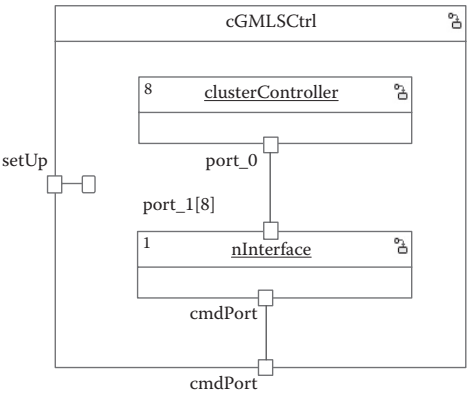


FIGURE 4.16 Decomposed model for *cGMLSCtrl*.

connected by the baseline network. The main controller is *gmlsCtrl*, and the eight clusters are grouped as *clusters*. The main controller only distributes jobs to the clusters through the network. When a cluster finishes a job, it sends back a DONE message to the main controller. Then, the main controller distributes a new job. Clusters perform jobs at the level of functions/tasks. They are connected through a generic network named *net. setupModule,* an agent for the preparation of simulation. Then, *cGMLSCtrl* is again decomposed into the controller and network interface, as shown in Figure 4.16. *nInterface*, which is an object to model the network interface, plays the role of a proxy between *clusterController* and the physical network. This model assumes that controller *cGMLCtrl* is just a job distributor and the actual computation for the application is performed by the clusters.

Figure 4.17 is the state machine of *nInterface* object. The right-hand side of the machine is used for id/address mapping of *cGMLSCtrl* objects to the network, and

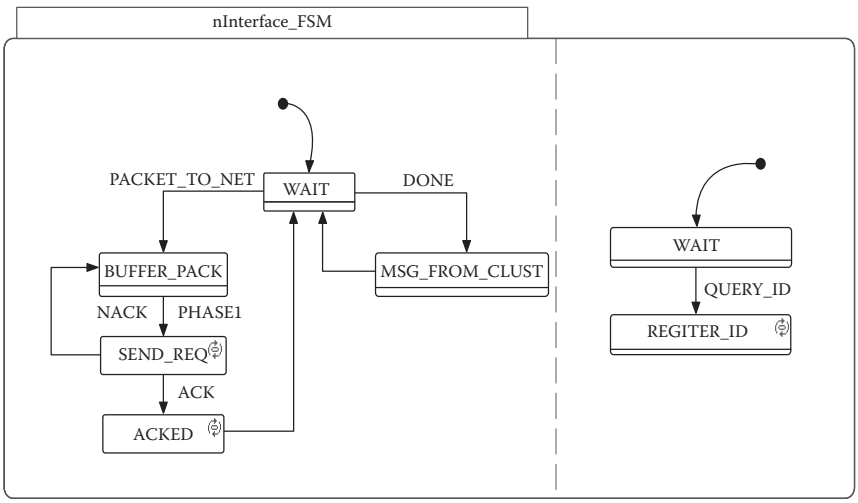


FIGURE 4.17 State machine for network interface at controller side.

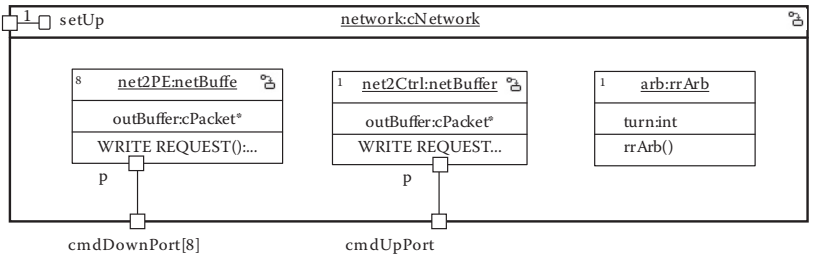


FIGURE 4.18 Decomposed model of network.

the left-hand side is for sending and receiving messages. If there is a packet to be sent to a network, it goes to the *BUFFER_PACK* state, where the packet is generated by encoding data from *clusterController*. At *PHASE1*, the machine sends a *WRITE_REQUEST* to the network. If it receives *NACK*, it should wait for the next cycle. Otherwise, it sends the packet to the network and then gets back to the *WAIT* state. When it receives a *DONE* message from the network, it processes and again goes back to the *WAIT* state. Then, the high-level network model is required to perform the following tasks:

- Message buffering
- Message routing
- Arbitration

The *net* instance of the *cNetwork* class is decomposed into the buffers at the input/output port and the arbiter, as shown in Figure 4.18. The *net2PE* and *net2Ctrl* are instances of the same class, *netBuffer*, and are buffers that are equipped with the state machine for buffer management. In this example, the network arbitration scheme is assumed as round robin.

The state machine of the *netBuffer* class is shown in Figure 4.19. Basically, a state machine of UML is hierarchical and concurrent. The figure shows two concurrent state machines collaborating to achieve the goal. At the *WAIT* state, which is the initial state of the machine, it receives a *WRITE_REQUEST* event from outside the network. It checks if there is enough room for the message. If there is no room, it returns to the *WAIT* state, with a *NACK* message sent back to the initiator of the *WRITE_REQUEST*. If buffering is possible, it sends *ACK* to indicate that the initiating component can really send the message. Then, the buffer receives the message and pushes it to the input buffer.

The network should transfer data at a specific instance of time. Figure 4.20 shows the message transfer state machine. The upper six states of the left-hand side of the machine, INIT, PRE1, PRE2, PRE3, PRE4, and PRE5, are implementation-specific components. They are used to prepare the simulation by registering addresses of external entities to the network for message routing. They are used to fetch id's or addresses from external entities. The *cNetwork* stores the list (of addresses, port pointers) for message routing, which corresponds to a routing table.

This state machine has two events for synchronization, which is shown at the right half of the state machine. *PHASE1* corresponds to the rising edge of the clock

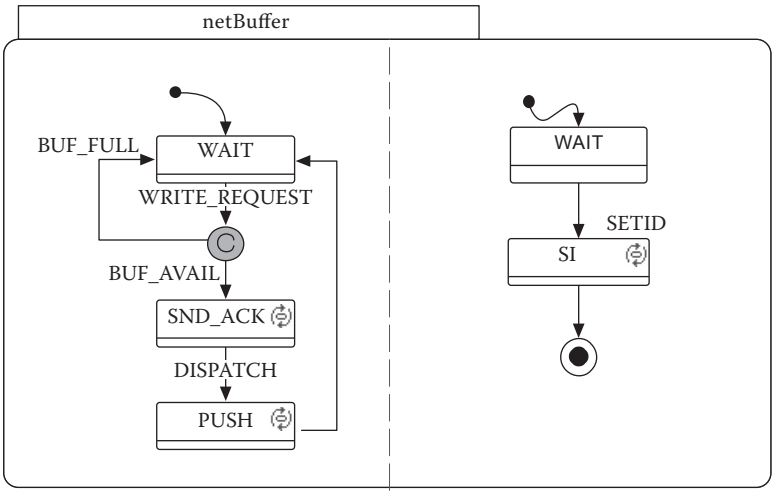


FIGURE 4.19 State machine for buffer.

and *PHASE2,* to the falling edge. *PHASE1* is generated at the *P1* state and *PHASE2* at *P2*. Every active component sends messages to the network at *PHASE1*. The network transfers data at *PHASE2* to avoid conflict or indeterminancy. The bottom three states at the left half of the state machine performs the actual message routing. At the *WAIT* state, whenever *PHASE2* occurs, it transfers messages, checking if there is any imminent message at each input buffer and if there is enough space at the buffer of the destination port. At the *TRANSFER* state, it checks and tags the possibility of transferring data from buffer to buffer, executing arbitration if necessary. Finally, at the *ROUTE* state, messages tagged as transferrable are routed. It just moves the transferrable messages from one buffer to the other buffer.

This high-level simulation of the baseline communication model is very useful for lower-level communication system designs because it gives the traffic pattern between entities, the potential traffic bottleneck, conflict possibilities, desirable arbitration schemes, and so on. In addition, we can obtain a timed model directly by specifying the number of cycles for each component, including latency determined by the packet length.

4.4.2 Communication Design Methods

From now on, we will study how to design a real communication implementation from abstract specifications automatically or manually. For example, an automatic network generation, which consists of the network and link designs, on top of the SpecC framework, has been proposed [9,10]. The framework transforms the abstract specification to the complete structural net list. In this section, we would like to focus only on communication implementation. Figure 4.21 depicts the overall framework for the network design. The design gets three inputs: input architecture model, design decision, and network protocol database. The input architecture model assumes an architecture into which applications have been partitioned and mapped. Therefore, the model consists of computational components linked with

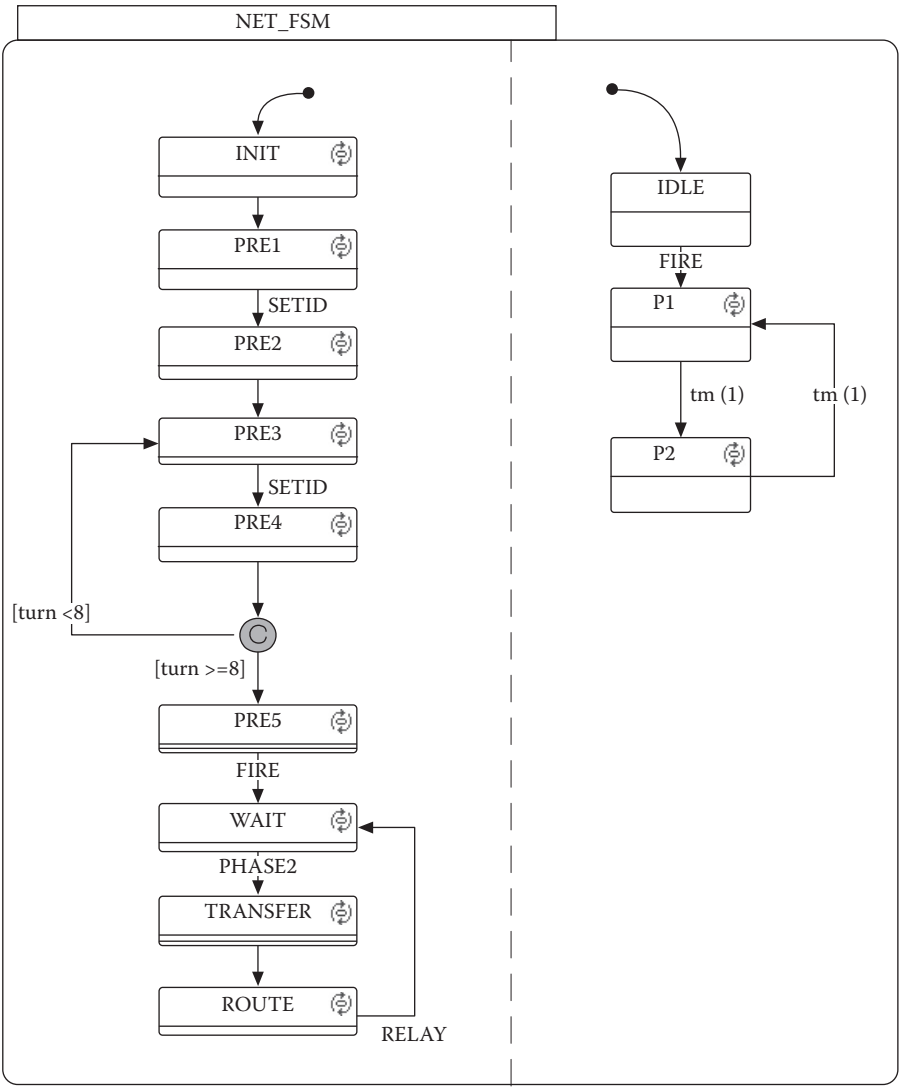


FIGURE 4.20 State machine for message routing.

abstract message-passing channels. Design decision includes the physical building blocks of the network such as bus allocation, selection of communication elements, connectivity between components and busses, mapping of abstract communications to busses and protocol selection. Finally, the network protocol database contains a set of communication functions such as transducers and bridges.

The network design is to determine the presentation, session, transport, and network layers. Table 4.1 describes the roles of the respective layers in more detail. The presentation layer performs data formatting, such as conversion of abstract data type to blocks of ordered types defined by lower layers, considering data type conversion, or endianness, as shown in Figure 4.22.

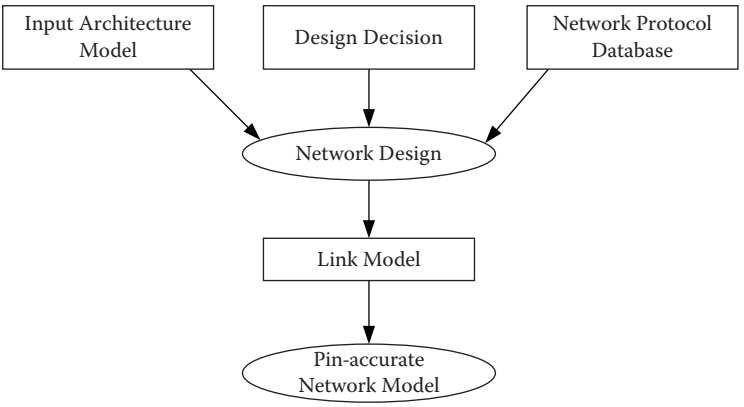


FIGURE 4.21 Network design. (From Rainer Doemer, Daniel D. Gajski, and Adnreas Gerstlauer, SpecC Methodology for High-level Modeling, IEEE/DATC Electronic Design Processes Workshop, 2002.)

TABLE 4.1
Layer for Network Design

Layer	Role
Presentation	Conversion of word-level/abstract data to ordered bytes
Session	Management of connection between entities for synchronization
	Message multiplexing transferred through a number of end-to-end sequential streams
Transport	End-to-end flow control/error correction
Network	Message routing through point-to-point interconnection

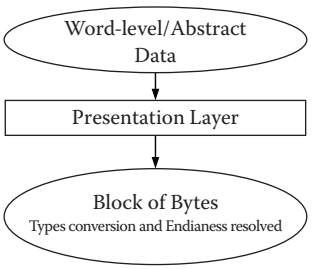


FIGURE 4.22 Presentation layer.

There can be many channels between different processing elements, where a channel implies a message path. The session layer merges these different channels into a number of sequential message streams by investigating the features of the channels as follows. For two different channels to be merged, they should never transfer messages in parallel as specified in the input architecture model. For example, A1, A2, and A3 and B1, B2, and B3 in Figure 4.23a run sequentially. This implies that messages sent by B1 will reach only A1. The same rule applies to A2/B2 and A3/B3. Therefore, three channels, cA, cB, and cC, can be merged into one channel, cABC, as shown in Figure 4.23b, because there is no possibility of a mix of message channels. On the contrary, Figure 4.23c shows a situation in which B1, B2, and B3 can execute in parallel. In this case, cA, cB, and cC cannot be merged because, if they are merged, PE1 and PE2 would not be able

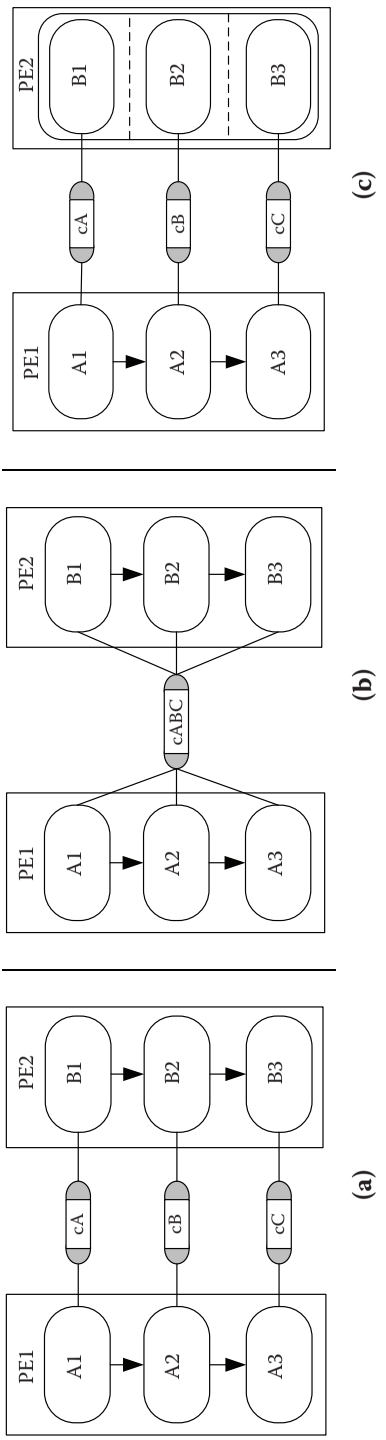


FIGURE 4.23 Channel merging.

to differentiate their message sources. This can be analyzed more systematically by using (a) the conflict graph, whose vertex is the channel and edge the conflict between channels, and (b) the graph-coloring algorithm. Vertices with the same color imply that they can be merged into one channel. To avoid the NP-complete feature of the graph-coloring algorithm, greedy heuristics have been applied in this work [9].

For the on-chip interconnection, the communication medium is assumed to be error free. In the case of Internet, error probability in the channel is so high that the transport layer should perform various error detection/correction strategies. However, on-chip interconnection has a low error rate. Also, hardware cost for handling errors is very high. Therefore, the transport layer of the on-chip network does not provide error detection/correction strategies. The main role of the transport layer is data splitting to reduce the buffer size.

The network layer routes and connects heterogeneous networks. To this end, bridges and transducers are inserted if necessary. These are used to connect incompatible and different buses. The bus bridge always has two specific ports/interfaces. It behaves as a master on one side and slave on the other. It knows the protocols of both sides of the bus, so that it blocks the slave side bus protocol whenever needed instead of buffering the total bus protocol to its master side.

Unlike the bridge that should act as a master on one side and slave on the other, a transducer can act as a master and slave on each of its sides. In addition, the transducer buffers all the transactions from one side before starting those of the other. However, a transducer cannot act in between handshake communication. Here, the communication is split into three independent consecutive transactions: one to send the request, another to indicate Ack/Nack, and the third to send data. To put it differently, this transforms an end-to-end communication between PEs to point-to-point transaction between PEs and transducers. To resolve deadlock or confusion, separate buffers are placed for each direction.

In a work by Dongwan Shin et al. [9], JPEG, Vocoder, Baseband, and MP3 applications were introduced with initial architecture models consisting of Motorola DSP56600 processors, Motorola CodFire processors, and ASIC for various buses. It was reported that the network design gets faster by about 798 to 3385 times in terms of the code sizes modified from the initial architecture. For example, the code size describing the architecture for baseband application was 19,754 and the resulting architecture model had 20,227 lines of code. They computed the number of lines that had been inserted to or deleted from the initial architecture. It was estimated that manual design required 119.5 hr, compared with automatic design, which took only 0.75 sec.

A top-down approach that refines a synchronous communication model into a Network on Chip architecture has been introduced [11], which is based on the earlier framework for system design [12]. The methodology assumes multiprocessor architecture, as in Figure 4.24. They identified four kinds of communication patterns: producer–consumer, peers, client–server, and multicast models. Their features are summarized in Table 4.2. The producer–consumer communication pattern will be explained in detail as an example.

The specification model for Figure 4.24 is based on the synchronous model of computation, explained in Chapter 2. This kind of model is too abstract for treating

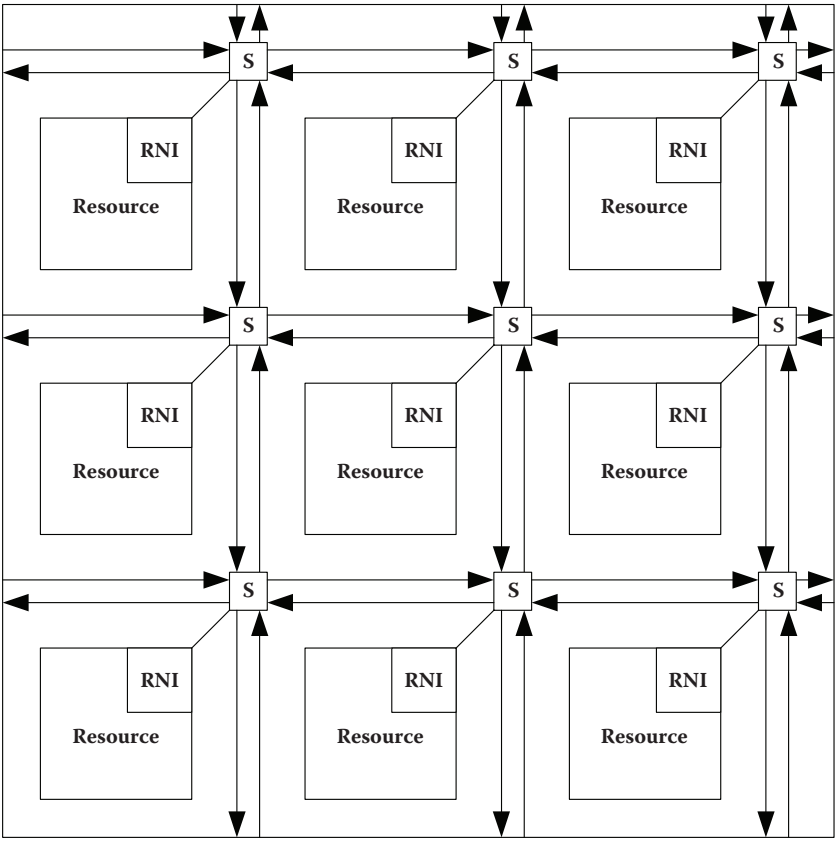


FIGURE 4.24 Architecture model: Multiprocessor system with mesh structure.

TABLE 4.2
Communication Model for *ForSyDe*

	Description
Producer–consumer	One-to-one relationship Sender is always sender and receiver is always receiver
Peer	One-to-one relationship Both processes can be sender/receiver at different instances of time
Client–Server	Many-to-one relationship One server provides a service or multiple services to one or more clients
Multicast	One-to-many relationship One master sends message to a group of recipients

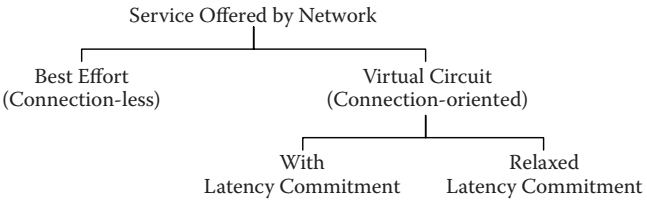


FIGURE 4.25 Types of service offered by switch.

implementation details such as buffer sizes because it assumes first in, first out (FIFO) queues between processes. This can be the starting model leading to the explicit implementation. The NoC platform in Figure 4.24 is a mesh structure, which connects resources or processing elements with switches.

The services offered by the switch fabric are the network communications, and RNI (Resource Network Interface) connects each resource with the communication fabric. The types of services offered by the network are shown in Figure 4.25. In connectionless communication, messages should be reordered because they may not arrive at the destination in order. On the contrary, connection-oriented communication ensures resource-to-resource connectivity, which guarantees message order and bandwidth. In case of connection-oriented virtual circuit communication, how to reduce latency is the crucial issue, and minimum and maximum latencies are usually sought. Virtual circuit with relaxed latency has the worst-case delay value with maximum delay latency.

An outline of NoC communication design is shown in the Figure 4.26. GALS (globally asynchronous, locally synchronous) is assumed; therefore, processes in synchronous communication can use different clock frequencies.

As shown in Figure 4.27, two interfaces are used to deliver the message from cycle domain f_1 to cycle domain f_2. First, the ratio between f_1 and f_2 ($f_1/f_2 = n/m$) is calculated. Thus, I_{rn}, the interface from the resource to the channel, is n times up-sampling, and I_{nr}, the interface from the channel to the resource, is m times down-sampling. When f_2/f_1 is 3/2, a three-times up-sampler and a two-times down-sampler

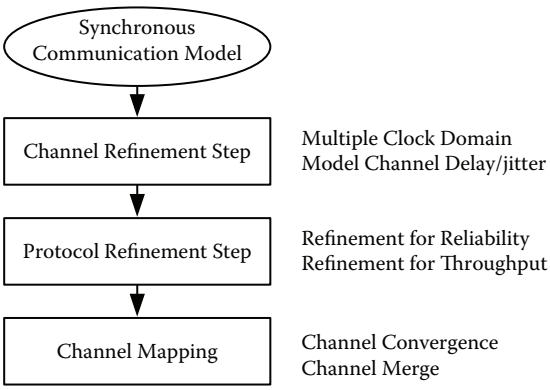


FIGURE 4.26 NoC communication design.

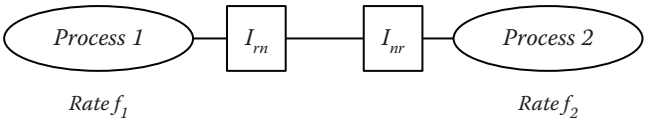


FIGURE 4.27 Interfacing between different clock domains.

are necessary, as shown in Figure 4.21. It is assumed that the frequency of the sending process is always lower than that of the receiving process.

The next step is to model channel delay and jitters. This is abstracted by adopting a stochastic process that generates random delay within maximum and minimum value, $D_{[min, max],}$ as in Figure 4.29.

Error probability of the channel is so low and the hardware cost for error management is so high that errors are hardly encountered. Therefore, the subsystems for channel error handling are not described here. However, fast-running message senders can overflow the buffer. Therefore, a subsystem for this kind of error is crucial. To solve this problem, the sending processor always waits for the ACK or NACK signal for the data just sent. With these signals, the sending processor can control the outgoing messages so as not to overload the network. Figure 4.30 shows a model that has been refined from the one in Figure 4.28. A buffer at the sender side is incorporated to store messages that have to be sent. The P′ process fetches data from

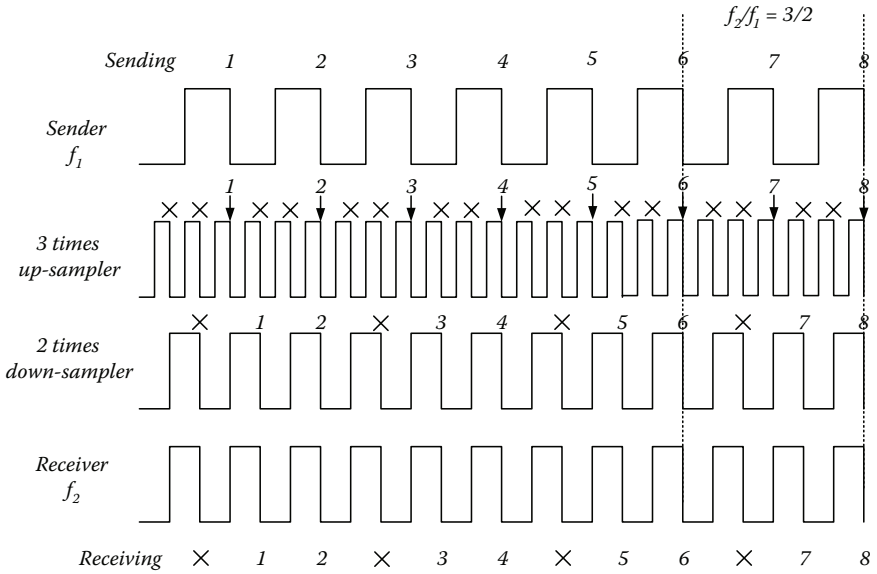


FIGURE 4.28 Timing diagram for different clock domains.

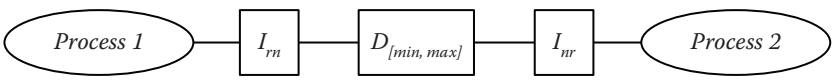


FIGURE 4.29 Channel Delay/Jitter model.

the buffer, sends it, and waits for the return message. By incorporating these buffers, the P' process and feedback look, buffer overflow can be detected and controlled.

The next step to be considered is the units of data to be acknowledged, because acknowledging every packet is inefficient and would make it impossible to meet the throughput requirement. The size of the message that corresponds to one acknowledgment is determined by considering the channel capacity. This can easily be implemented by introducing counters for the sending/receiving processes. The sending process keeps on dispatching packets until the value of the counter reaches the prespecified value. When the counter value equals this, the sending process waits to receive the acknowledgment signal. That is, a number of packets are sent before the receipt of the acknowledgment signal, rather than an acknowledgment signal being given for every packet. The exactly opposite procedure is followed by the receiving process. The channel capacity determines how many packets will be treated as a unit of transfer.

The last step is channel mapping, including channel convergence and channel merging in the RNI. Channel convergence implies sharing the virtual circuit with two different logical channels, if it does not violate the throughput requirement. Channel convergence reuses the virtual channels to reduce channel setup. On the contrary, channel merge brings messages together into one payload. This is possible by making the message format transparent to the receiving processes. For this purpose, *merge* and *split* processes are introduced. The merge process constructs a packet by assembling different messages, like channel merge, and split, the reverse of the merge process, detaches each message and transfers it to the relevant recipient.

4.5 NETWORK-BASED DESIGN

In the previous section, socket-based design is introduced as a method of using a prespecified standard interface to provide electrical, logical, and functional connections between cores [13]. In this section, we will take a look at network-based design.

Figure 4.30 shows the network-based system design methodology. Communication and computation separately defined and designed with their own design flows. For the communication design, the network library is explored under the communication requirement. Thus, we obtain a set of candidate network designs, which include the simulation model, description of protocols, IP blocks such as bridge, and regulations that should be followed while designing the computation block. If there are no communication systems that meet requirements, a completely new network, including the interconnection fabric, bridge, and so on, should be designed. We can also customize the best option available in the database. The same points apply to the IP design flow. Two integration steps are incorporated: one for integration of OCP-IP

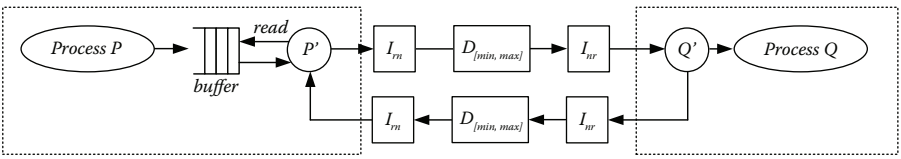


FIGURE 4.30 Buffer insertion.

compatible components and the other for integration of incompatible interfaces. The former, integration of OCP-IP compatible components, consists of designing or modifying wrappers that will be inserted between the computation and OCP-IP drivers. The latter, and last step of the design flow, is to make bridges/transducers between components with completely different signals and protocols.

A network system can be decomposed into network topology, with protocol defined on the topology. Topology defines connectivity between each port and interface, or between ports or interfaces and internal blocks, such as a bridge. Protocol defines the ordering of data/control signals. Therefore, improving a network system implies either introducing new topology or developing better protocol on existing topology, or both. However, the physical property of communication media limits the bandwidth. In such a case, new connectivity can solve the problem, which is the modification of existing topology or, a completely new topology.

For example, Figure 4.31 shows how we can construct a network with a given system implementation. By storing and elaborating the network itself, and network interface that supports the transactions for the network, we can obtain standardized network architecture, and a sample plug, which can be put into the network. The front end of the network acts as a socket, which provides various services to the components that has enough plugs. Now, we have only to develop a computation engine, and customize the preexisting wrapper or design a wrapper at high-level abstraction, which is followed by synthesis. Thus, the socket-based design is reduced to the development or customization of wrappers.

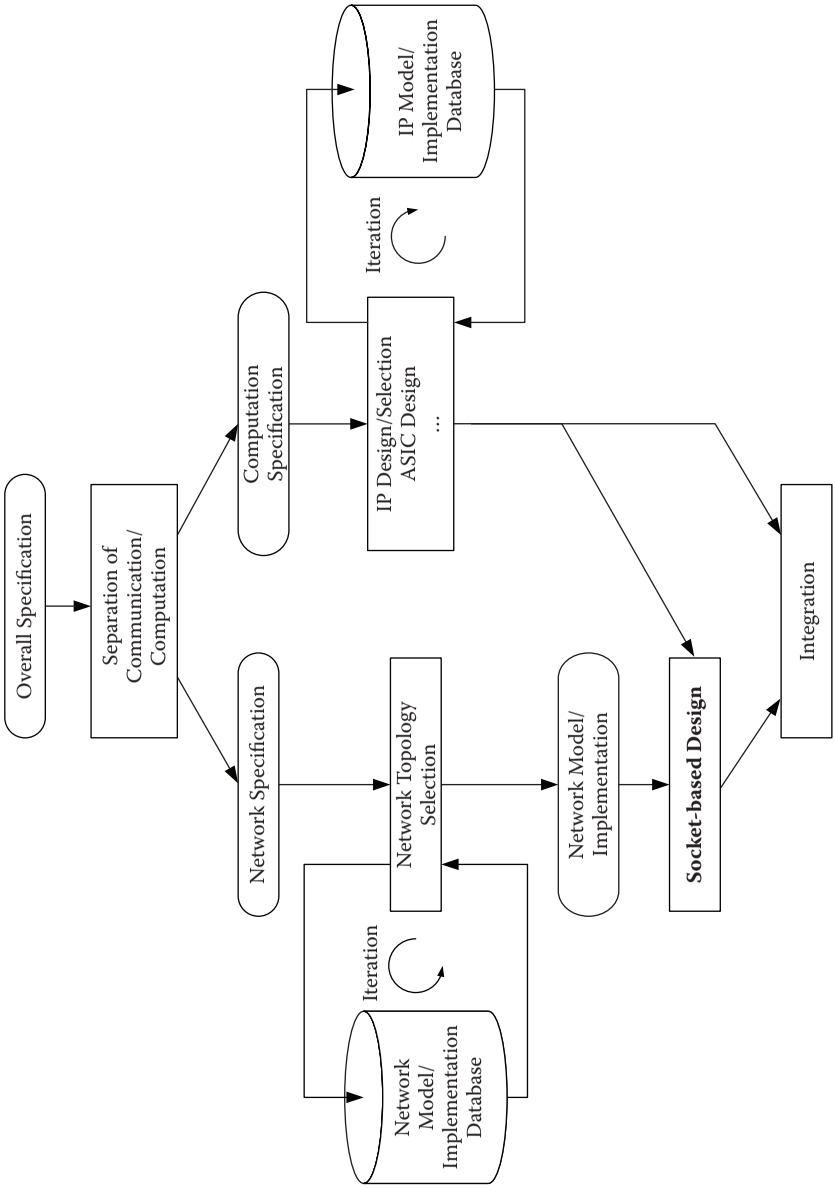


FIGURE 4.31 Network-based design.

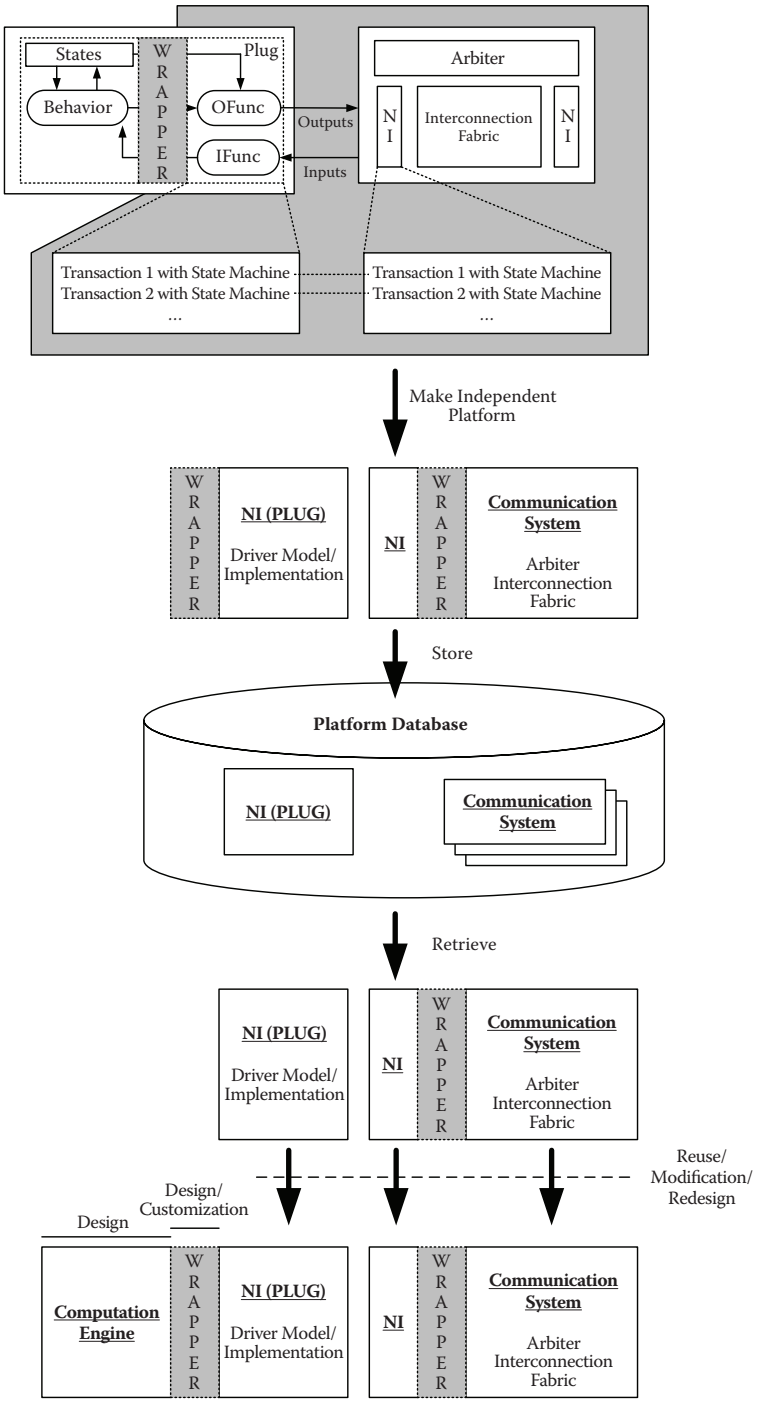


FIGURE 4.32 System design.

REFERENCES

1. Giovanni De Micheli and Luca Benini, *Networks on Chips,* Morgan Kaufmann.
2. AMBA 2.0 Specification, http://www.arm.com/products/solutions/AMBA_Spec.html.
3. Multilayer AHB, http:// www.arm.com/pdfs/DVI0045B_multilayer_ahb_overview.pdf.
4. Ed Smith, Bus Protocols Limit Design Reuse of IP, *EE Times*, May 15, 2000.
5 AMBA AXI Protocol, http://www.arm.com/products/solutions/axi_spec.html.
6. Mick Rosner and Darrin Mossor, Designing Using the AMBA 3 AXI Protocol—Easing the Design Challenges and Putting the Verification Task on a Fast Track to Success, Synopsys.
7. Socket-centric IP Core Interface Maximizes IP Application, http://www.ocpip.org/data/wp_pointofview_final.pdf.
8. OCP 2.2 Specification, http://www.ocpip.org/.
9. Dongan Shin et al., *Automatic* Network Generation for System-on-Chip Communication Design, *Proc. 3rd IEEE/ACM/IFIP Int. Conf. on Hardware/Software codesign and system synthesis*, 2005, pp. 255–260.
10. Rainer Doemer, Daniel D. Gajski, and Adnreas Gerstlauer, SpecC Methodology for High-level Modeling, IEEE/DATC Electronic Design Processes Workshop*,* 2002.
11. Zhonghi Lu et al., Towards Performance-oriented Pattern-based Refinement of Synchronous Models onto NoC Communication, in *9th Euromicro Conf. on Digital System Design (DSD 2006),* August 2006.
12. Ingo Sander and Axel Jantsch, System Modeling and Transformational Design Refinement in ForSyDe, *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol. 23, No. 1, January 2004, pp. 17–32.
13. Jari Nurmi, *Interconnect-Centric Design for Advanced SoC and NoC*, Kluwer Academic Publishers.

Part 2

NoC-Based Real Chip Implementation

5 Network on Chip-Based SoC

As the chip scale grows, current System on Chip (SoC) designs generally incorporate a number of processing cores to answer high performance requirements with reasonable power consumption [1–4]. This design methodology has the advantage of achieving high performance with moderate design efforts because, once a processor is designed and verified, it can be replicated and reused. However, integrating a number of processors on a single chip does not necessarily mean an SoC design. Depending on the applications, SoC also requires the integration of numerous peripheral modules, such as on-chip memory, external memory controller, I/O interface, and so on. As a result, it is getting more and more important to provide efficient interconnections among numerous processing cores and peripheral modules of an SoC. Traditional interconnection techniques, such as on-chip bus or point-to-point interconnections, are not suitable for current large-scale SoCs because of their poor scalability. In recent years, a design paradigm based on a Network on Chip (NoC) was proposed as a solution for interconnection techniques of large-scale SoCs [5,6]. The modular structure of NoC makes chip architecture highly scalable, and well-controlled electric parameters of the modular block improve the reliability and operation frequency of the on-chip interconnection network. After the proposal of the NoC design paradigm, research concerning NoC has fairly advanced. In this chapter, the advantages and building blocks of NoC are briefly described, and then considerations for real chip implementation of the NoC are discussed.

5.1 NETWORK ON CHIP

5.1.1 NoC for SoC Design

To date, there are two distinctive types of SoCs; one is the homogeneous SoC and the other is the heterogeneous SoC. In homogeneous SoCs, properly designed modules are replicated and repeatedly placed on a single chip in a regular topology. The MIT RAW processor [7] and Intel's 80-tile processor implementation [1] are examples of the homogeneous SoC. These processors consist of modular tiles, which include processing cores and five-port crossbar switches. In each tile, a five-port crossbar switch provides connections to the four neighbor tiles of the mesh topology and processing core inside the tile. For homogeneous SoCs, crossbar switches are commonly adopted instead of conventional buses because the nonblocking characteristic of the crossbar switch has the advantage of enabling concurrent interconnections between multiple processing cores and peripheral modules. Figure 5.1a depicts how the homogeneous SoC is organized in a regular array structure. On the other hand, the heterogeneous SoC [4,8,9] integrates various functional modules that are usually dedicated for accelerating specialized computation of target applications. In the

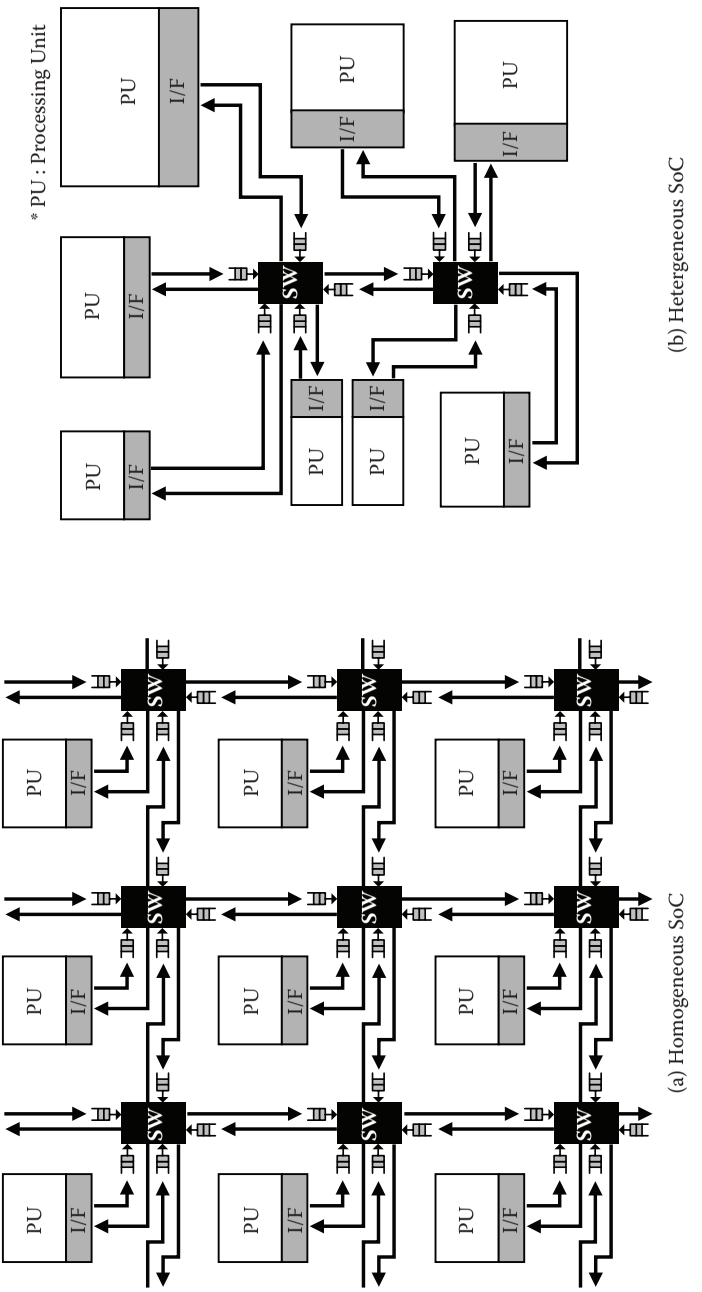


FIGURE 5.1 Two types of SoCs.

case of heterogeneous SoC, it is difficult to build up regular structures such as a 2D array of processors, because sizes of the functional modules are different from one another. In addition, adopting regular interconnections may result in waste of wire resources because of highly localized and fixed traffic patterns among functional modules. Therefore, in most cases, the heterogeneous SoC requires an optimized interconnection that is tailored to the data traffic patterns of the target application. Figure 5.1b shows an example of a simplified heterogeneous SoC interconnection.

In the previous examples, entirely homogeneous or heterogeneous SoCs were introduced first to highlight the distinctions between the two types of SoCs. With the increase in features demanded, however, hybrids of the homogeneous and heterogeneous SoCs will be widespread in the near future. Because homogeneous architecture is mainly beneficial for providing huge computing power, integrating additional functional modules such as external I/O or application-specific accelerators is usually necessary. This results in a hybrid of homogeneous and heterogeneous SoC architecture. In this case, a scalable and structured interconnection is essential to establish efficient interconnections for complex SoC architectures of any topologies. When compared to the conventional bus and point-to-point architecture, NoC has huge advantages in building up complex SoC architectures. In Subsection 5.1.2, the benefits of adopting NoC for SoC design are explained after a brief introduction.

5.1.2 Comparison of Bus-Based and NoC-Based SoC Design

The NoC-based SoC design utilizes two major concepts, different from those of the usual bus-based SoC architecture. First, it is a packet transaction rather than a circuit transaction, and, secondly, it is a distributed network structure, rather than a conventional globally shared bus or a centralized matrix. In the NoC-based SoC design, each of the functional modules should be designed latency-insensitive to support packet transactions. Although this makes functional module design slightly more complex, many benefits are gained from the packet transaction. It improves reliability and operational speed of the interconnection link because the packet transaction is intrinsically pipelined, so that the physical length of the interconnection link is kept short. Efficient link utilization is the other advantage because only a part of the end-to-end path between the functional modules is occupied by the traversing packets. Another advantage is that electrical parameters of the existing NoC is not influenced by the addition of other NoC modules because of the structured characteristic of the NoC. This enables building up of large-scale SoCs from small existing components by addition of required functional modules. For these reasons, advanced bus architectures are also gradually considering the packet transaction concept in their protocol; e.g., multiple outstanding addressing, split transactions, and multithreaded transactions [10,11]. In the near future, it is expected that commercial bus architectures for high-end SoCs would take the NoC design into their specifications [12,13].

As already mentioned, the NoC paradigm has come forward to alleviate a design complexity of a very-large-scale SoC design that was not possible to be fabricated on a single chip so far. As manufacturing scale goes into very-deep-submicron (VDSM) scales and a few tens of functional modules are integrated on a single chip, communication between the integrated modules become more and more complicated

and unmanageable with the conventional design methodology. Even worse, communication itself becomes a design bottleneck in respect of performance, design effort, area cost, and power consumption. The NoC paradigm tries to solve this problem based on a well-defined layered architecture, which is much like Open System Interconnection (OSI), the basic reference model [14]. This methodology divides the communication mechanism into several layers that are independent of one another, so that the design becomes more manageable and easily modifiable. The detailed correspondence of NoC to the OSI reference model is described in Subsection 5.1.3. Before going further, we will summarize the pros and cons of a NoC-based design and a bus-based design, as shown in Table 5.1. For many reasons, the NoC-based design is much more advantageous than the bus-based design for high-performance SoCs. Details are addressed in the table.

5.1.3 OSI Seven-Layer NoC Model

The NoC exploits a layered-stack approach to communicate between integrated processing elements (PE), which may be processors, dedicated functional blocks, or memories. The NoC decouples the communicational part from the computational part efficiently from an early design stage. This dissociation enables a parallel design of PEs and interconnection structures without any interference between them. In this subsection, we focus on the communicational part, the functions of which are provided by NoC. Figure 5.2 represents the OSI seven-layer model [14] and its correspondence to the building blocks of NoC. The NoC involves the four bottom layers of the OSI seven-layer model, realized as hardware on a fabricated chip.

Figure 5.2b describes design issues to be considered for each of the OSI model layers. Because the physical layer defines electrical and physical specifications, its design issues include operational frequency of the wire link, signaling scheme, clock synchronization, and so on. In contrast to the traditional personal computer (PC) networks, NoC physical layer design has significant impact on power consumption and performance of SoC because a large volume of on-chip data transactions among PEs occur frequently and concurrently, compared to inter-PC communications. Therefore, inefficient design of the physical layer easily results in a huge amount of wasted power, and poor performance. In addition, maximum clock frequency and wire width of the physical layer determine the theoretical bandwidth limit of the NoC design. After the first prototype chip fabrication of NoC [16], there have been researches concerning the physical layers of NoC [15–18].

The data link layer provides abstraction of data transactions between NoC building blocks such as crossbar switches and network interfaces (NIs). Although the data link layer of traditional computer networks include error detection and correction of the physical layer, these features are usually omitted in NoC. This is because reliability of the on-chip wire is much higher than conventional media such as twisted-pair cables, and on-chip silicon resources for error correction/detection are limited in many cases. Because simple hardware implementation is very important to reduce overheads of on-chip interconnection, the buffer size of each NoC component is usually as small as a few packets, and a strict flow control scheme is adopted to prevent packet loss.

TABLE 5.1
Comparison between NoC-Based and Bus-Based Design Methodologies

	NoC-Based Design			Bus-Based Design
Bandwidth and speed	Nonblocked switching guarantees multiple concurrent transactions. Pipelined links: higher throughput and clock speed. Regular repetition of similar wire segments, which are easier to model as DSM interconnects.	+	–	A transaction blocks other transactions in a shared bus. Every unit attached adds parasitic capacitance; therefore electrical performance degrades with growth.[a] [21]
Resource utilization	Packet transactions share the link resources in a statistically multiplexing manner.	+	–	A single master occupies a shared bus during its transaction.
Reliability	Link-level and packet-basis error control enables earlier detection and gives less penalty. Shorter switch-to-switch link, more error-reliable signaling. Reroute is possible when a fault path exists (self-repairing).	+	–	End-to-end error control imposes more penalty. Longer bus-wires are prone to error. A fault path in a bus is a system failure.
Arbitration	Distributed arbiters are smaller, thus faster.	+	–	All masters request a single arbiter; thus the arbiter becomes big and slow, which obstructs bus speed.
	Distributed arbiters use only local information, not a global traffic condition.	–	+	A central arbitration may make a better decision.
Transaction energy	Point-to-point connection consumes the minimum transaction energy.	+	–	A broadcast transaction needs more energy.
Modularity and complexity	A switch/link design is reinstantiated, and thus less design time. Decoupling b/w communicational and computational designs	+	–	A bus design is specific, thus not reusable.
Scalability	Aggregated bandwidth is scales with network size.	+	–	A shared bus becomes slower as the design gets bigger and thus is less scalable.
Clocking	Plesiochronous, mesochronous, and GALS fashion do not need a globally synchronized clock; much advantageous for high-speed clocking.	+	–	A global clock needs to be synchronized over the whole chip bus area.

Continued

TABLE 5.1 (Continued)
Comparison between NoC-Based and Bus-Based Design Methodologies

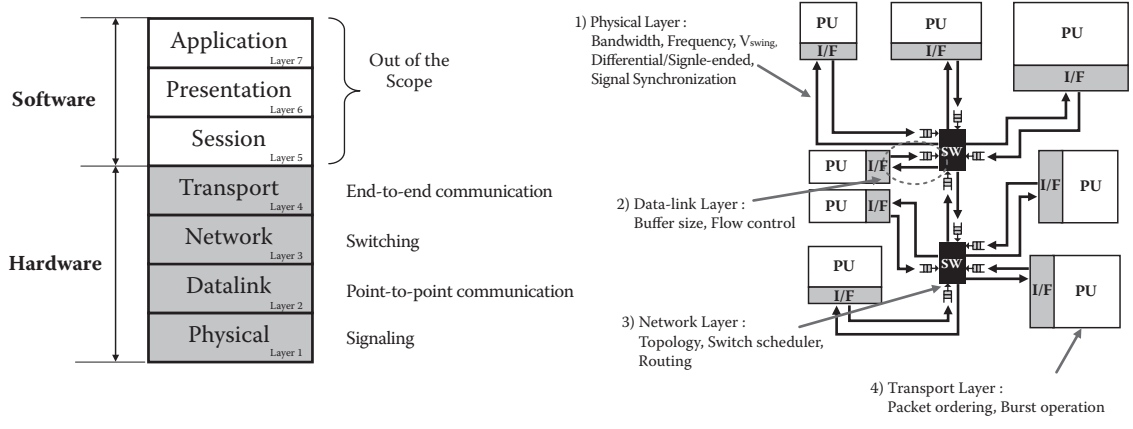
	NoC-Based Design			Bus-Based Design
Latency	Internal network contention causes a packet latency. Repeated arbitration on each switch may cause cumulative latency. Packetizing, synchronizing, and interfacing cause additional latency.	–	+	Bus latency means a wire speed once a master has a grant from an arbiter.
Overheads	Additional routers/switches and buffers consume area and power.	–	+	Less area is consumed.[b] Less buffers are used.[b]
Standardization	There is no NoC-oriented global standard protocol yet; however we can use legacy interfaces such as OCP, AXI, etc.	–	+	AMBA and OCP protocols are widely used and designed for many functional IPs.

[a] Recent advanced buses are using pipelined wires by inserting registers in between long wires [16,17].

[b] Recent advanced buses uses crossbar switches and buffers (register slices) in their bus structure.

The third layer of the OSI seven-layer model is the network layer. The major design issues of the network layer involve NoC topology selection, packet routing schemes, and quality-of-service (QoS) guarantee. Topology of the NoC should be very carefully selected because of its significant impact on overall performance and power consumption. To reduce communication overheads, PEs with frequent data transactions need to be placed at a close distance, although sufficient bandwidth for every node should be supplied to avoid performance degradation. The details of NoC topology selection are explained in Chapter 6. As for low-overhead routing schemes, source routing is generally adopted in NoC because only simple decoding logic is needed for packet routing at each router instead of large look-up tables. Finally, guaranteeing QoS is also important for efficient utilization of bandwidth. In the NoC, QoS guarantee is implemented by supporting priorities or constructing different classes of virtual channels. The Æthereal NoC [19] implements both Guaranteed Throughput (GT) and Best Effort (BE) router for worst case QoS guarantees. An arbitration look-ahead scheme, which aims at reducing packet switching latency, is also reported [20].

The transport layer is the highest level of the OSI seven-layer model implemented by the NoC. In the NoC-based SoC design, each of the PEs and functional blocks should be designed according to NoC protocols to support interfaces to the on-chip network. NI modules, which perform packet generation and parsing, provide abstractions of end-to-end data transactions between PEs and other functional modules. In many NoC implementations, out-of-order packet transmission is not supported because of limited buffer resources on a chip.



(a) OSI 7-Layer Model

(b) Layered architecture of NoC

FIGURE 5.2 OSI seven-layer model and its correspondence to the building blocks of NoC.

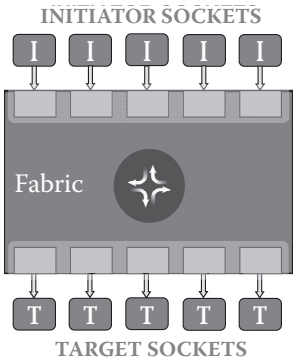


FIGURE 5.3 Block diagram of Sonics MicroNetwork.

In summary, NoC provides similar network functions as a traditional PC network. However, power and area efficiency in NoC implementation is emphasized because of a huge amount of on-chip data transactions and limited silicon resources.

5.1.4 An Example of NoC-Based SoC Design

This subsection introduces an example of large-scale SoC design flow, which is based on NoC. SoC design in the forthcoming billion-transistor era will imply numerous predesigned semiconductor IP blocks, and multicore SoCs will dominate. The current design methodologies, which are based on bus-based communication architectures, are not expected to provide the necessary design productivity to cope with such multibillion transistor SoC designs. To meet the productivity and signal integrity challenges of SoC designs using nanotechnologies, an independent design of sophisticated and reliable interconnection platform, easily tailored to the target application, would be beneficial. Using the NoC approach, a SoC design can be viewed as a composition of complex functional modules that are interconnected by a structured on-chip packet transaction network. A functional module can be a microprocessor core, a memory core, a digital communications building block, or custom logic. Instead of interconnecting such functional modules at the top level using an ad hoc routing of dedicated global wires, as is done today, a better approach is to interconnect them by a structured on-chip packet-switching network that routes packets between them.

As an example of an NoC-based SoC platform, MicroNetwork was developed from Sonics [21] as, shown in Figure 5.3. A configurable agent within MicroNetwork decouples the functionality of each functional module from the communication between cores and thereby enables each core to be rapidly reused, without rework, in subsequent or parallel SoC designs. A typical SoC integration flow using MicroNetwork is as follows:

Step 1. Precharacterize the agents of the MicroNetwork across a broad range of configurations.
Step 2. Determine the base MicroNetwork architecture, which refines the SoC architecture to a block diagram.
Step 3. Choose basic MicroNetwork design parameters such as the peak bandwidth, datapath width, pipeline depths, and clock frequency.
Step 4. Build a dataflow model using behavior models or IP core models. Simulate and analyze the constructed simulation model.
Step 5. Improve the model by integrating more accurate IP core models, allocating the required bandwidth.

Step 6. Synthesize, place, and route an early MicroNetwork net list to test physical predictability.
Step 7. Integrate IP cores and verify functionality.
Step 8. Map control flow by establishing interrupt and error architectures.
Step 9. Verify system functionality and complete the physical design.

The advantages of using such a network-based SoC design approach include both modularity and performance benefits. By localizing dedicated wiring to individual modules and imposing a modular structure, the design process can be greatly simplified. Each module can be designed and verified individually with respect to performance. The NoC approach also facilitates modular design by defining a standard interface. In many cases, previously designed modules may be reused in new designs. In addition to reusability, a standard interface also facilitates interoperability between modules and simplifies the design process by abstracting away the complexity of intermodule communications.

5.2 ARCHITECTURE OF NOC

5.2.1 Basic NoC Design Issues

Figure 5.4 illustrates an overall architecture of an NoC, and basic design issues are pointed out. First, appropriate topology and protocol should be selected when NoC design begins. In the case of NoC topology, it can be configured with regular topologies such as Mesh, Torus, Tree, or Star, or an optimization can be carried out to build an application-specific topology without regular pattern. Secondly, protocols including packet format, end-to-end services, and flow control should be defined and implemented in the NI module. In the case of packet and flit size definition, it affects the buffer capacity requirement and multiplexing gain in the network. These topics will be discussed in more detail in Chapter 6. The packet switching scheme is the next factor to be determined. There are many switching

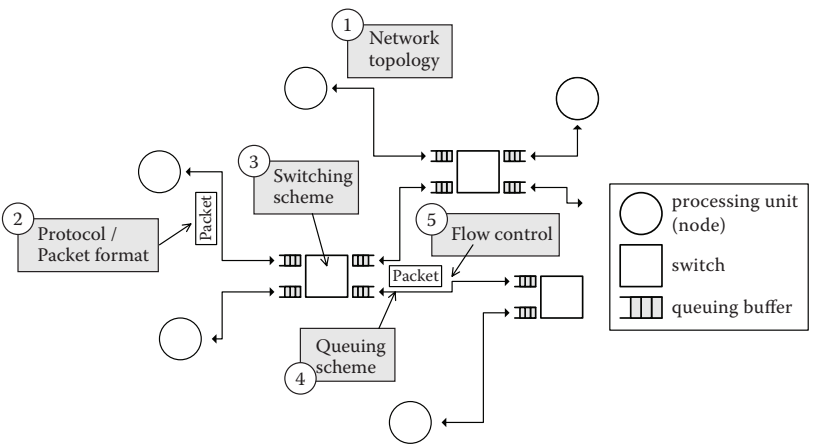


FIGURE 5.4 Basic design parameters of NoC.

methods, and among them three are famous: store and forward, wormhole switching, and cut-through switching.

Store and forward: The entire data of a packet at the incoming link are stored in the buffer for switching and forwarding. A buffer with a large capacity is required.

Wormhole routing: An incoming packet is forwarded right after the packet header is identified and the complete packet follows the header without any discontinuity. The path that the packet uses traveling through the switch is blocked against access by other packets.

Virtual cut through: The path is determined as in wormhole routing. In case the next hop is occupied by another packet, the packet tail is stored in a local buffer waiting for clearing up the path. Its buffer size can be smaller than the store-and-forward switch, but if the packet size is large many local buffers would be occupied to delay the switch throughput.

It is necessary that an appropriate switching scheme is chosen considering the target application and the silicon resource budget. In many NoC implementations, the wormhole routing scheme is chosen because of its low buffer resource requirement.

Once the basic topology, protocol, and routing method are determined, operation of the crossbar switch is as follows. When the input packets arrive at the input port, the crossbar switch scheduler gets the destination information from the input packets. If every packet arrives at a different input port and wants to leave from another output port, there are no input and output conflicts. Then, the scheduler connects the cross junctions so as to connect the packets at the input ports to their output ports. If conflicts occur, the scheduler should resolve them by the predefined algorithm. Buffers are important to store the packet data temporarily for congestion control.

There are three queuing schemes distinguished by the location of the buffers inside the router (switch): input queuing, output queuing, and virtual output queuing (see Figure 5.5).

Input queuing: Every incoming link has a single input queue so that N queues are necessary for N × N switches. Input queuing suffers from the head-of-line blocking problem; i.e., the switch utilization gets saturated at 58.6% load.

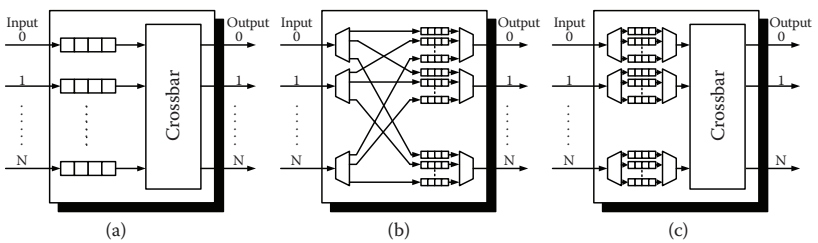


FIGURE 5.5 Queuing schemes: (a) input queuing, (b) output queuing, and (c) virtual output queuing.

Output queuing: The queues are placed at the output port of the link, but N output queues for every outgoing link are required to resolve the output conflict, resulting in N^2 queues. Because of the excessive number of buffers and their complex wiring, in spite of its optimal performance, output queuing is not used.

Virtual output queuing: The advantages of the input queuing and output queuing are combined. A separate input queue is placed at each input port for each output requiring N^2 buffers. The head-of-line blocking problem occurring in input queuing is resolved by scheduling. Complicated scheduling algorithms such as iterative algorithms are needed.

The last issue in Figure 5.4 is flow control or congestion control. There are several solutions that prevent packets from causing output conflict and buffer overflow.

Packet discarding: Once the buffer overflows, the packets coming again are simply dropped off.

Credit-based flow control: A back pressure scheme uses separate wires for the receiver to inform the transmitter of buffer congestion so as to prevent packet loss, as shown in Figure 5.6a. The propagation delay time between transmitter and receiver should be considered carefully to avoid packet transmission while the wait signal is coming on the wire. In the Window scheme of Figure 5.6b, the receiver regularly informs the transmitter about available buffer space.

Rate-based flow control: The sender gradually adjusts packet transmission according to the control flow messages from the receiver. For example, the error control uses the Go-back-N algorithm and related signals. In this case, a certain amount of buffer space is guaranteed for effective flow control, but because of long control loop, the rate-based flow control potentially suffers from instability.

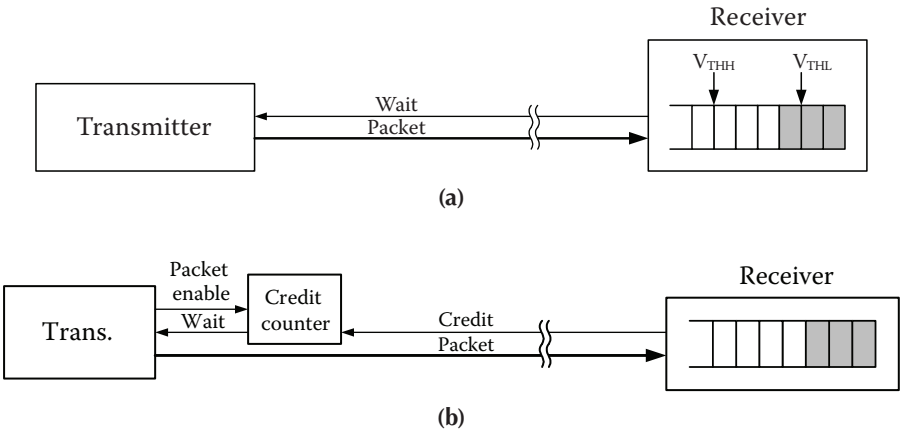


FIGURE 5.6 Flow control schemes.

In addition to basic design parameters required to be determined for an NoC design, additional functionalities are required to enhance the NoC performance and optimize cost overheads. QoS is one of the most critical issues. The quality factors in an NoC should be bandwidth and latency. Guaranteed bandwidth and limited latency enable packet transactions to be punctual and thus make it possible to execute real-time applications in NoC-based SoC.

5.2.2 Design of NoC Building Blocks

In this section, the circuits of the basic building blocks of the NoC will be introduced. The basic building blocks are high-speed signaling circuits, queuing buffers and memories, switches, and crossbar switch schedulers.

5.2.2.1 High-Speed Signaling

For high-speed signaling, bit width, operating frequency, and differential signaling are carefully explored. If we use a channel with increased bit width, the interference among channel wires would hinder high-speed operation of the channel. Differential signaling can achieve high-speed operation with relatively high SNR, but it consumes twice the area needed for single-wire signaling, and, for wide bit width, it is impractical. Figure 5.7a shows voltage mode signaling, which usually uses repeaters along the wire to reduce the capacitance load of driver circuits. Figure 5.7b shows current mode signaling circuits; it is known that current mode signaling is faster than voltage mode signaling.

5.2.2.2 Queue and Buffer Design

The effective bandwidth of the data-link layer is heavily influenced by the traffic pattern and queue size. The queue buffer is located at the input or output port of the switch and in the network interface that interconnects the logic blocks to the sender or receiver. The queuing buffer consumes most of the area and power among all the composing building blocks in the NoC. There are two ways to implement the buffers, flip-flop based (register) and SRAM based. Figure 5.8 shows four different register designs: (a) a conventional shift register, (b) push-in shift-out register, (c) push-in bus-out register, and (d) push-in mux-out register. A SRAM-based design is also shown in Figure 5.9.

In a conventional shift-register design as shown in Figure 5.8a, bubble cells may occur when the packet input/output rates are different. Shifting all the registers at every *packet-out* consumes a huge amount of power. Furthermore, the minimum latency in a queue is as long as the physical queue rather than the backlog. Although

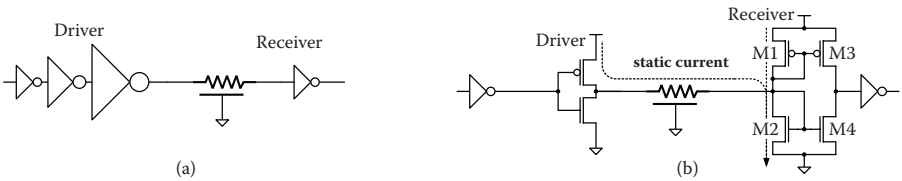


FIGURE 5.7 (a) Voltage mode and (b) current mode interconnection driver.

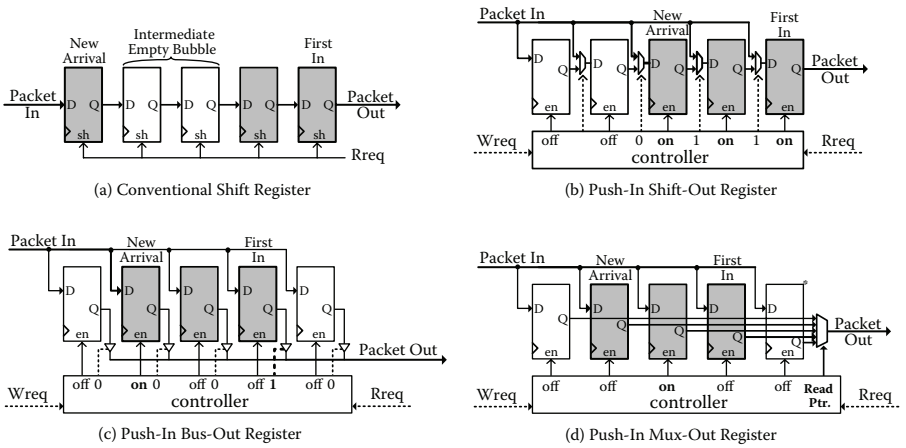


FIGURE 5.8 DFF-based queues.

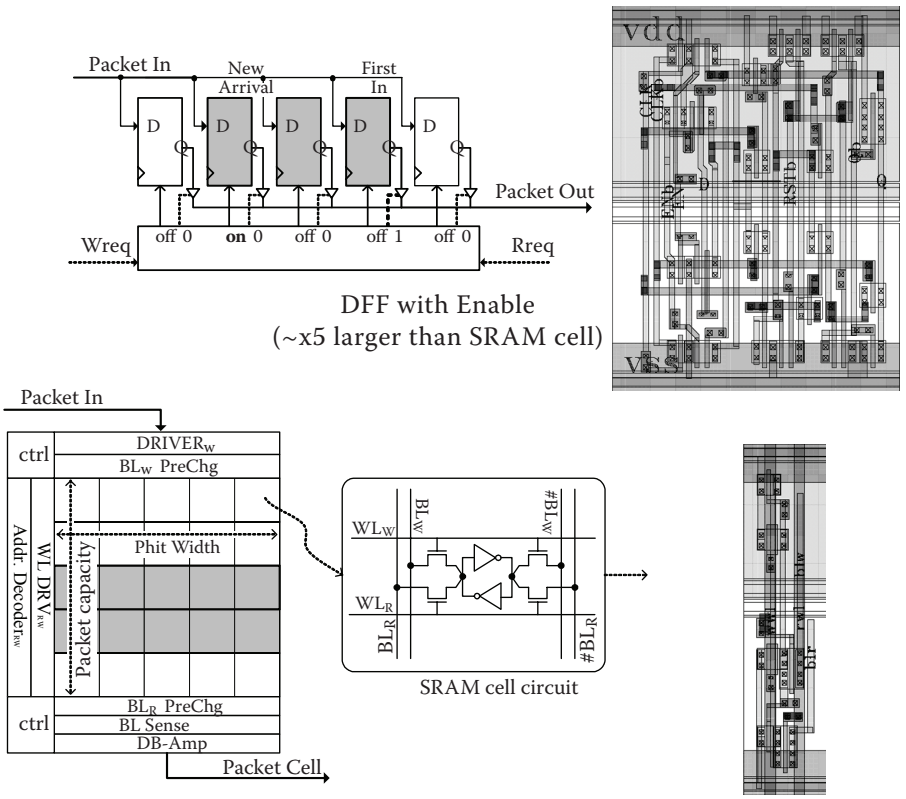


FIGURE 5.9 SRAM-based queues.

this design is the simplest, it is not desirable to be implemented on a chip due to its long latency and power waste.

To remove the intermediate empty bubble, an arrival packet can be stored at a front empty cell rather than the tail of a queue. This input style is called *push-in*, as illustrated in Figure 5.8b. It can remove unnecessary latency and power consumption caused by the empty bubble. Only the occupied register cells are enabled. However, the shifting register style still consumes unnecessary power by shifting all of the occupied cells when a packet is output. To avoid this shifting operation, outputs of all registers are tied to a shared output bus line via tristate buffers, as shown in Figure 5.8c. The register holding the *first-in* packet is connected to the output bus and turns on the tristate buffers. In this design, only a cell in which a newly arrived packet is stored is enabled. As queuing capacity increases, the capacitance of the shared bus wire rises as well, because of the parasitic capacitance of tristate buffers, and the delay and power consumption become considerable. To eliminate this effect, output multiplexers can be used, as shown in Figure 5.8d.

Although the area and power consumption of the register-based buffer are relatively large, its capacity, or buffer size, is limited. As the queuing capacity rises to a dozen of packets, register-based implementation is not good in respect of both area and power. The dual-port SRAM cell is used for large-capacity queuing, as shown in Figure 5.9. It shows the circuit and layout of a unit cell, and, for comparison, a layout of the D-FF, too. A SRAM cell occupies only a fifth of a register (flip-flop) area.

5.2.2.3 Switch Design

The conventional switch consists of input queue (IQ), scheduler, switch fabric, and output queue (OQ), as shown in Figure 5.10a. There are two kinds of switch fabric design: a cross-point and Mux-based switch fabric as presented in Figures 5.10a and 5.10b, respectively. The cross-point switch has pass transistors at each crossing junction of input and output wires. In this switch fabric, the capacitive loading driven by the input driver is the junction capacitance of the pass transistors on input and output wires, as well as the wire capacitance itself. These parasitic capacitances and the series resistance of the wire bring up RC delay, limiting the bandwidth of the switch. The voltage swing on the output wire is reduced to $V_{DD}-V_{th\_N}$ because of the

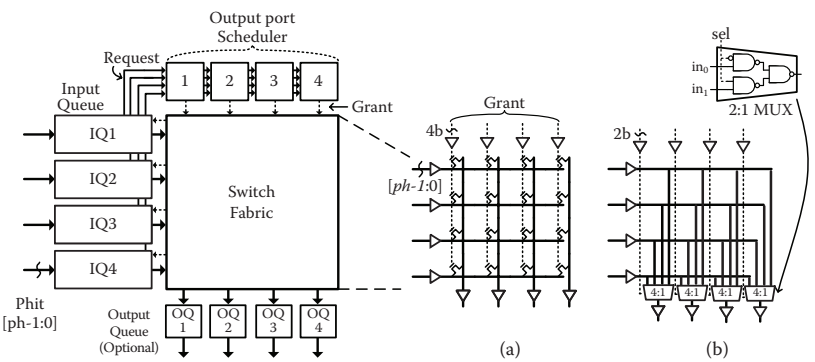
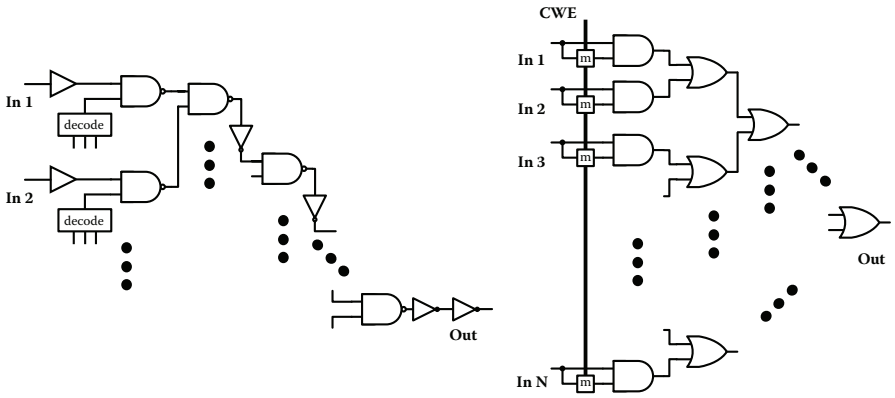


FIGURE 5.10 (a) Cross-point and (b) Mux-based switch fabrics.



(a) A Circuit based on a tree of NAND-inverter **(b) Pre-programmed crossbar**

FIGURE 5.11 Circuits for crossbar switch.

threshold voltage drop of the NMOS pass-transistor, thus reducing power dissipation. The CMOS pass gate can be used to avoid voltage drop, but another control wire is required to increase the area consumption. The fabric area is determined by the wiring area and not by the transistors, so its area cost can be minimized. The Mux-based switch uses a multiplexer for each output port. The capacitive loading driven by the input driver is the input gate capacitance of the multiplexers, as well as the input wire capacitance. The parasitic capacitance is larger than the cross-point switch, but the high resistance of the pass transistor does not exist. It can operate at relatively high speed and be synthesized by EDA tools.

Figure 5.11 shows other circuit implementations of the 256 × 256 crossbar switch [26]; Figure 5.11a shows a circuit based on a tree of NAND inverter (AND) and local decoders, and (b), a pre-programmed crossbar.

To arbitrate output conflicts, a scheduler is used on each output port. Arbitration scheduling is required to perform fair routing, no-starvation, and maximum throughput at high levels, although it does not significantly increase the latency, power, and area of the switch design. The latency of the arbiter becomes larger than that of the switch fabric as the switch size gets bigger than 16 × 16. As shown in Figure 5.12, the scheduler occupies a similar area as the switch fabric when the phit (*ph*ysical dig*it*) width of a port is 10 bits. Therefore, the scheduler design is as important as the switch fabric.

5.2.2.4 Scheduler Design

Many scheduling algorithms are known, such as round-robin type, propagation basis, and maximum-weight-matching algorithms. However, diagonal/rectilinear propagation methods, or weighting matching, such as LQF and OQF, are too complicated to be laid on the chip with a small area. The round-robin scheduling algorithm is most widely used in the on-chip network because of its fairness and no-starvation properties. In round-robin scheduling, each port in turn is given the opportunity to transmit a fixed amount of data over a limited time. The port may not take the

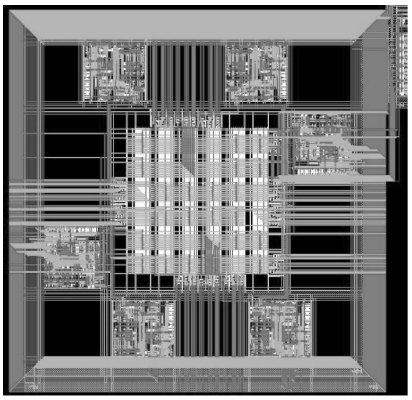


FIGURE 5.12 Switch layout diagram of 6 × 6 and phit width of 10 bits/link.

opportunity, and after the fixed period of time, the opportunity is moved to the next port. Figure 5.13a shows an example of round-robin scheduling for a six-port system, in which ports 1, 3, 4, and 6 are ready to transmit the packet. The round-robin scheduler can be implemented by using two priority encoders, as shown in Figure 5.13b, or mux-tree-connected logic.

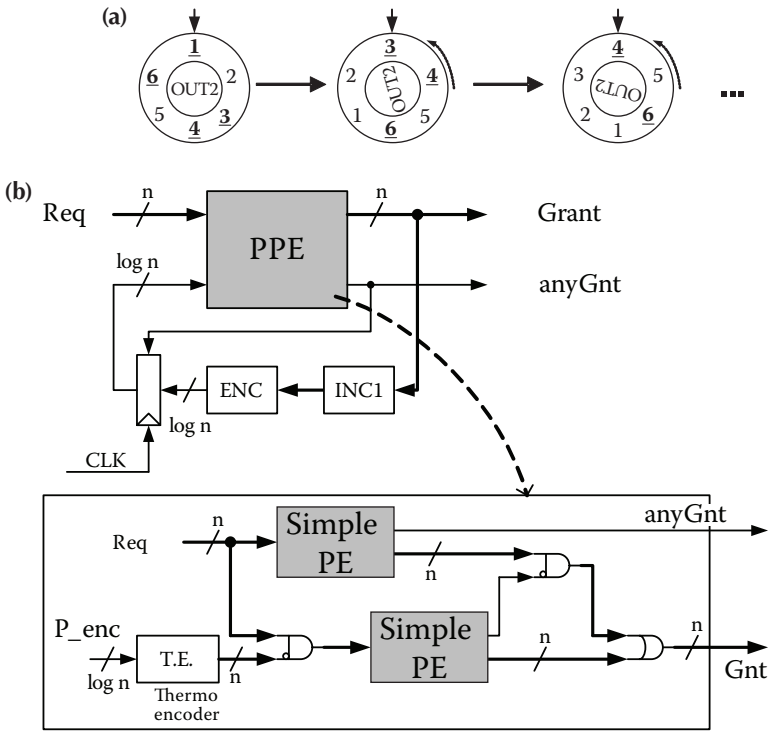


FIGURE 5.13 (a) Round-robin algorithm, (b) round-robin algorithm circuits with 2 priority encoders.

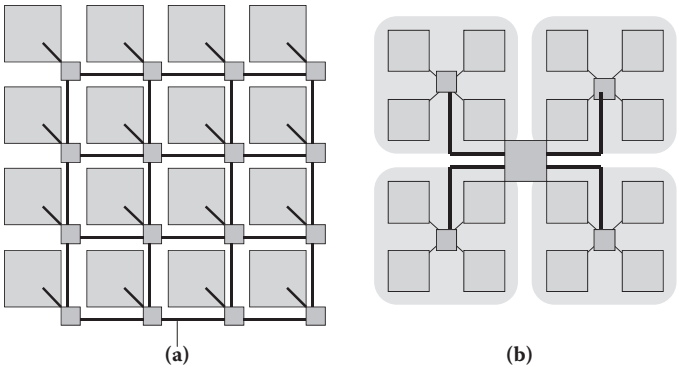


FIGURE 5.14 (a) Mesh topology and (b) hierarchical star topology.

5.3 PRACTICAL DESIGN OF NOC

The approaches to NoC design can be divided into two main categories: (1) determining or optimizing fundamental network parameters and (2) proposing new additional features for enhancing performance or reducing cost. These NoC designs are based on the top-down approach, which tries to shrink the legacy network to fit it into the silicon area. Thus, such approaches have a tendency to follow the communication and network features adopted in legacy networks, but some features seem to be inadequate or unnecessarily complicated for on-chip situations. Although the NoC borrows basic concepts from legacy network architectures, for implementing the network architecture on a chip, many physical design issues regarding chip implementation need to be considered. However, most of the previous work on NoC design did not encounter actual implementation issues because chip implementation was not achieved, with the on-chip characteristics being ignored. Their architecture specifications such as topology, routing, switching, and link bit width are determined without consideration of implementation constraints such as low-power consumption and small chip area. More physical level issues such as on-chip serialization and mesochronous communication also must be considered to make an NoC on a silicon. In this section, some architectural decision and physical implementation issues for the practical design of NoC will be described from a chip designer's point of view.

5.3.1 Topology Selection

Traditionally, mesh topology, as shown in Figure 5.14a, has been widely used and studied for parallel computing architectures due to its highly scalable and regular architecture. Most of the previous NoC implementations [5,22] utilized the mesh topology or its derivative and a wide link, over 128 bit. However, mesh topology must be reinvestigated for the practical NoC design in terms of area and power consumption in an SoC environment. Although a wide link has been widely used in an on-chip situation, a large number of metal wires complicates not only metal routing but also the placement of processing elements (PEs). In an actual layout, PEs may be placed irregularly to minimize the chip area; therefore, the regular structure of the mesh topology may not be seen in an SoC design. Furthermore, wide link means wide switch fabrics,

which increases the network area significantly. Also, mesh topology results in inefficient global packet transactions because of its large hop counts. On the contrary, star topology has not been popular because of its poor scalability, in spite of its highest bisection bandwidth and small hop count. However, in the case of an on-chip situation, the number of integrated PEs (N) is limited to a few tens. Even if N is larger than 100, they can be interconnected hierarchically (i.e., hierarchical star topology) as shown in Figure 5.14b; PEs communicating each other intensively will be grouped as a cluster, and local network will interconnect them; the clusters will be interconnected by a global network. Then, the interconnection in a cluster (as well as between clusters) becomes the same issue as the interconnection of a few tens of PEs. Therefore, the star or hierarchical star topology can be a potential candidate, rather than the mesh topology, for practical NoC design because hierarchical star topology shows the highest cost efficiency, as well as lowest latency [23]. A detailed comparison between the mesh and hierarchical star topologies will be presented in Chapter 6.

5.3.2 Routing Scheme

The adaptive routing method is widely used for general macronetworks because it enables fault-tolerant packet transfer and hotspot avoidance by using alternative packet routing paths, and leads to higher throughput. However, the packets may arrive out of order in adaptive routing; thus, huge scratch buffering resources are needed to store incoming packets temporarily, and packet reordering based on a packet sequence number is required (see Figure 5.15). Because of such a huge area cost and unpredictable latency, and considering hardware overhead versus performance improvement, adaptive routing is not suitable for NoC implementation. On the other hand, a deterministic routing method such as source routing may be a good solution for a practical NoC design. In deterministic routing, a packet is transferred to a destination through a fixed routing path that is defined at a network interface. A switch does not support the adaptive routing function, and NIs need no packet reordering function. As a result, the NoC hardware implementation cost is very low compared to adaptive routing. There are still problems regarding the hotspot problem in deterministic routing. However, if the bandwidth is limited by hotspot, we can say that it is the maximum bandwidth the network can afford. Also, in an SoC environment, the designer has a good knowledge of the traffic characteristics for specific applications and can avoid the hotspot problem by allocating the routing paths wisely.

5.3.3 Switching Scheme

Packet switching methods such as wormhole-based routing is widely used for NoC because of limited buffering resources. A packet-switched network uses channel

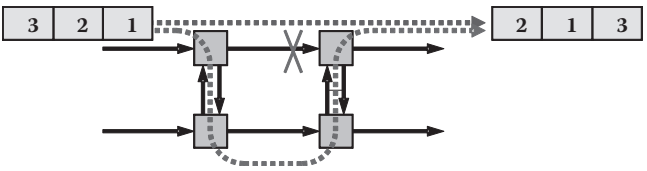


FIGURE 5.15 Adaptive routing method.

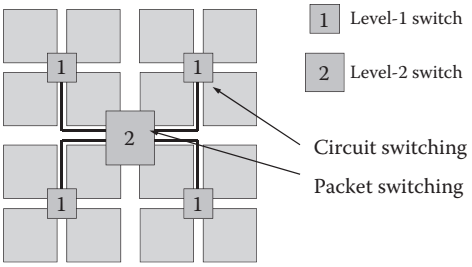


FIGURE 5.16 Adaptive switching method.

resources more efficiently than a circuit-switched network because the route from a source to a destination is pipelined. The advantage increases as the route becomes longer. However, route length in current SoCs is insignificant, so this alone does little in a packet-switched network. Clearly, if a NoC uses a high clock frequency and small packets, the packet-switched network works well as a global interconnect architecture. However, if the NoC has a lower clock frequency for low power consumption or uses burst packet transfer, repetitive processes such as synchronization, packet queuing, and arbitration in all the intermediate switches become redundant and inefficient, leading to large latency. To solve this problem of the packet-switching NoC, we recommend the use of adaptive switching methods in which circuit switching techniques are combined with packet switching, as shown in Figure 5.16. The NoC has level-1 and level-2 switches. A level-1 switch supports both switching modes; a level-2 switch supports only packet switching. The first packet of a burst packet flow enables the level-1 switches' circuit-switching mode, so that the remaining packets bypass the level-1 switches. This mechanism effectively reduces the number of switches along the end-to-end route, and the level-2 switches still provide the advantages of a packet-switched network. When a level-1 switch is in the circuit-switching mode, a synchronizer, a FIFO buffer, and the input port's packet-parsing logic are bypassed, and the packets are routed to the prescheduled output port. This effectively reduces delay and energy consumption for a packet transfer.

5.3.4 Phit Size Determination

Phit is the physical transfer unit in which a packet is divided and transmitted through the core network. The phit size is the bit width of a link and determines the switch area. Therefore, for a practical NoC design, the phit size should be carefully determined, considering both the NoC cost and performance. On-chip serialization techniques can be effectively used to reduce the area and power of the NoC. The phit size determination is largely related to on-chip serialization. If the phit size is smaller than the packet length, serialization must be performed by a factor calculated as follows:

$$\textit{Serialization ratio (SERR)} = \textit{packet size/phit size}$$

Using a large phit size, as shown in Figure 5.17a, obtains high bandwidth easily but is inefficient in respect of NoC cost, i.e., area and power consumption. On the contrary, use of a small-size phit, as shown in Figure 5.17b, reduces the number of link wires;

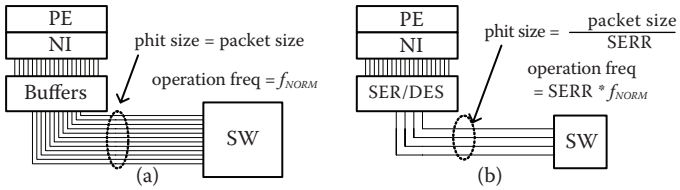


FIGURE 5.17 NoC structure according to phit size.

thus the space between the link wires can be widened to decrease the coupling capacitance. Besides, a small phit reduces the switch fabric size; therefore, area and power consumption of a switch can be reduced [24]. Specifically, switch power consumption is the sum of the arbiter and switch fabric power, and the switch fabric power is dominant. When serialization is used, the operation frequency of the core network must be increased by a factor of the SERR to maintain the network bandwidth. Considering additional circuitry for frequency increments, energy consumption, and area of NoC building blocks such as serialization and deserialization (SERDES), link, synchronizer, and switch are analyzed according to the SERR. Operational frequency without serialization is set to be 200 MHz, and the detailed designs of blocks are based on implementation results. Figure 5.18 shows the analysis results of energy consumption per packet transmission and the area of building blocks in star topology NoC. In Figure 5.18a, energy consumption in a switch and the links decrease as the SERR increases, as mentioned previously. On the contrary, those of a SER, DES, and SYNC increase because of additional circuitry for the serialization. In most cases, a SERR of 4 minimizes the overall power consumption. As shown in Figure 5.18b, serialization also reduces the overall NoC area effectively, and a SERR of 4 is optimal. A SERR of 8 remains the same as one of 4 in terms of area, but is inefficient in terms of power. In the implemented chip [16], the phit size is determined as a quarter packet length to cover design issues on a 4:1 serialized network.

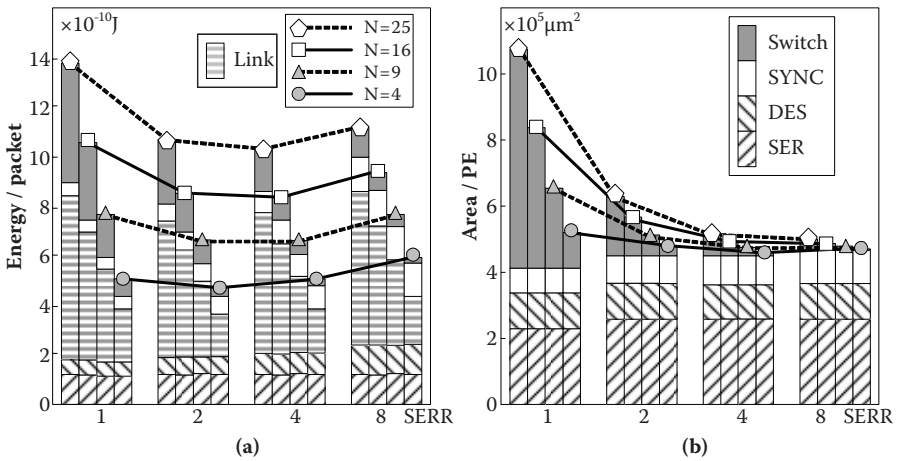


FIGURE 5.18 (a) Energy and (b) area of NoC according to serialization ratio (SERR).

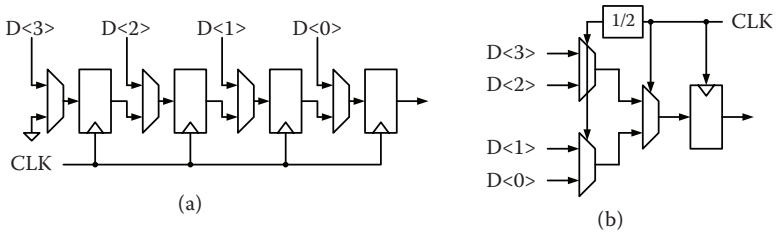


FIGURE 5.19 (a) Shift-register type and (b) Mux-tree type serializers.

5.3.5 SERDES Design

As stated previously, the on-chip serialization technique reduces the area of the NoC significantly; therefore, a serializer and deserializer (SERDES) circuit is an essential building block for a practical NoC design. There are two typical SERDES circuits: shift register and MUX tree, as shown in Figure 5.19. The shift register (SR) serializer fetches parallel packets through 2:1 MUXs when the load signal is enabled. Then, the shift mechanism of the series F/Fs realizes high-speed serialization. The MUX-type serializer divides a packet into several parts, and then multiplexes them into the serialized link. In both serializer types, maximum clock frequency is limited by a delay time of D–FF, and a high-speed clock is required for the serialization speed. To overcome these limitations, a new serializer structure has been introduced [25]. It uses physical delay constant of delay elements (DEs) as a timing reference instead of the clock and signal propagation phenomenon instead of the shifting mechanism. The serialization scheme is called wave-front-train (WAFT).

Figure 5.20 shows a schematic of a 4:1 WAFT serializer and deserializer. When EN is low, D<3:0> is waiting at QS<3:0>. The VDD input of MUXP, which is called a pilot signal, is also loaded to QP. The GND input of MUXO discharges the serial output (SOUT) while the serializer is disabled. If EN is asserted, QS<3:0> and the pilot signal starts to propagate through the serial link wire. Each signal forms a wave front of the SOUT signal, and the timing distance between the wave fronts is the DE and MUX delay, which we call a unit delay. The series of wave fronts propagates to the deserializer like a train.

When the SOUT signal arrives at the deserializer, it propagates through it until the pilot signal arrives at its end, or the STOP node. As long as the unit delay time of the sender and the receiver are the same, D<3:0> arrives at its exact position when the pilot signal arrives at the STOP node. When the STOP signal is asserted, the MUXs feed back its output to its input, so that the output value is latched.

5.3.6 Mesochronous Synchronizer

One of the contributions of NoC to SoC design is the easing of the burden of global synchronization by using mesochronous communication, meaning that network blocks share the same clock source, but the clock phases of functional blocks may be different from one another because of the asymmetric clock tree design and difference in load capacitance of the leaf cell. Without a mechanism to compensate for the phase difference, nondeterministic operation, such as metastability, would impair stability

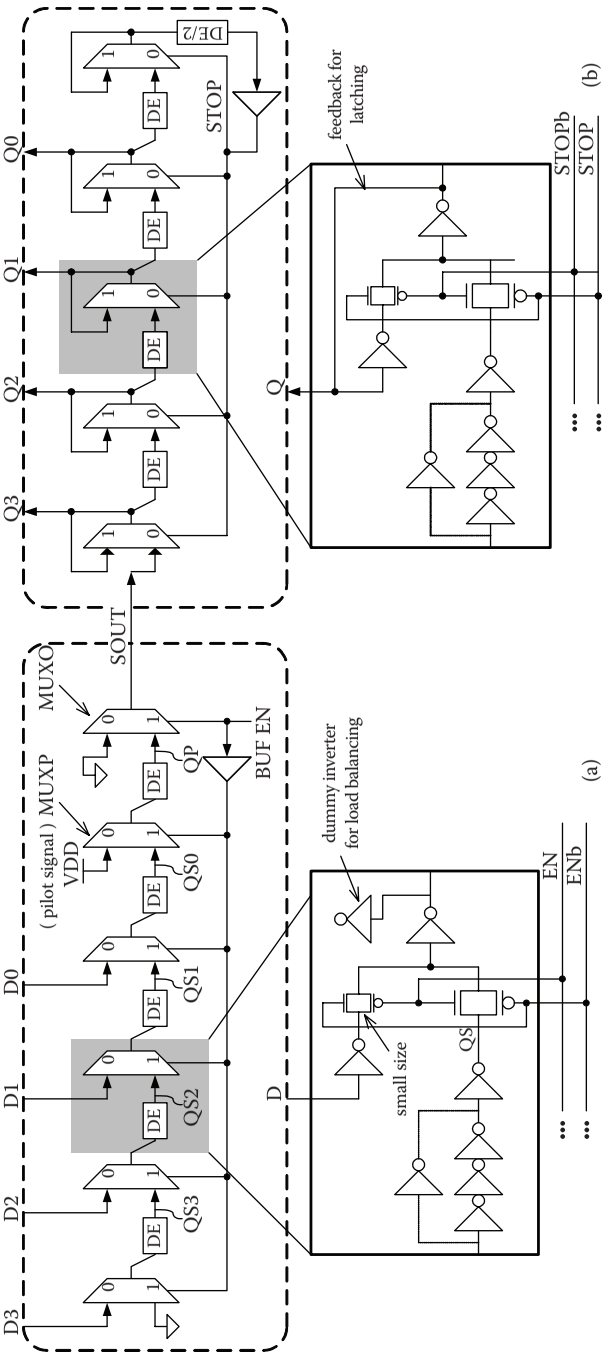


FIGURE 5.20 WAFT (a) serializer and (b) deserializer.

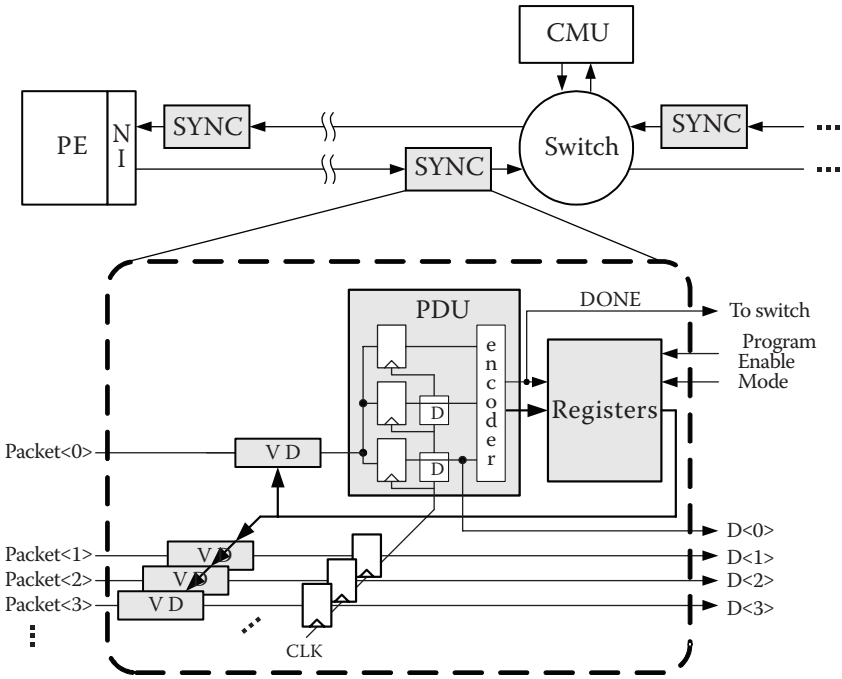


FIGURE 5.21 Programmable delay synchronizer.

or functionality of NoC-based systems. To resolve such a problem, synchronizers are required between the clock domains. Several kinds of synchronizers, such as the FIFO, delay-line, and simple pipeline synchronizers, have been used for the NoC design [15,16]. The FIFO synchronizer is useful when the phase difference between clock domains is unknown. However, its power and latency overheads are considerable. A single-stage pipeline synchronizer provides the best performance and lowest overheads if intensive SPICE simulations under various conditions eliminate all possible synchronization failures. However, as the operating frequency and the number of network nodes increase, a full-custom solution is not practical. To resolve these problems, a programmable delay synchronizer is devised, as shown in Figure 5.21. A Variable Delay (VD) is connected to a simple pipeline synchronizer, and the VD is controlled according to the network circumstance. The appropriate VD setting for a certain circumstance is obtained through a calibration process. This process is performed as follows: In network initialization period, the calibration master unit (CMU) sends packets to all the PEs in the network. When a synchronizer (SYNC) receives a packet, the phase detecting unit (PDU) in the SYNC finds the best timing to sample the input signal. Then, the timing information is encoded into the appropriate VD setting value, and it is programmed into a register and the VDs. After the VD setting, the PDU will sample input signals correctly and asserts the done signal. Then, the switch fetches the output of the SYNC and performs packet switching. While the packets from the CMU are delivered to every PE, all the SYNCs in the CMU-to-PE paths are calibrated. The NI is designed to return the packet when it receives one

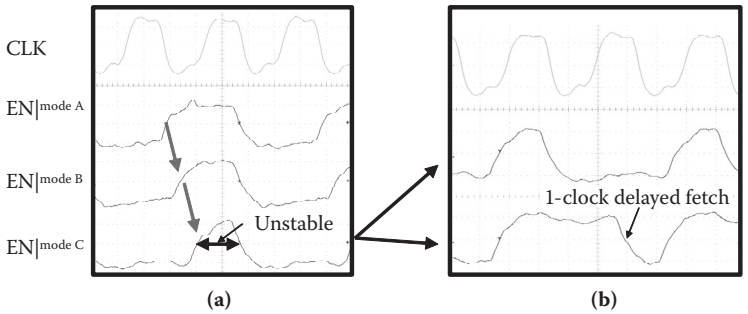


FIGURE 5.22 (a) EN signal variation according to the modes, and (b) unstable EN signal fetching.

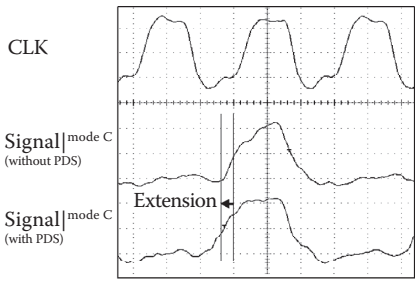


FIGURE 5.23 EN signal expansion using programmable delay synchronizer.

in the initialization period. Therefore, the SYNCs in the PE-to-CMU paths are also calibrated. This calibration process is repeated for all combinations of network circumstances such as operating frequency and topology configuration.

The programmable delay synchronizer operation is measured. In Figure 5.22, the EN signal is fetched at the positive edge of the clock. After that, the EN is deasserted by the handshaking protocol. The problem is that the EN signal is not synchronized with the clock, so that the rising edge of the EN could be very close to the "rising edge minus setup time margin." In Figure 5.22a, the mode C represents a situation in which the rising edge of the EN signal is very close to the clock timing. When the mode is enabled, fetching the EN signal becomes a unstable operation, so EN would be fetched at the next clock cycle, as shown in Figure 5.22b. Figure 5.23 shows the measured waveform when the programmable delay synchronizer is enabled. When the mode C is enabled, delay time applied to the EN signal is reduced, so the fetching timing is effectively delayed compared to a normal operation, which ensures a sufficient timing margin to fetch the EN signal. As a result, the deassertion of the EN signal is delayed as shown in Figure 5.23.

REFERENCES

1. Vangal and Sriram et al., On An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS, Digest of Technical Papers, *IEEE Intl. Solid State Circuits Conference,* pp. 98–589, 2007.
2. Abbo. A. et al., XETAL-II: A 107 GOPS, 600 mw Massively-Parallel Processor for Video Scene Analysis, Digest of Technical Papers, *IEEE Intl. Solid State Circuits Conf.,* pp. 270–602, 2007.
3. Khalany and Brucek et al., A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing, Digest of Technical Papers, *IEEE Intl. Solid State Circuits Conf.,* pp. 272–602, 2007.

4. Lattard, Didier et al., A Telecom Baseband Circuit based on an Asynchronous Network-on-Chip, Digest of Technical Papers, *IEEE Intl. Solid State Circuits Conf.,* pp. 258–601, 2007.
5. Dally, W. J. and Towles, B., Route Packets, Not Wires: On-Chip Interconnection Networks, *IEEE Proc. Design Automation Conf.,* pp. 684–689, June 2001.
6. Luca Benini and Giovanni De Micheli, *Networks on Chips: A New SoC Paradigm, IEEE Computer,* Vol. 35, pp. 70–78, 2002.
7. Taylor, M.B. et al., The Raw microprocessor: a computational fabric for software circuits and general-purpose programs, *IEEE Micro,* Vol. 22, Issue 2, pp. 25–35, March-April 2002.
8. Ju-Ho Sohn et al., A 155-mW 50M vertices/s graphics processor with fixed-point programmable vertex shader fro mobile applications, *IEEE J. Solid-State Circuits,* Vol. 41, Issue 5, pp. 1081–1091, 2006.
9. Intel Xscale Processor, http://www.intel.com/design/intelxscale/.
10. AMBA AXI Specification.
11. OCP 2.0 Protocol Specification.
12. STBus Functional Specs, STMicroelectronics, public web support site, http://www.stmcu.com/inchtml-pages-STBus_intro.html, STMicroelectronics, April 2003.
13. Kees Goossens et al., Æthereal Network on Chip: Concepts, Architectures, and Implementations, *IEEE Design & Test of Computers,* pp. 414–421, September-October 2005.
14. Hubert Zimmermann, OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications,* Vol. 28, no. 4, April 1980, pp. 425–432.
15. Kangmin Lee et al., A 51 mW 1.6 GHz on-chip network for low-power heterogeneous SoC platform, *IEEE Int. Solid-States Circuits Conference, Digest of Technical papers,* pp. 152–518, February 2004.
16. Se-Joong Lee et al., An 800 MHz star-connected on-chip network for application to systems on a chip, *IEEE Int. Solid-States Circuits Conf., Digest of Technical papers,* pp. 468–469, February 2003.
17. Worn, F., Lenne, P., Thiran, P., and De Micheli G., A robust self-calibrating transmission scheme for on-chip networks, *IEEE Transactions on VLSI Systems,* Vol. 13, Issue 1, pp. 126–139, 2005.
18. Ivan Miro Panades and Alain Greiner, Bi-synchronous FIFO for synchronous circuit communication well suited for Network-on-Chip in GALS architectures, *Proc. 1st IEEE/ACM Int. Symp. on Networks-on-Chip,* pp. 83–92, May, 2007.
19. Rijpkema E., Trade offs in the design a router with both guaranteed and best-effort services for networks on chip, Design, *Automation and Test in Europe Conference and Exhibition,* pp. 350–355, 2003.
20. Kwanho Kim et al., An arbitration look-ahead scheme for reducing end-to-end latency in networks on chip, *Proc. IEEE Int. Symp. on Circuits and Systems,* pp. 2357–2360, 2005.
21. Wingrad, D., MicroNetwork-Based Integration for SOCs, *Proc. Design Automation Conf.,* pp. 63–677, June 2001.
22. M.Millberg, et. al., The Nostrum backbone—a Communication Protocol Stack for Network on Chip, *Proc. Int. Conf. on VLSI Design,* pp. 693–696, 2004.
23. Kangmin Lee et al., Low-power network-on-chip for high-performance SoC design, *IEEE Transactions on VLSI systems,* Vol. 14, pp. 148–160, February 2006.
24. Se-Joong Lee et al., Packet-switched on-chip interconnection network for system-on-chip applications, *IEEE Transactions Circuits and Systems II,* Vol. 52, pp. 308–312, June 2005.

25. Se-Joong Lee et al., Adaptive network-on-chip with wave-front train serialization scheme, in *IEEE Symp. on VLSI Circuits Digest of Technical Papers,* pp. 104–107, June 2005.
26. Kyusun Choi and William S. Adams, VLSI Implementation of a 256 × 256 Crossbar Interconnection Network, *Proc. IEEE 6th Int. Parallel Processing Symp.,* pp. 289–293, 1992.

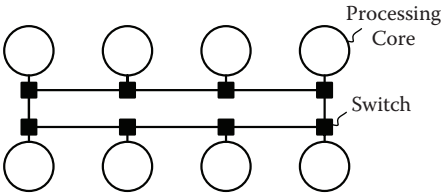
6 NoC Topology and Protocol Design

6.1 INTRODUCTION

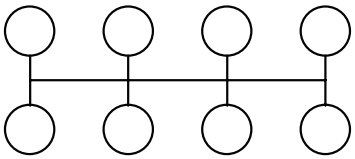
A topology is the connection map between the constituent processing elements (PEs). There exist many network topologies, and the simplest one is a ring or a shared bus. A three-dimensional cube or torus is a more complicated topology, but a mesh topology is widely considered as a typical NoC topology, especially for the homogeneous SoC. The mesh topology, however, is neither a unique nor an optimal solution for the heterogeneous multiprocessor. You should choose an optimum topology for your system and application in terms of the system performance, power-consumption, performance/power, and area cost. Traditionally, interconnection networks can be categorized into two classes: direct and indirect network. The former provides a direct connection between processing nodes. On the other hand, in an indirect network, the communication between any two nodes has to be carried through some switches. Most of the NoC topologies are indirect networks even for the mesh topology. However, as the distinction between these two classes of networks is blurring, the categorization becomes meaningless [1].

Many network topologies have been proposed in terms of their graph-theoretical properties. Most of them were proposed for minimizing the network diameter for a given number of nodes and node degrees [1]. However, very few of them have ever been analyzed and implemented in NoCs. Figure 6.1 shows a few of the most famous topologies and also an application-specific one as examples. There has been research on NoC topology exploration. Murali et al. have developed a tool for automatically selecting an application-specific topology for minimizing average communication delay, area, and power dissipation [2]. Wang et al. explore the technology-aware topology of various meshes/tori [3]. Kreutz et al. present a topology evaluation engine based on a heuristic optimization algorithm [4]. In these works, the candidate pool of topologies was limited to the typical, regular and homogeneous topologies such as a mesh, torus, cube, tree, or multistage network.

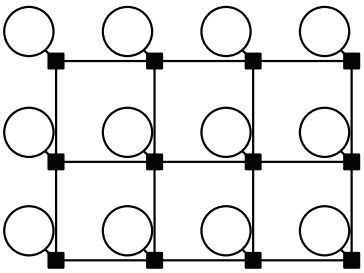
In heterogeneous SoCs such as embedded or mobile systems, however, communication flows are localized, not uniformly distributed [5]. In this case, it is highly possible that the optimal topology is a heterogeneous and hierarchical topology rather than a homogeneous and flat topology. For example, George et al. propose a hybrid interconnecting structure incorporating a point-to-point (locally), mesh (semiglobally), and tree (globally) topologies for a low-power FPGA application [6], which leads to the Pleiades chip implementation [7]. Therefore, we need to investigate such hierarchical and heterogeneous topologies in more detail.



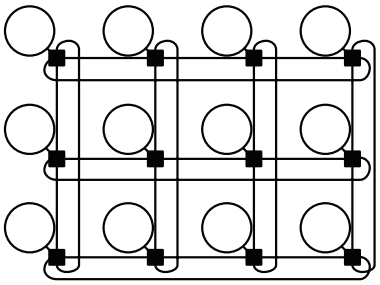
(a) ring topology or segmented bus



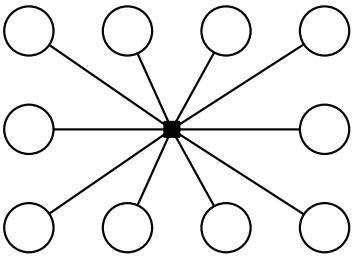
(b) shared-bus topology



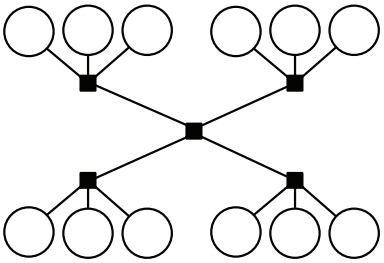
(c) mesh topology



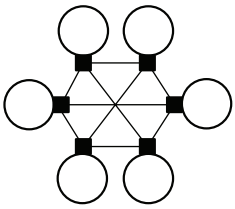
(d) torus topology



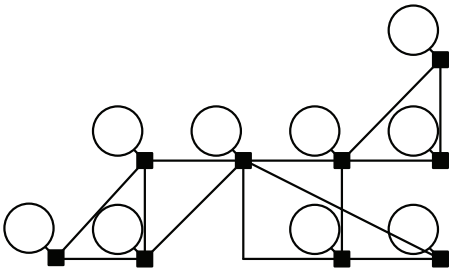
(e) star topology



(f) tree topology



(g) fully connected topology



(h) application specific topology

FIGURE 6.1 Typical topologies.

In this chapter, we present an analytic methodology to predict the energy consumption of various topologies—not only basic topologies, including bus, mesh, star, and point-to-point topologies but also hierarchical and hybrid topologies, for example, a hierarchical bus, local-star global-mesh, or local-bus global-star topologies. In this analysis, analytic models are proposed, and physical parameters depending on the process technologies and circuit designs are used. Later, we will look into how to design the protocol for your NoC. The design issues in link layer protocol and network layer protocol will be examined, and the packet structure decision will be explained.

6.2 ANALYSIS METHODOLOGY

6.2.1 Topology Pool and Target System

Topologies are categorized into two groups in this analysis: flat topologies and hierarchical topologies. Figure 6.2 shows flat topologies such as a bus, star, mesh, and point-to-point and also hierarchical topologies, for example, *local bus* and *global star, local star* and *global mesh,* and *local star* and *global star.* The hierarchical topologies consist of local and global network topologies in which the local and global networks can be of any type among the basic topologies. We comparatively analyze the four flat topologies and the sixteen hierarchical topologies in this chapter.

We assume that the size of each PE is uniform, 1 mm × 1 mm, and the PEs or processing cores are placed as a square matrix regardless of the topologies shown in Figure 6.2. A PE could be a single processor or a subsystem, such as a multimedia accelerator, a memory system, or an external interface. Each PE can behave as a

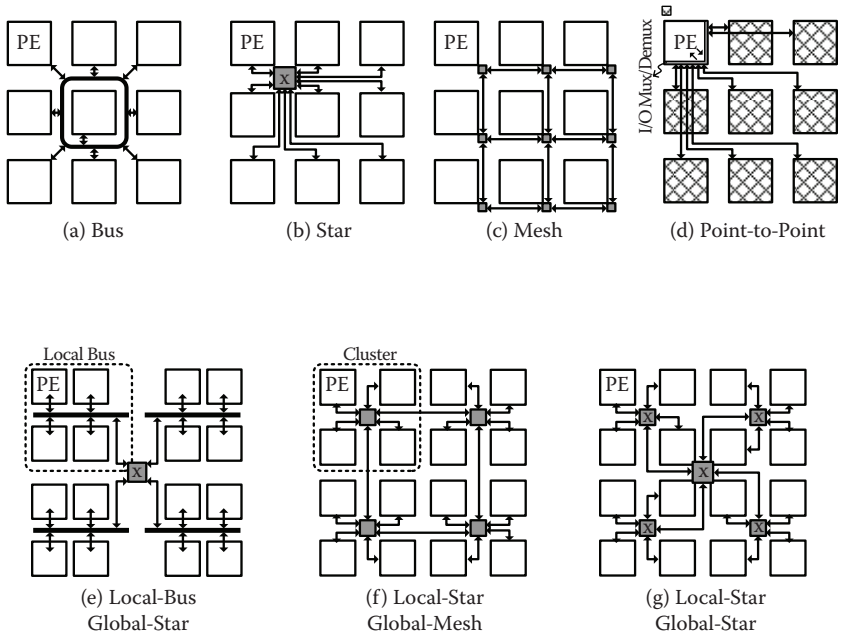


FIGURE 6.2 Flat and hierarchical and heterogeneous topologies.

master (initiator) or slave (target) depending on its operation. The number of PEs, N, scales from 16 to 100. The hierarchical topology is assumed to be divided into $\sqrt{N}$ of clusters and each cluster contains $\sqrt{N}$ of PEs.

6.2.2 NoC Traffic and Energy Models

The bus and point-to-point topologies do not have internal data buffers in their interconnection networks but could have them in their interface. Meanwhile, the star and mesh topologies have internal packet buffers in every switching hop. The buffer capacity of each switch is determined by considering the flow control mechanism and congestion level of the switch. The transaction unit is a packet, which is composed of a 16-bit header and a 64-bit payload (32-bit address and 32-bit data). The packet is serialized onto a unidirectional 10-bit link, which consists of 8-bit packet signals, a 1-bit STROBE signal as a timing reference, and a 1-bit end-of-packet signal [8]. We assume that the clocks of the integrated PEs are plesiochronous; i.e., PEs use different clock frequencies of their own and are not synchronized with each other.

There are two kinds of traffic patterns; one is uniform random traffic, and the other is localized traffic with a locality factor, α, whose value is between 0 and 1. The locality factor means the ratio of the intracluster traffic to the overall traffic, as illustrated in Figure 6.3. As α gets close to 1, the traffic becomes highly localized; i.e., most of the transactions occur within an intracluster (local network) domain. If α is 0.5, a half of the traffic is in the intracluster domain and the other half in the intercluster (global network) domain. It is obvious that PEs with low latency and large bandwidth communication can give more synergetic performance if placed in

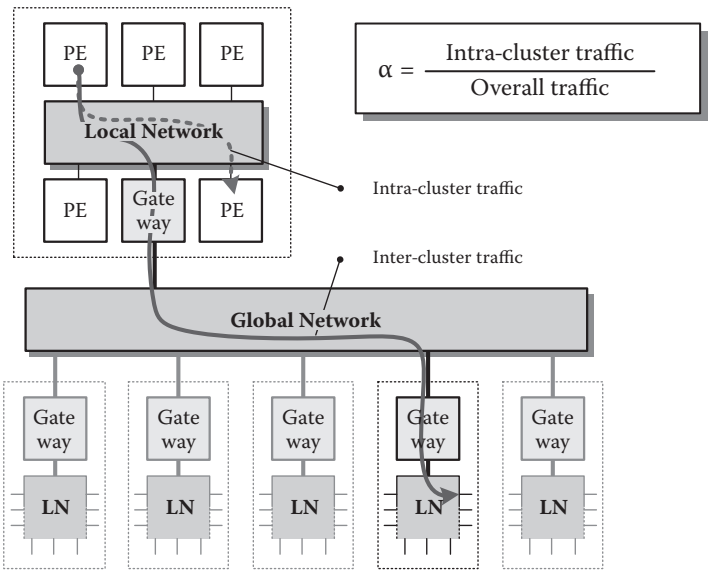


FIGURE 6.3 Traffic model for locality issues.

TABLE 6.1
Physical Parameters in 0.18 μm CMOS Technology

Category	Description	Typical Value	Symbol
Energy (J)/1-packet-traversal	Buffer (write/read)	1.97×10^{-10}	E_{Queue}
	Switching fabric/port	6.25×10^{-12}	E_{SF}
	2:1 multiplexer	3.04×10^{-12}	E_{MUX}
	Arbitration/port	1.79×10^{-13}	E_{ARB}
	1-mm link	4.38×10^{-11}	E_{Link}
	1-mm link (P-to-P)	8.76×10^{-11}	E_{Link_PtP}
Dimension (mm^2)	Processing element (PE)	1×1 mm^2	$(L_{PE})^2$
	Cluster unit	$\sqrt[4]{N} \times \sqrt[4]{N}$ mm^2	$(L_{CU})^2$
Area (μm^2)	Three-packet queuing buffer	8.40×10^4	A_{Queue}
	Crossbar-fabric	$1.47 \times 10^3 \times$ (# of s/w ports)^2	A_{SF}
	M:1 10b multiplexer	$9.52 \times 10^2 \times$ (M−1)	A_{MUX}
	Arbitration logic	$2.70 \times 10^3 \times$ (# of s/w ports)	A_{ARB}
	20b 1-mm link	3.80×10^4	A_{Link}
Latency (ns)	Arbiter	$0.33 \times \log_2$(# of s/w ports)	T_{ARB}
	Switching fabric	6.5×10^{-3} x (# of s/w ports)^2	T_{SF}
	1-mm link (repeated link)	0.42	T_{Link}

Source: From Kangmin Lee et al., A 51 mW 1.6 GHz On-Chip Network for Low-Power Heterogeneous SoC Platform, in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers,* 2004, pp. 152–153.

the same cluster, based on their communication spatial locality, so that the intracluster traffic becomes dominant. In such a heterogeneous system, the locality factor can represent the localized traffic pattern quantitatively.

We use an average packet traversal energy, E_{pkt}, as the network energy efficiency metric, which can be estimated by the following equation, summing up the energies on switching hops, links, and the final destination buffer [9].

$$E_{pkt} = H_{Avg} \cdot (E_{Queue} + E_{ARB} \cdot SS_{Avg} + E_{SF} \cdot SS_{Avg}) + L_{Avg} \cdot E_{Link} + E_{Queue} \quad (6.1)$$

where H_{Avg} and L_{Avg} are the average hop counts and average distance, respectively, between the sender PE and the receiver PE. SS_{Avg} is an average switch size, i.e., the number of I/O ports in a switch. Energy consumption on a switching hop is composed of energy consumption in an input queuing buffer or latch, $E_{Queue,}$ switching fabric, E_{SF} and arbitration logic, $E_{ARB.}$ E_{Link} stands for transmission energy on a unit-length link. These energy terms are measured from the circuit implementation in 0.18-μm technology, as shown in Table 6.1.

6.3 ENERGY EXPLORATION

6.3.1 Bus Topology

A conventional Mux-based bus structure [10] has two unidirectional buses; one is from masters to slaves and the other is from slaves to masters, as shown in Figure 6.4a. This master/slave bus topology cannot support direct messages passing between two masters; so they share a memory to communicate with each other. To cope with the limitation of connectivity, a fully connected bus can be used, as illustrated in Figure 6.4b. It has a single shared bus, which connects all of the PEs regardless of the types of masters/slaves. Figure 6.4c shows a hierarchical bus structure in which local buses are master/slave buses and a global bus is a fully connected bus. By using the fully connected bus in a global network, direct access between two clusters is possible without a shared memory.

The following equations show the E_{pkt} of each bus in Figure 6.4.

a. Master/slave bus (the number of masters: *N/2*; the number of slaves: *N/2*):

$$E_{pkt}^{MSB} = \left\{E_{ARB} \cdot \frac{N}{2} + E_{MUX} \cdot \left(\frac{N}{2} - 1\right)\right\} + \left[E_{Link} \cdot \left\{\frac{1}{2}\sqrt{\frac{N}{2}} + \frac{N}{2}\right\} \cdot L_{PE}\right] + E_{Queue} \tag{6.2}$$

The multiplexer and arbiter on the M‡S bus have *N/2* inputs. The average distance from a master to the multiplexer can be derived as $1/2\sqrt{N/2} \cdot L_{PE}$, and the length of the shared bus is $N/2 \cdot L_{PE}$.

b. Fully connected bus:

$$E_{pkt}^{FB} = \left\{E_{ARB} \cdot N + E_{MUX} \cdot (N-1)\right\} + \left[E_{Link} \cdot \frac{3(N-1)}{2} \cdot L_{PE}\right] + E_{Queue} \tag{6.3}$$

The multiplexer and arbiter on the bus have *N* inputs. The average distance from a master to the last multiplexer is $(N-1)/2 \cdot L_{PE}$, and the length of the shared bus is $(N-1) \cdot L_{PE}$.

c. Hierarchical bus:

$$E_{pkt}^{HB} = E_{pkt\_Local}^{MSB} \cdot \alpha + (2 \cdot E_{pkt\_Local}^{MSB} + E_{pkt\_Global}^{FB}) \times (1-\alpha) + E_{Queue} \tag{6.4}$$

$E^{HB}{}_{pkt}$ can be derived by summing the local and global traverse energy according to the traffic locality factor. The local and global traverse energy can be obtained from Equation 6.2 and 6.3, respectively, by replacing *N* with $\sqrt{N}$.

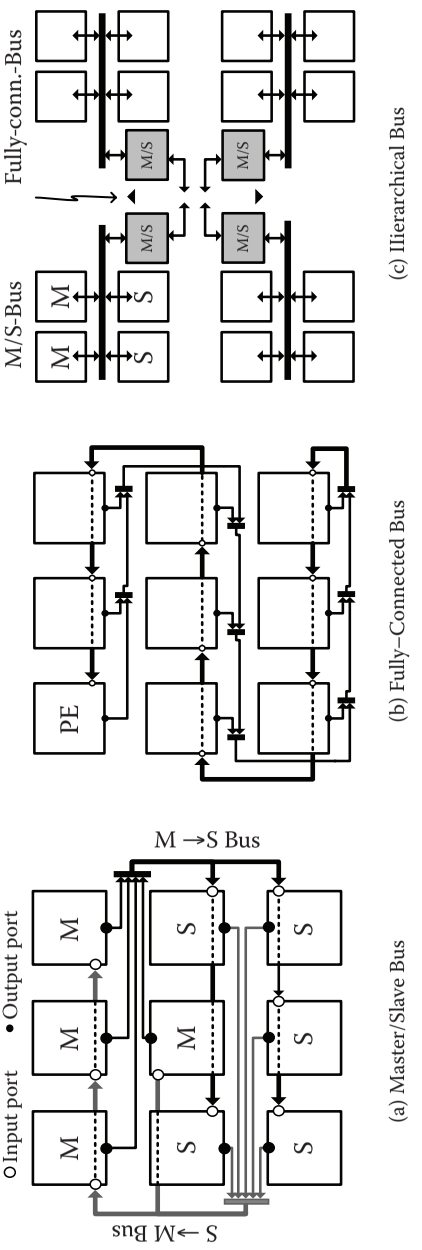


FIGURE 6.4 Various bus topologies.

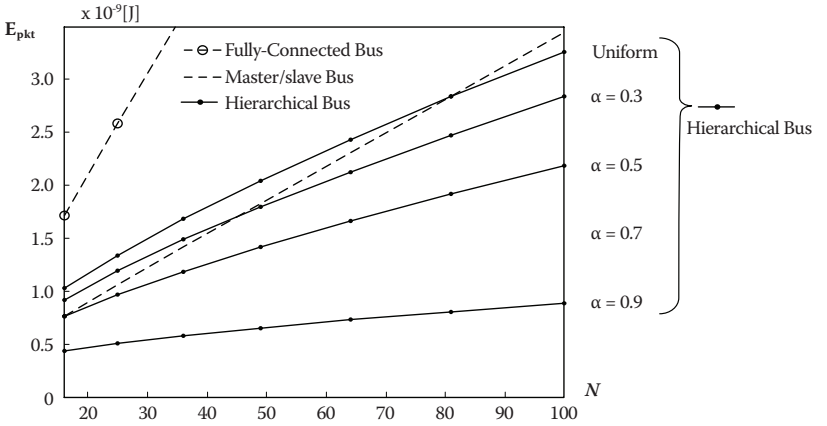


FIGURE 6.5 Energy consumption of bus topologies.

Figure 6.5 shows the E_{pkt} of the three buses according to the number of PEs with various traffic patterns. Under uniform traffic, the flat master/slave bus outperforms the hierarchical bus. However, as the traffic gets localized, i.e., *realistic*, the energy consumption of the hierarchical bus is significantly reduced. As expected, a hierarchical bus, rather than a flat bus, has the best energy efficiency under localized traffic.

6.3.2 Mesh Topology

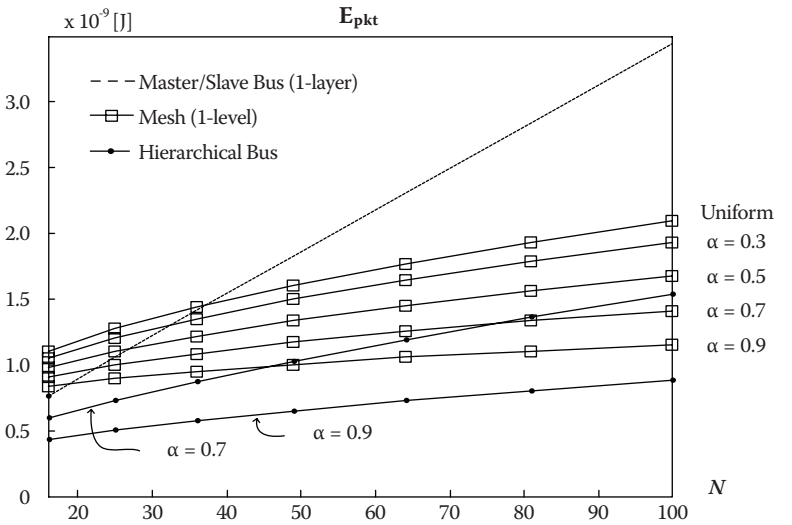
The following equation shows the E_{pkt} of a 2-D flat mesh:

$$E_{pkt}^{FM} = \left(\frac{2}{3}\sqrt[4]{N}\cdot\alpha + \frac{2}{3}\sqrt{N}\cdot(1-\alpha)\right)\times \left[\left\{E_{Queue} + (E_{ARB} + E_{SF})\cdot SS_{Avg}\right\} + E_{Link}\cdot L_{PU}\right] + E_{Queue} \quad (6.5)$$

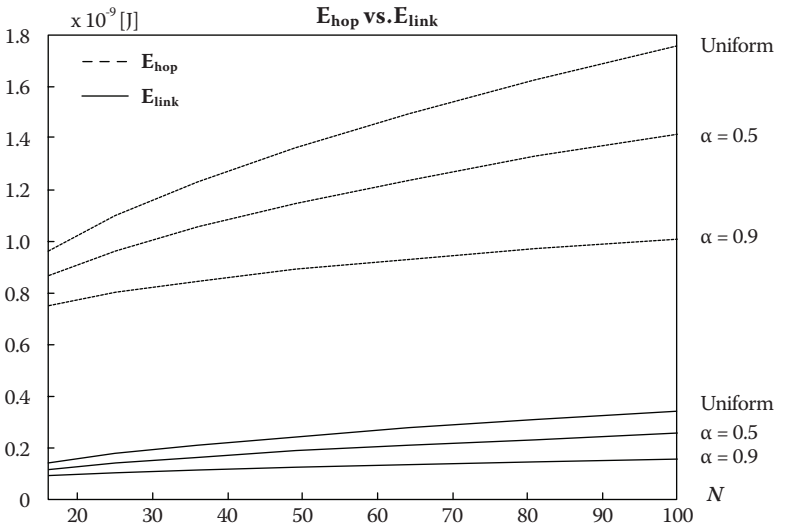
In the equation, $2/3\cdot\sqrt[4]{N}$ and $2/3\cdot\sqrt{N}$ are average hop counts of local and global transactions, respectively.

Figure 6.6a shows the $E^{FM}{}_{pkt}$ compared with bus topologies. Under uniform traffic, the mesh topology shows better energy efficiency than the master/slave bus does when N is larger than 36. As the traffic gets localized, the energy consumption of the mesh decreases, but it is still higher than that of the hierarchical-bus topology. The mesh topology shows flatter slopes than bus topologies do as the size of the network increases. It is known to be more scalable than the bus topology. This work reveals the energy trends with quantitative figures.

It is also interesting to compare the energy consumption of hops (switches) and links, as shown in Figure 6.6b. In mesh topology, the energies on hops are much higher than the energies on links, about 5 to 8 times, in this implementation condition.



(a) E_{pkt} in a mesh compared with Buses



(b) E_{hop} vs. E_{link} in a mesh

FIGURE 6.6 Energy consumption of mesh topologies.

6.3.3 Star Topology

In a star topology, the hop count is always 1 and every transaction goes through the central crossbar switch. The following equation represents the E_{pkt} of a flat star topology:

$$E_{pkt}^{FS} = (1) \times \left\{ E_{Queue} + E_{ARB} \cdot N + E_{SF} \cdot N \right\} + E_{Link} \times \left(\sqrt{N} - 2 \right) \times L_{PU} + E_{Queue} \tag{6.6}$$

The central switch has a number of N I/O ports and the average distance between two PEs via the central switch is $(\sqrt{N} - 2) \cdot L_{PU}$. The energy of a hierarchical star is given by the following equation:

$$E_{pkt}^{HS} = E_{pkt\_Local}^{S} \cdot \alpha + (E_{pkt\_Local}^{S} + E_{pkt\_Global}^{S}) \cdot (1 - \alpha) \tag{6.7}$$

The local and global energy, $E^{S}_{pkt\_Local}$ and $E^{S}_{pkt\_Global}$, can be obtained from Equation 6.6, by replacing N with $\sqrt{N}$. In the case of a global network, L_{PE} also should be replaced by L_{CU}.

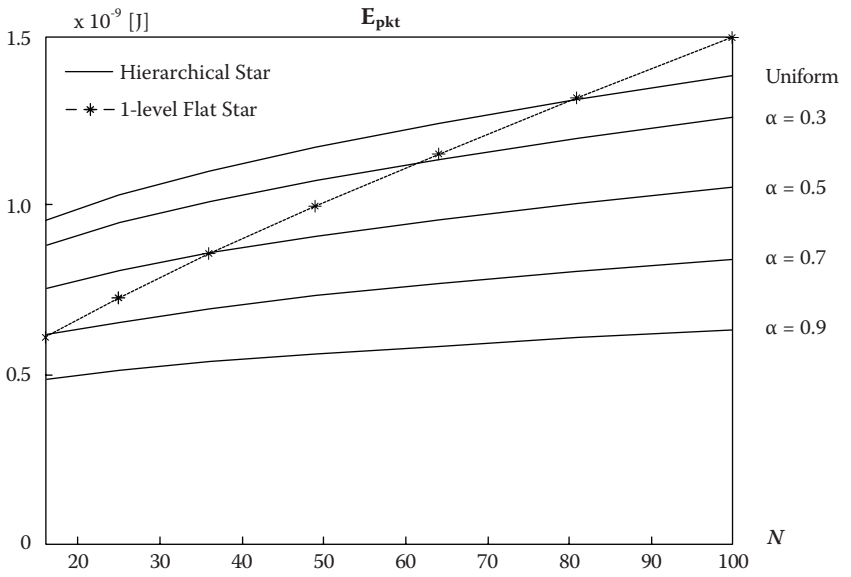
Figure 6.7a shows a comparison of E^{FS}_{pkt} and E^{HS}_{pkt}. When the traffic is less localized and N is less than a few tens, the flat-star topology shows higher energy efficiency because of less hop count than the hierarchical-star topology. However, as the traffic gets localized ($\alpha \geq 0.7$), the hierarchical-star outperforms the flat-star topology. Moreover, the hierarchical star shows a very flat energy profile with increasing network size, so that it is highly scalable in terms of energy cost. Figure 6.7b shows the energy comparison between switches and links.

6.3.4 Point-to-Point Topology

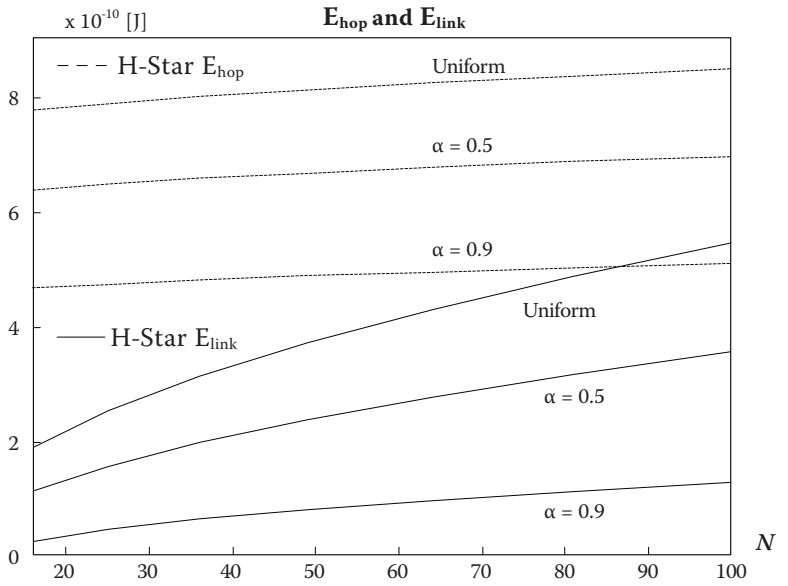
Point-to-point topology has a dedicated link between each pair of PEs, as shown in Figure 6.2d. It provides the shortest link length without any intermediate switches between a sender and a receiver, and thus shows the lowest energy consumption among all topologies. However, it suffers from a huge link area and large number of input/output ports. The following equation describes the E_{pkt} of the flat point-to-point topology:

$$\begin{aligned} E_{pkt}^{FP} = {} & \left\{ E_{ARB} \cdot (N-1) + E_{MUX} \cdot (N-2) \right\} \times 2 \\ & + E_{Link\_PtP} \cdot \left\{ \left(\frac{2}{3} \sqrt[4]{N} \right) \cdot \alpha + \left(\frac{2}{3} \sqrt{N} \right) \cdot (1 - \alpha) \right\} \cdot L_{PU} + E_{Queue} \end{aligned} \tag{6.8}$$

Each PE has 1:(N–1) demultiplexer and (N–1):1 multiplexer in its input/output port, respectively. The average link length of a point-to-point topology is the same as that of a mesh topology—the shortest Manhattan distance.



(a) E_{pkt} in a hierarchical star and a 1-level flat star



(b) E_{hop} and E_{link} of a hierarchical star

FIGURE 6.7 Energy consumption of star topologies.

6.3.5 Heterogeneous Topologies

In the previous sections, we have analyzed the basic topologies—bus, mesh, star, and point-to-point—as well as some hierarchical but homogeneous ones such as hierarchical-bus and hierarchical-star topologies. It was proved that the hierarchical topologies show better energy efficiency and scalability than flat topologies. Therefore, it is worthwhile to examine the energy efficiency of other hierarchical and heterogeneous topologies, for example, a local-star global-mesh or a local-bus global-star topology. To consider such hierarchical and heterogeneous topologies, we evaluate the basic topologies in two hierarchical domains, a local (intracluster) domain and a global (intercluster) domain.

Figure 6.8 shows a comparison of energy efficiency in a local and global network. In a local network, the energy cost on a link is much lower than that in a switch because of the shorter distance between the communicating nodes. Because a mesh topology has a larger number of hops than others, it shows the highest energy cost. On the other hand, in a global network, the energy cost on a link becomes significant. As a result, the energy cost of the bus topology, which has the longest wires, increases. Considering both the local and global networks, the point-to-point topology is the most energy efficient, and the star topology is the next.

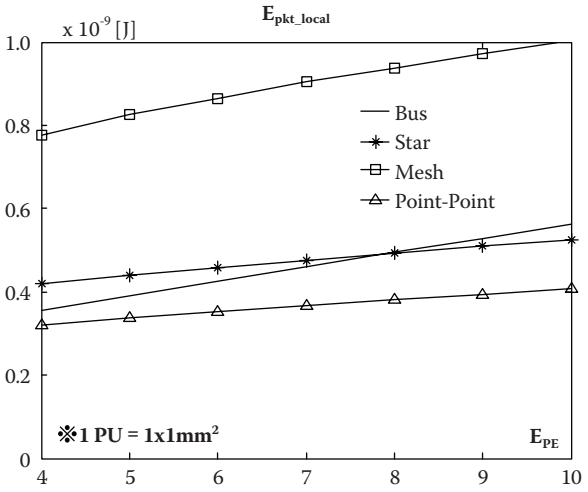
All of the topologies are compared under various traffic conditions in Figure 6.9. In any traffic condition, the point-to-point topologies show the best energy efficiency. If the point-to-point topologies cannot be adopted because of its infeasibility, the performance of star topologies is the best among the others. If N is fixed as 36 (see Figure 6.9c), for instance, the flat star is the best for less localized traffic, whereas the hierarchical star (L-star G-star) is the best for more localized traffic ($\alpha > 0.5$). The mesh always consumes about 30 to 80% more energy than the hierarchical star does. The mesh outperforms the hierarchical bus by about 10 to 20% in less localized traffic, but the hierarchical bus shows much better energy efficiency than the mesh in highly localized traffic.

6.4 NOC PROTOCOL DESIGN

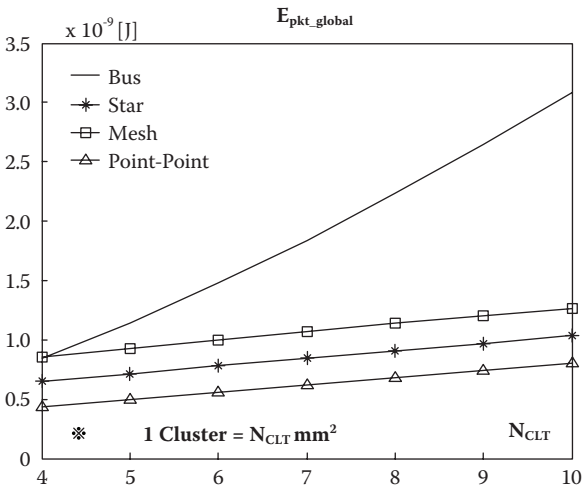
A protocol is a set of conventions that governs the communication between two or more entities for interaction. The term *entities* stands for anything that is capable of transmitting or receiving information. For two or more entities to communicate with each other, they must speak in a common language or, at least, have appropriate translators; in addition, they must have proper rules to start, stop, and maintain the communication—this is called the protocol.

To design a protocol, we should consider the following functions [1]:

1. Segmentation and reassembly—Bit streams will be broken into several block units, which are called protocol data units (PDUs). There are pros and cons for segmentation. It allows smaller buffer size, depending on the block size, and increases the efficiency of error control. Also, synchronization becomes easier with segmented data blocks, with regular start and stop of frames. However, there are disadvantages of segmentation, too. Because



(a) Number of PEs in a cluster



(b) Number of clusters on a chip

FIGURE 6.8 Energy consumption of (a) local networks and (b) global networks.

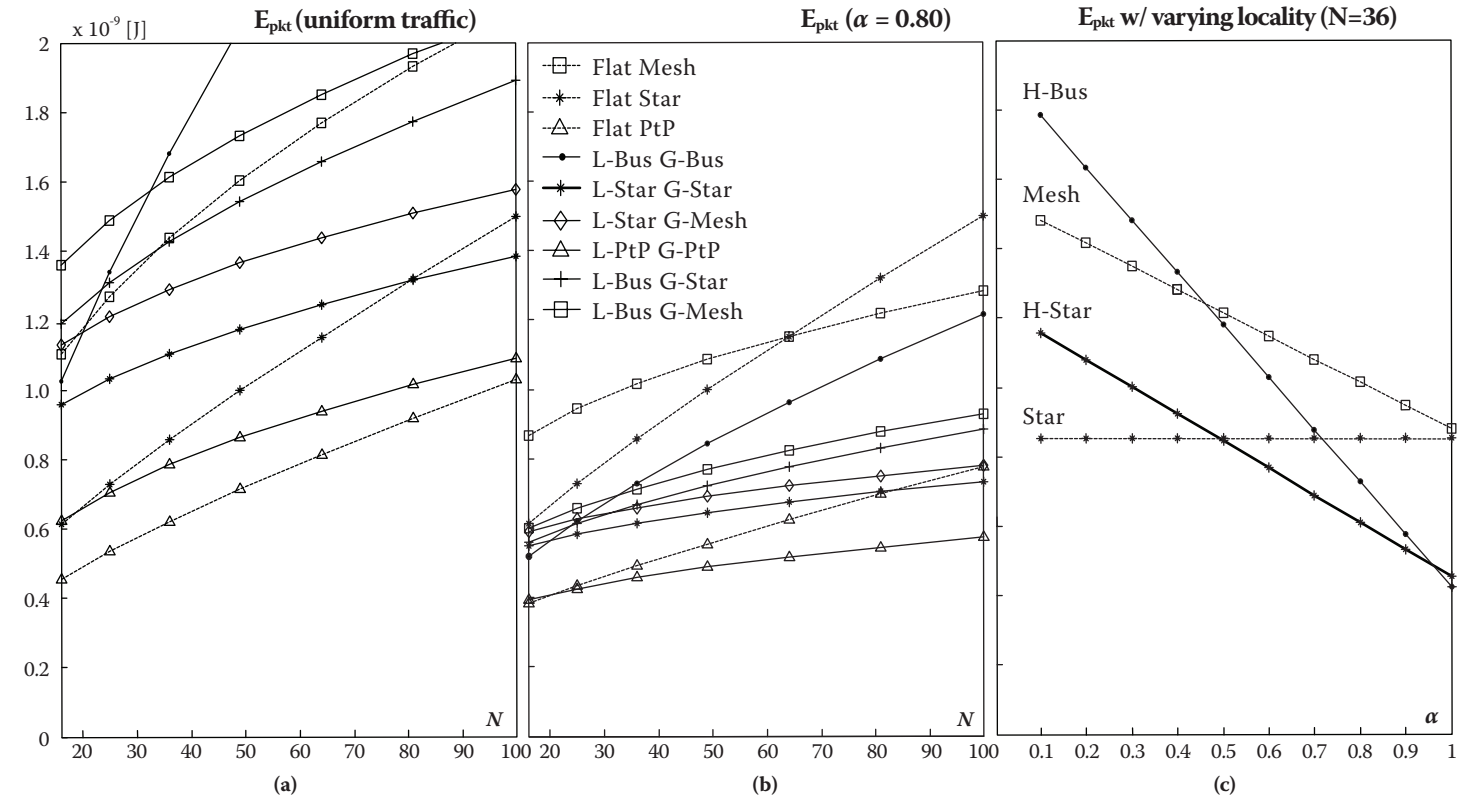


FIGURE 6.9 Energy consumption of all topologies.

each block includes some control information, the smaller the block size, the more the overheads. In addition, processing the control information may increase computational power. Therefore, a protocol designer should trade off between segmented block size and resource overheads. The opposite term of segmentation is blocking; when a message size is too small, and it is more efficient to transfer one with a larger block size, we can combine several blocks into a bigger one before transfer. This is called *blocking*. As an example of NoC, the KAIST BONE™ series, as will be shown later in this chapter, has a unit block 88 bits long.

2. Encapsulation—Each block is composed of control information and data. As we will see later in this chapter, the higher-layer PDU is encapsulated in current layer control information. This encapsulated block is another PDU for the lower layer.
3. Connection control—In connectionless data transfer, the transfer occurs without prior coordination between entities. However, for reliable transmission, connection-oriented data transfer is preferred, for which the following three steps must exist:
   - Connection setup
   - Message transfer
   - Disconnect

   A protocol designer should carefully set up the connection control rule.
4. Flow control—The flow control provides backpressure mechanisms to control the transfer rate or amount according to the status of the destination, which, usually, refers to buffer size and buffer availability.
5. Error control—For reliable transmission, a protocol should have an error control strategy. When error occurs during data transfer, either error correction or retransmission of PDU is required to be done. Automatic Repeat Request (ARQ) is an example of error control based on retransmission, and there are many types of ARQs, such as Stop-and-Wait, Go-Back-N, Selective Repeat, and so on. A designer should select the appropriate type of error control scheme. If you adopt a low-voltage signaling for a power-efficient NoC design, the signal integrity becomes worse. Therefore, the error control scheme will be necessary in the NoC protocol stack. The error control codes such as CRC, Viterbi, and Turbo are well developed in wire-line and wireless communications. However, these codes need heavy computation at the receiving entity and many bits of redundancy on the communication wires. These heavy codes are not appropriate for the on-chip network because they consume unacceptable latency, on-chip wire, and logic resources. Lightweight and low-latency error control schemes are crucial for on-chip communications.
6. Synchronization and timing recovery—Destination application must reconstruct temporal relation from the input stream to properly recovered transmitted data because network transfer inevitably introduces delay and jitter. Without synchronization and timing recovery, the error rate increases and data integrity is not guaranteed. Timing recovery protocols use certain time stamps and sequence numberings to control the delay and jitter of the information.

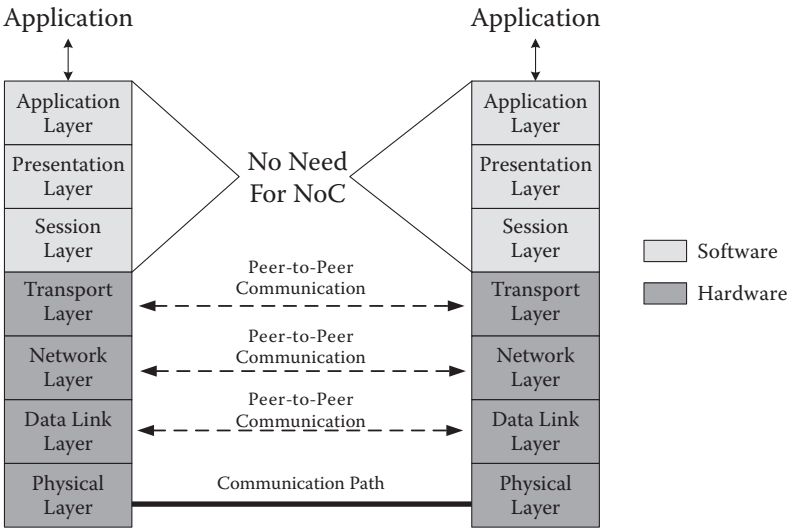


FIGURE 6.10 The OSI reference model.

7. Multiplexing—In layered architecture, as shown in Figure 6.10, multiplexing enables layer n + 1 users to share the layer n service. For multiplexing, a tag is required to identify specific users at the destination—this is called the *multiplexing tag*.

6.4.1 Layered Architecture

Because the overall communication process between multiple machines connected across multiple networks is very complex, the layering concept is introduced, as we have examined in Chapter 5. Layered architecture contributes to hierarchical modularity. The module or function performs a given behavior in support of the overall behavior of the system. Such a behavior is often called the *service* provided by the function. Layering partitions communication functions into *manageable* groups, and each layer provides a service to the earlier layer. With the aid of layering, we can simplify the design, implementation, and testing, and the protocol in each layer can be designed separately from those in other layers. In NoC, PEs are largely heterogeneous, with different functions or different models. Generally, we cannot adopt hardware blocks developed for a specific chip to another chip instantly because different intellectual properties (IPs) use different data formats or different conventions. Even if the designers are the same, different IPs may use their own types of data. With hundreds of thousands of different types of IPs under operation, it is clear that it is practically impossible to make a single unit to cope with such variability; it must be partitioned into manageable parts. We can use the International Organization for Standardization's (ISO) Open Systems Interconnection (OSI) reference model, adopted for data communication in 1983 [12], for the NoC, too. The OSI model gives standards for linking heterogeneous computers.

Figure 6.10 shows the seven-layer OSI reference model. In the case of NoC, the session and the presentation layers are incorporated into the application layer and are

out of the scope. These layers are handled inside the IPs, which are outside of the on-chip interconnection network.

Two peer modules communicate using a lower-layer module as the communication system. There are two distinctive aspects that are exploited for communication between two modules at the same layer N. The first is the protocol (or distributed algorithm) that the peer modules use in exchanging messages or data so as to provide the required functions or services to the next higher layers, as shown in Figure 6.11a. The second is the interface specification between the layer N module and layer N–1 module at the same node through which the messages are exchanged, as shown in Figure 6.11b.

Therefore, by adopting layered architecture, each layer performs its own function independent of other layers and provides a defined service to the previous layer, which operates according to a defined protocol. With this scheme, we can easily implement complex communication systems simply by adding layers between the application layer and the lower link layer.

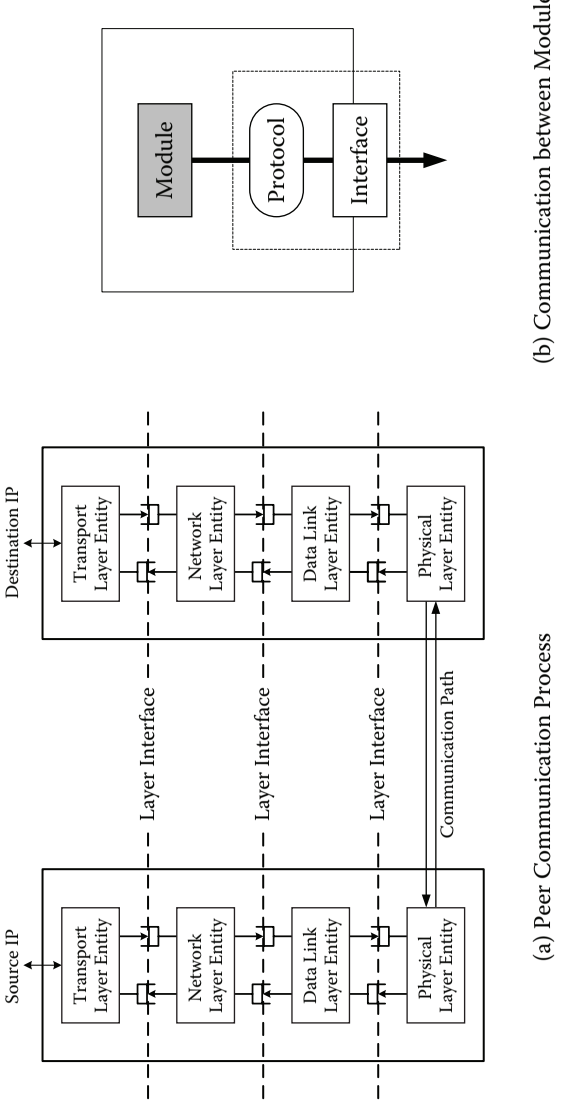
From now on, we will look closer into OSI layers from the view of the NoC (see Figure 6.12).

6.4.2 Physical Layer Protocol

In the traditional computer network, the physical layer defines electrical and physical specifications between the communicating devices. In the case of NoC, significance of the physical layer as a common protocol is not emphasized because all the on-chip devices are readily designed with the same process technology and supply voltage. However, the power- and area-efficient design of the physical layer is very important due to its impact on overall NoC performance. It is common wisdom that the amount of on-chip traffic is about a few orders of magnitude larger than that of off-chip traffic. Therefore, power-inefficient designs of the physical layer result in enormous power consumption, which makes heat sink solutions very costly. In addition, because of the limited silicon resources, a simple circuit implementation without complex control is usually preferred. One advantage of the NoC design is that the cost of wire is relatively cheap compared to the off-chip interconnection. Therefore, a trade-off among power consumption, silicon area, and wire latency is possible. In this subsection, we will briefly discuss possible trade-offs applicable to each design aspect of the physical layer.

Power: By controlling the swing voltage of long wire interconnections, it is possible to trade-off power consumption with signal integrity. Although a low voltage swing reduces power consumed on an interconnection link, the wire link becomes more sensitive to the on-chip noise. This trade-off also influences maximum bandwidth of on-chip wires.

Area: By adopting differential signaling, signal integrity can be improved at the cost of a larger on-chip area occupied by the wires. It is also possible to trade-off wire spaces with signal integrity, because larger wire space contributes to the reduction of coupling capacitance among wires.



(a) Peer Communication Process (b) Communication between Modules

FIGURE 6.11 Layers: (a) Peer communication process and (b) communication between modules.

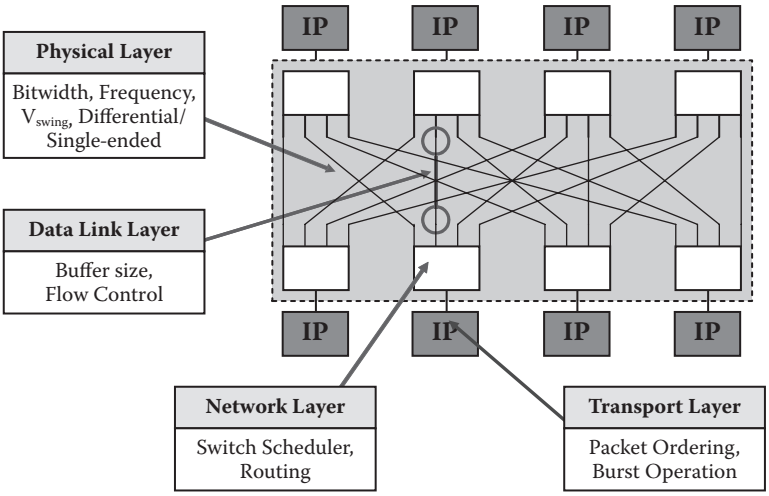


FIGURE 6.12 Layered architecture of NoC.

Latency: Depending on the choice between voltage sensing and current sensing schemes, latency and power consumption can be traded off. The current sensing scheme usually consumes more static power because of its bias current while providing low latency.

6.4.3 Data Link Layer Protocol

The data link layer transfers frames across direct connections; it groups bit streams into frames and checks bit errors by checksum. It is within the data link layer that retransmission of frames occurs when errors are detected. However, when it comes to NoC, error detection or error correction schemes are rarely used because of the limited resources, except for very-high-speed links. Instead, the physical layer reliability is improved by exploiting intensive SPICE simulations because such approaches are more convenient and it is easier to control the electric parameters in NoC.

Flow control is another key property of the data link layer. Although the general network also has flow control within the transport layer (e.g., TCP), NoC mainly uses flow control within the data link layer, with buffer size and queuing control. However, retransmission, which allows packet losses, is too costly and unsuitable for NoC, so we often use the flow control scheme, which does not allow packet losses. This is mainly because on-chip data transactions are well over off-chip data transactions. Besides, in NoC, the queuing buffer size is constrained to hold only a few packets. Therefore, if the backpressure flow control scheme is adopted, we must consider the backward transmission time of flow control signals and insert the appropriate amount of buffers. Credit-based flow control, on the other hand, has relatively low buffer utilization but is more reliable.

Figure 6.13 shows two switches within the NoC, and data link layer connects them through the queuing buffer. Here, we should consider effective bandwidth, and

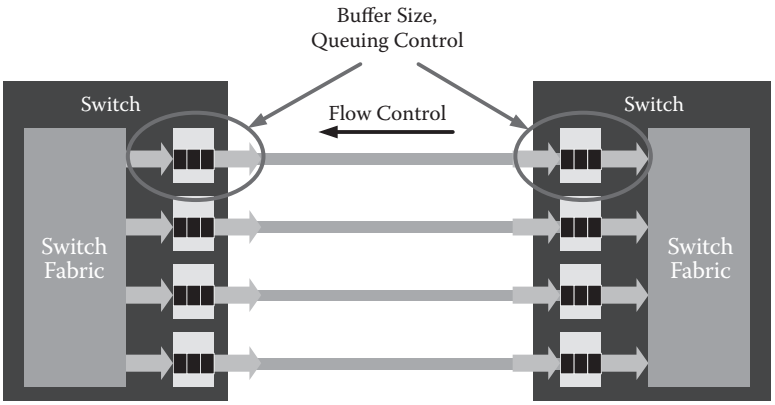


FIGURE 6.13 Data link layer.

the traffic pattern determines the buffer requirement. In other words, an effective bandwidth is determined by traffic pattern and queue size.

6.4.4 Network Layer Protocol

The network layer, which is the third stack of the OSI seven-layer reference model, is responsible for the transfer of packets across multiple links or multiple networks. In the case of NoC, design issues of the network layer include the routing scheme, arbitration policy for congestion control, and topology construction methodology. Because it is expected that the scale of most SoC designs would be less than a few thousand cores [Landscape], supporting huge scalability for millions of cores is not required. However, power/area efficiency and modular design for each building block of the NoC is still emphasized as with other lower-level OSI layers. In relation to the NoC protocol design, the routing scheme is closely related to the packet header format, which implicitly specifies manipulation of routing information. On the other hand, the arbitration policy for congestion control is generally left unspecified, as the designer's choice, so that it is tailored to the target application. In this subsection, we will investigate the design issues of the network layer in detail with the specific implementation example of NoC, the KAIST BONE ™ series [13]. BONE is the acronym for *B*asic *O*n-chip *NE*twork, which puts emphasis on power and area efficient implementation rather than complex and fancy schemes.

The BONE series adopt source routing for low-overhead and architecture-independent routing schemes. In BONE implementations, a packet is transferred to a destination according to the routing information (RI) field in the header. Then, a response packet of the destination, if necessary, is returned to the source along the reverse path. Each switch output port has its own port index, as shown in Figure 6.14a. The RI field of the header contains a series of port indexes, which are called output port indexes (OPIs). An OPI_k represents an index of a switch port to which the packet must be routed by the *k*th switch. When a packet arrives at a switch, the switch fetches an OPI at the RI field of the header. Then, the RI field is shifted to locate the next OPI at the most-significant-bit part of the packet header, and the inverted

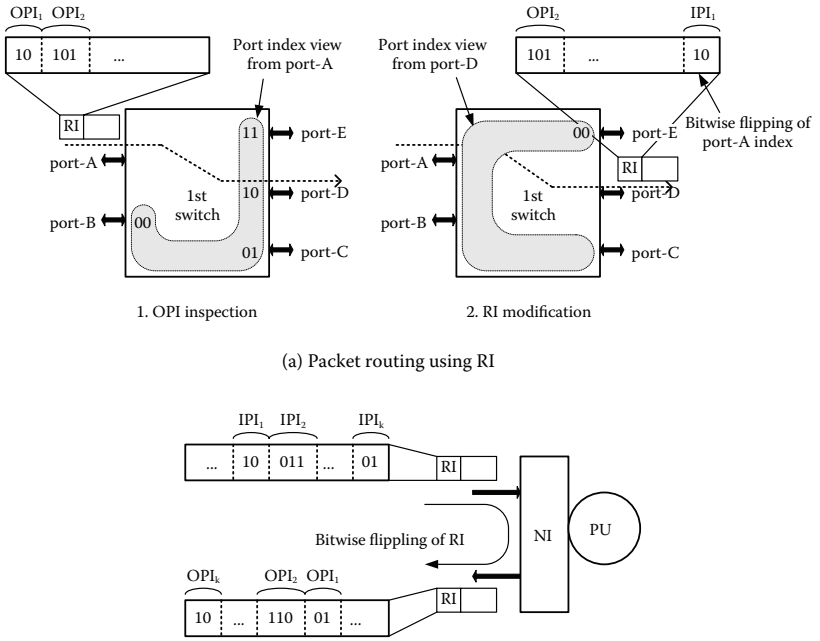


FIGURE 6.14 (a) Routing information (RI) modification mechanism and (b) RI flipping at destination network interface (NI).

port index (IPI) of the input port is attached to the tail of the RI field, as shown in Figure 6.14b. Using this RI field modification, each switch can find its OPI at the fixed location of the packet header, and a network interface (NI) of a destination PU obtains return path information by bitwise flipping of the RI field of the packet header. Because an IPI is a flipped index of an input port through which the packet has passed, the bitwise flipping outputs correct OPIs.

Using this routing scheme, switches do not include routing look-up tables (LUTs), which contain topology-dependent data. As a result, the core network design is decoupled from the system design, which makes it easy to construct various network topologies demanded by the target applications. If the NoC can be designed independently of SoC architecture, the building blocks of NoC are easily replicated and reused.

In the network layer, congestion control is generally provided by arbitration performed on each of crossbar switches. Congestion occurs when more than two functional modules—IPs in the case of NoC—are trying to send packets to the same IP or shared link at the same time. Figure 6.15 shows how arbitration is performed to resolve congestion in a crossbar switch. For example, packets from input port #2 and #N−1 request arbiter #2 for access to output port #2 at the same time. In response to the requests, the arbiter grants one input port according to the predefined scheduling algorithm. While packets from input port #2 are transferred through output port #2 after the grant signal is asserted, packets of input

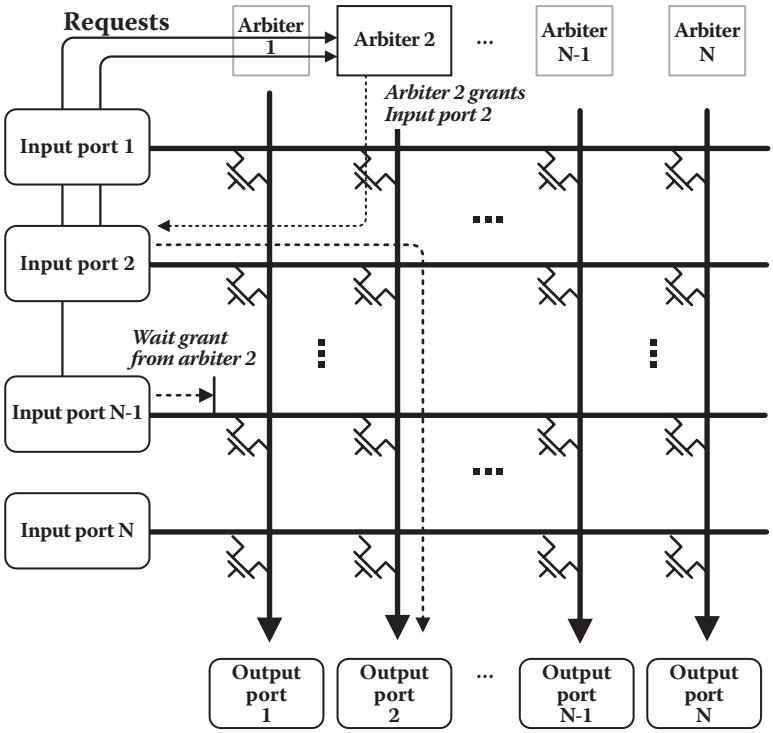


FIGURE 6.15 Arbitration of crossbar switch.

port #N–1 are waiting in their input queue. In the BONE series implementations, simple round-robin scheduling is preferred, because of its fairness and low power/area overhead, to complicated and high-latency scheduling algorithms such as iSLIP [18]. For a BONE implementation, the scalable and light-weight scheduler for crossbar switches is proposed [19].

6.4.5 Transport Layer Protocol

The transport layer provides transparent end-to-end data transactions among IPs. In the case of the NoC, the transport layer is usually implemented in the NI modules. The NI module performs packet parsing and generation to support abstraction of the on-chip network. Traditional features, such as packet reordering and retransmission are not supported in many NoC implementations because of the huge overheads in the implementation of hardware on a silicon chip.

In this subsection, with a case study of BONE implementations, four special functions for the end-to-end packet transaction in NoC are described: (a) Multiple-outstanding-addressing (MOA), (b) write with acknowledge, (c) burst packet transfer, and (d) enhanced burst packet transfer are defined to provide high-bandwidth, energy-efficient, and reliable packet delivery. The detailed description of each function follows.

6.4.5.1 Multiple-Outstanding-Addressing

When a source PU issues a read command to a destination, MOA enables the additional issues of the read commands before the previous read transaction completes. The MOA scheme, which is a well-known technique in advanced bus architectures [14], hides the latency of a communication channel and thus increases throughput. To use the MOA scheme, however, multiple FIFOs are required, as many as the MOA capability depth allows. In this work, NI controls it, as shown in Figure 6.16a. An NI inspects the destination of a read command packet, and if successive read command packets target the same destination, MOA is admitted. Because the network provides a single fixed path for a single destination in this work, packet sequence is not modified in the network. However, if the destinations of two packets are different, their latency would be different, so the orders of packet issue and arrival may not be the same. Therefore, if a following read command packet targets a different destination, the NI holds the packet transmission of the source and thus MOA is blocked. The NI holds the transmission until all the previous packet transactions are completed.

6.4.5.2 Write with Acknowledge

Because a network has variable latency, a source PU does not know when a write packet has arrived at the destination. In some applications, a PU must perform certain operations after writing data; therefore, it needs to know the timing of the write packet's arrival at the destination. For example, let us think about a situation in which a microprocessor transfers data to a memory and issues a command packet to a dedicated hardware unit to process the data in the memory. In this case, the command packet should be transferred after the last data is delivered to the memory. The write with acknowledge (WA) field in the packet header shown in Figure 6.19 is used in such a situation. When the WA field is enabled, the destination NI returns an acknowledge packet to inform the source NI that the packet has arrived at the destination. The source NI holds the packet transmission of the source PU when it transmits a packet with WA and resumes transmission when it receives the acknowledge packet. The conventional bus protocols such as AXI and OCP-IP support WA, which is also called a *posted write transaction*.

6.4.5.3 Burst Packet Transfer

To support bulk data transfer for multimedia signal processing applications, burst read and write packet transfer modes must be supported. In a burst read mode, multiple sequential data are accessed using a single read address as a base address and incrementing predefined offset, which is typically 4. In a burst write mode, multiple data are written in series with a single reference address and the predefined increment offset. For the burst mode packet transfer, an issue is that the flow of the series data must not be interrupted by other packet flows. An interrupt means violation of the burst mode protocol.

In this work, burst packet flow is protected from interruption by holding arbiter operations. The HA header fields (see Figure 6.19) of every packet, except the last

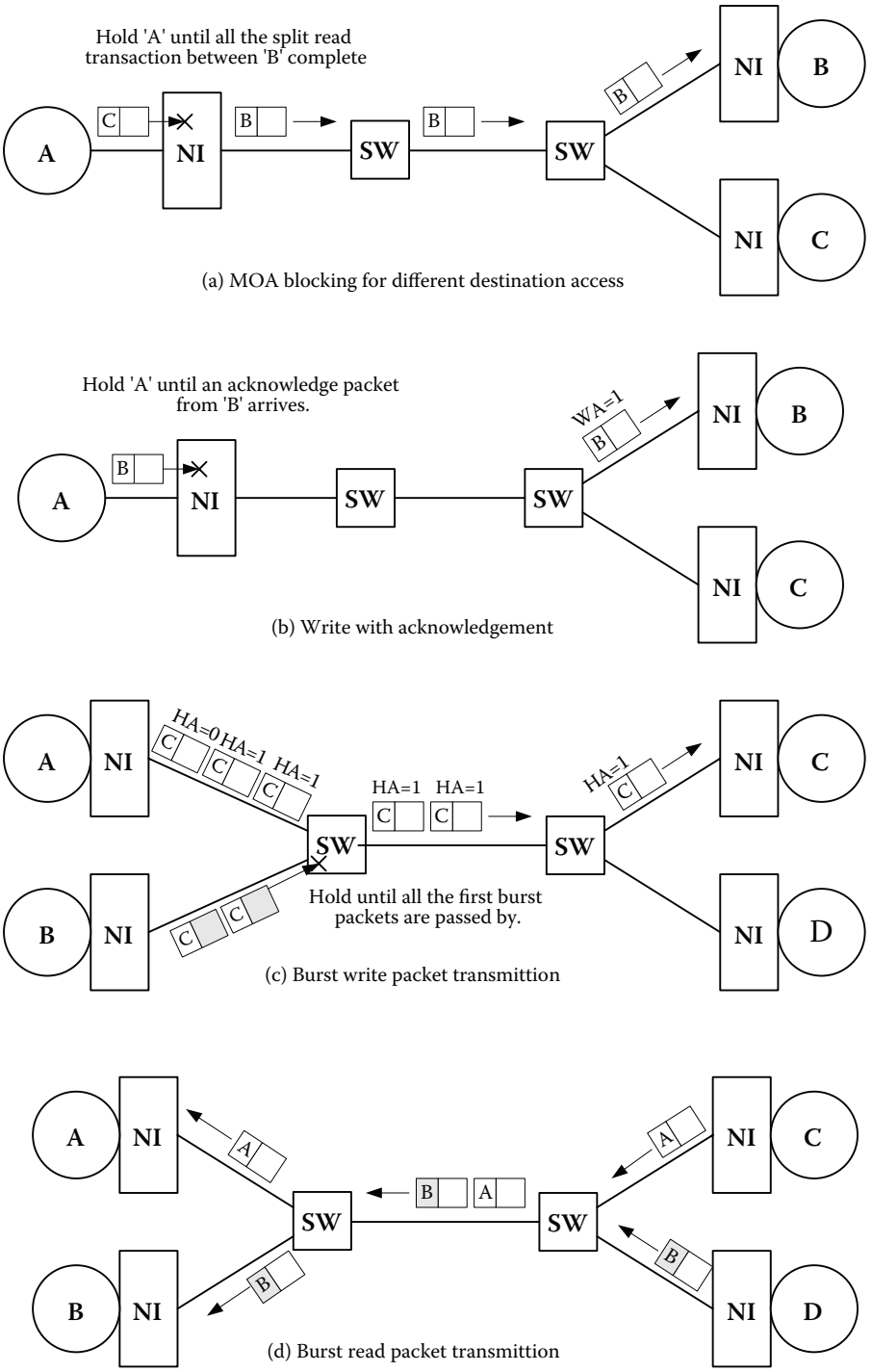


FIGURE 6.16 Protocols for four different end-to-end services.

packet of the burst packet flow, are enabled. When the HA field is enabled, the arbiter of a switch holds its grant output; thus the output port assigned to the burst packet flow are not switched to other inputs until the arrival of the last packet.

In a specific situation, a burst read flow does not need to be protected. If PUs are categorized into master and slave groups, the burst read flow occurs only from slaves to masters. Besides, a master is expected to receive packets that it has initiated. In other words, when a master issues a burst read packet to a destination, packets come only from the destination. This implies that even if the burst packet flow from a slave to a master is interrupted by other flows, a master will receive burst data sequentially.

6.4.5.4 Enhanced Burst Packet Transfer

When bulk packets are transferred from a source to a destination, for instance, in a burst packet transaction, repeated processes such as synchronization, queuing, and arbitration in all the intermediate switches are redundant and inefficient. The packet bypassing technique (PBT), described in this part, is devised to solve this problem.

The concept is as follows: Switches are divided into two groups, level 1 and level 2. When the bypass mode is enabled, packets bypass the level-1 switches; thus, effective switching hop count along the end-to-end route is reduced, and the advantages of a packet-switched network are still provided by the level-2 switches. To implement this feature, level-1 switches are designed with bypass paths for synchronizers and FIFOs.

Figure 6.17 illustrates the PBT protocol. Transmitting packets with bypass mode follows this sequence: (1) A transmitter enables a specific field of a packet, called *bypass-enable* (BE) field, and outputs the packet. The packet is called *first bypass-mode packet* (FBP). Meanwhile, the transmitter resets its credit counter and thus stops the next packet transmission. (2) If a level-1 switch detects the FBP, it asserts a BE signal to skip the synchronization and queuing processes for the next packets. (3) If a level-2 switch receives the FBP, it continues to route the FBP to the next switch,

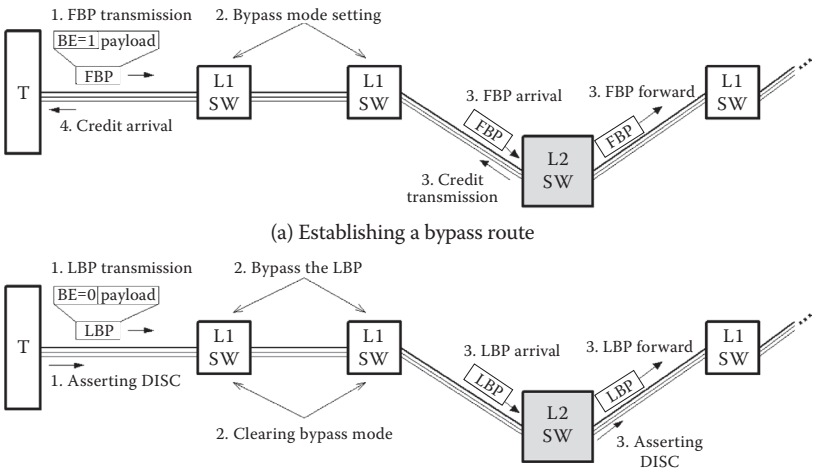


FIGURE 6.17 Packet bypassing protocol.

and sends credits to the transmitter, as many as the empty queue space would allow. (4) If the transmitter receives a credit, it resumes transmitting packets, which will be transferred with the bypass mode. (When it transmits packets, it enables the BE fields of the packets.) Disconnecting the bypass route follows this sequence: (1) A transmitter disables the BE field of the last packet and outputs the packet, which we call *last bypass-mode packet* (LBP). Then, the transmitter asserts the disconnect signal (DISC). (2) The DISC signal clears the bypass-mode of the level-1 switches as it propagate. Meanwhile, the LBP, which propagates a little ahead of the DISC signal, bypasses the level-1 switches and arrives at a level-2 switch. (3) If a level-2 switch receives the LBP, it forwards the LBP to the next switch. And then, it asserts its DISC signal as the transmitter does.

In the implementation results of the PBT using 0.18 μm technology, the PBT is found to reduce energy consumption and delay time of a 40 b 5 × 5 switch by 70 and 85%, respectively. A disadvantage of the PBT is performance degradation due to the time required to establish a bypass route. When an FBP establishes a bypass route, it does not bypass the switches but is transferred as normal packets. Such a problem can be resolved by setting high priority for an FBP or by a technique presented in the literature cited, [15] which establishes bypass routes using the sideband arbitration network.

6.4.6 Protocol Design with Finite State Machine Model

Now that we have covered the OSI reference model and its relevant NoC layers, we will look into an example of implementation using the finite state machine (FSM). Because physical and data link layers require novel circuit schemes but do not need complex controls and the network layer has relatively low control complexity compared to the transport layer, transport layers will be the target of interest in this subsection.

The transport layer is responsible for direct interface with each IP in the application layer, and most of the abstraction (i.e., packetization/depacketization, packet reordering if necessary, and blocking/segmentation) occurs at this layer. Therefore, the transport layer is, in general, the most complex one among the lower four layers in the OSI reference model, and we can implement this layer using an FSM to simplify and to cope with its complex operation.

In this subsection, we will see how an FSM can be used to implement and support the features of the transport layers described in Subsection 6.4.4.1, i.e., MOA, WA, and BPT. The enhanced BPT case is left as an exercise.

Figure 6.18 depicts the FSM for the packet structure of NoC: the KAIST BONE™ 3.0. The FSM is a simplified version to improve readability. Each arrow stands for a command or operation outside NIs, whereas each circle represents an operation within NIs.

To support MOA, transceiver and receiver FSMs are decoupled at the NI; even if there is no response for the request packet, another request packet can be sent. Also, for burst mode operation, the burst and single modes are separately implemented at the transceiver. In addition, acknowledgment can be used in packet or flit format, and the designer is allowed to trade off between overhead and reliability.

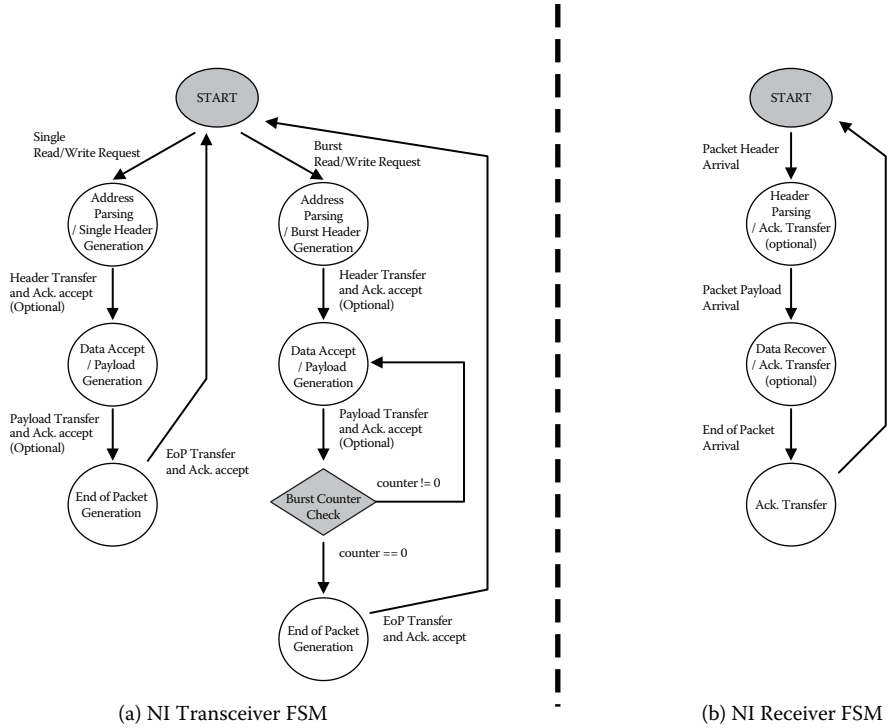


FIGURE 6.18 FSM for the packet structure of a NoC: KAIST BONE™ 3.0.

6.4.7 Packet Design for NoC

Figure 6.19 shows the packet structure of a NoC, the KAIST BONE™ [13]. The foremost interest in composing a packet in NoC is the simple hardware implementation. Because packet generation and parsing occur on chip, and as the on-chip environment strictly limits the power and area, a simple but powerful hardware implementation is the key. The total length of a packet is 88 bit, which is designed to fulfill the requirements of NoC; as described earlier, too long a packet makes it too costly to deal with error control, and too short a packet increases segmentation and reassembly overheads.

The packet consists of an NI header, an SW header, and three payloads. The NI and SW headers are used for end-to-end NIs and switches, respectively. Payload 1

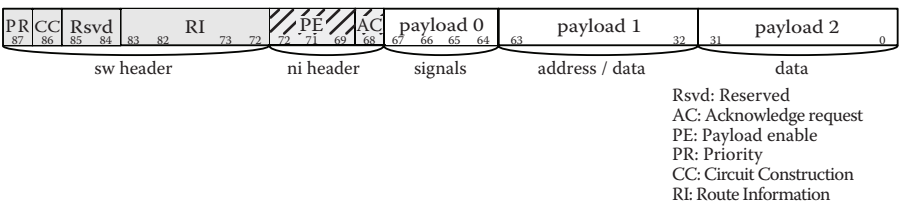


FIGURE 6.19 Packet structure of a NoC: KAIST BONE™ 3.0.

and payload 2 carry 32 b addresses and data. Payload 0 is an extra field for transmitting control signals such as write-enable and chip-select.

Unlike in data networks, the switches in KAIST BONE™ 3.0 do not have routing tables to route a packet. Instead, it exploits source routing; the packet itself has routing information, as the RI field shown in Figure 6.19. Without any calculation, a switch routes a packet through proper links by just parsing the header. In addition, without any routing table, we can reduce the amount of memory used in each switch. Because memory tends to occupy large area and consume much power, source routing reduces power and area overhead, which is crucial in NoC applications. Detailed descriptions of each field of the packet will be given in Chapter 8.

Whether packet length is fixed or variable affects the complexity of packet parsing logic and serialization methods. Two different serialization methods depending on the packet length flexibility are shown in Figure 6.20a and 6.20b: field-based serialization (FBS) and packet-based serialization (PBS). In FBS, which can be applied to fixed-length packet serialization, dedicated link wires are assigned to each packet field. The advantages of this scheme are as follows: (a) The packet parsing procedure is very simple, and (b) the bit width of a field can be easily increased with additional link wires. A disadvantage of this scheme is that its link utilization is inefficient if some fields are disabled. For example, if a packet carries payload 2 (PL2) as it is empty, the link <21:14> remains idle but other packet transactions cannot use the idle link wires. The PBS, on the contrary, can utilize the channel resource efficiently. Even though a short packet is transferred, all the link wires are used, and transmission time of the packet is shorter compared to that of long packets. The disadvantages of this scheme are the complex packet parsing procedure and inflexible bit width adjustment.

In the implementation results of the NoC [8], which uses PBS, the parsing overhead of PBS is found in energy consumption of a deserialization unit (DES) at a destination. When a 1:4 DES is designed with the PBS scheme, control logic and datapath energy consumptions are increased by ×3 and ×1.3, respectively, compared to those of FBS. As a result, the DES energy consumption is increased by about 50%. Considering the portion of the DES energy consumption in overall network, the PBS overhead, which is a trade-off with link utilization enhancement, is not negligible [15].

Now we calculate the theoretical maximum frame size for a given capacity of links, overhead length, and total message length.

Let us assume that the total length of a message to be sent is M, an overhead portion of a packet (that is composed of a header and a footer) is V, and maximum packet length is L_{max}, as shown in Figure 6.21. If we assume that the message is divided into many L_{max}-bit long packets, the total number of packets will be $[M/L_{max}]$, where $[x]$ denotes the smallest integer greater than or equal to x [17]. The first $[M/L_{max} - 1]$ packets will be L_{max}-bit long, whereas the length of the last packet will be between 1 and L_{max} bits. As a result, the total length of the packets will be as follows:

$$\text{Total bits} = M + \left\lceil M/L_{max} \right\rceil V \text{ (bits)} \tag{6.9}$$

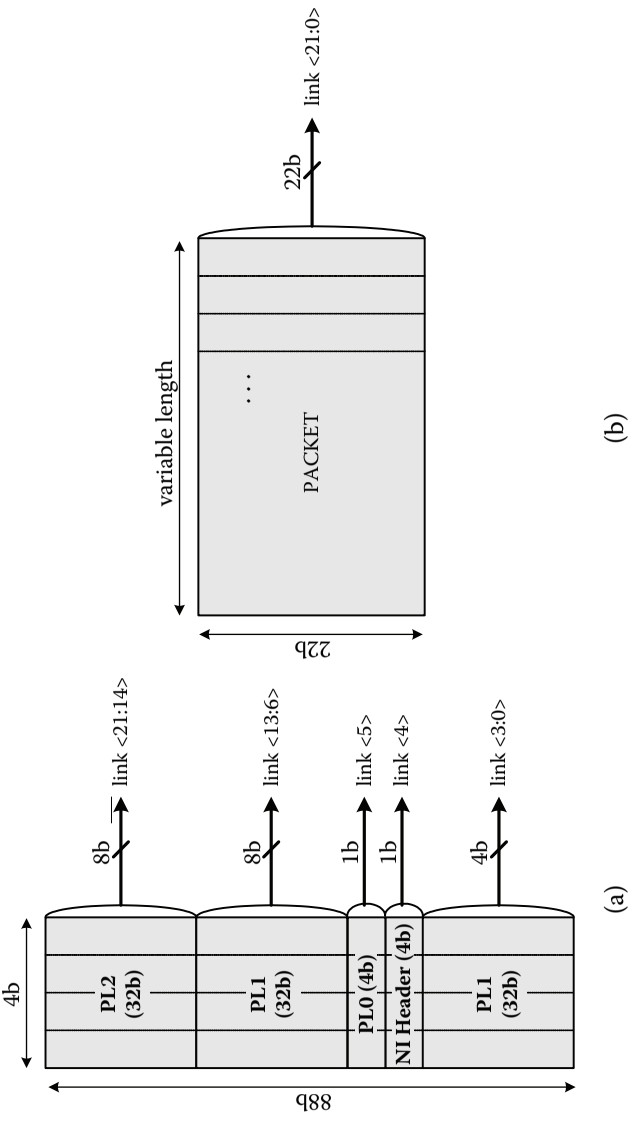


FIGURE 6.20 (a) Field-based and (b) packet-based serializations.

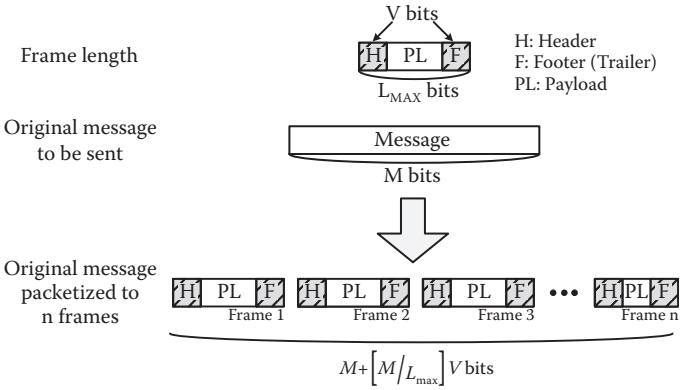


FIGURE 6.21 Framing.

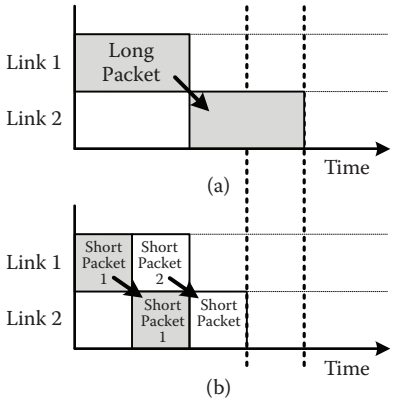


FIGURE 6.22 Pipelining: (a) long packet case and (b) short packet case.

In Equation 6.9, the second term $[M/L_{max}]V$ is the total overhead of the message, and from the relation we can conclude that the shorter the L_{max} is, the larger the number of packets, and, therefore, the bigger the overhead portion.

Now, suppose each packet must be completely received over one link before any transmission over the next links is started. In this case, if the packet length is too long, as shown in Figure 6.22a, because the long packet occupies the link, packet transmission will be delayed longer than in the case of the short packet length, as shown in Figure 6.22b—this is called *pipelining*.

If there are j links available, T is the total time required for transmitting a message to a destination, and C is the capacity of each link in bits per second, then TC is the number of bit transmission times required for message delivery, which can be represented as follows:

$$TC = \left(L_{\max} + V\right)\left(j-1\right) + M + \left\lceil M \Big/ L_{\max} \right\rceil V \tag{6.10}$$

The next step is to take the expectation $E\{TC\}$. Under the assumption that M is uniformly distributed over spans of L_{max} bits, we can approximate as follows [17]:

$$E\left\{\left\lceil M \Big/ L_{\max} \right\rceil\right\} = E\left\{M \Big/ L_{\max}\right\} + \frac{1}{2}$$

Therefore,

$$E\{TC\} \approx (L_{\max} + V)(j-1) + E\{M\} + E\{M\}V / L_{\max} + V/2 \tag{6.11}$$

From Equation 6.11, we can calculate the $L_{\max}$ that minimizes $E\{TC\}$ by taking the first derivative. The result is as follows:

$$L_{\max} \approx \sqrt{\frac{E\{M\}V}{j-1}} \tag{6.12}$$

Equation 6.12 gives insights into trade-off between overhead and pipelining. When the overhead V increases, the packet size $L_{\max}$ increases, too. On the contrary, when the path j increases, then $L_{\max}$ decreases.

6.5 SUMMARY

We have analyzed the energy cost of flat and hierarchical topologies with various network sizes and traffic patterns. The energy cost of the point-to-point topologies is the lowest, but its area cost is too high to be implemented. According to our analysis, the hierarchical bus (or multilayer bus) cannot be used when the number of integrated cores is larger than 25. The flat-mesh topology shows better energy efficiency than the hierarchical bus only when the traffic is uniformly distributed. However, as the traffic gets localized, the energy cost of the mesh does not scale down as much as other hierarchical topologies do. The hierarchical-star (local-star global-star) topology is the most energy-efficient and scalable topology for heterogeneous systems in which traffic is localized.

You should also analyze the performance (bandwidth and latency) and area cost for the topologies. The detailed performance and cost comparisons are well addressed in the literature [11].

We have also covered the concept of layered architecture, protocol function, and packet structure, which should be considered when a NoC protocol is designed.

REFERENCES

1. Jose Duato, Sudhakar Yalamanchili, and Lionel Ni, *Interconnection Networks,* Morgan Kaufmann.
2. Murali, S. et al., *SUNMAP:* A Tool for Automatic Topology Selection and Generation for NOCs, in *Proc. Design Automation Conf.,* 2004, pp. 914–919.
3. Wang, H. et al., A Technology-aware and Energy-oriented Topology Exploration for On-chip Networks, in *Proc. Conf. on Design Automation and Test in Europe,* 2005, pp. 1238–1243.
4. Kreutz, M. et al., Energy and Latency Evaluation of NoC Topologies, in *Proc. Int. Symp. on Circuits and Systems,* 2005, pp. 5866–5869.

5. Torii, S. et al., A 600MIPS 120 mW 70 μA Leakage Triple-CPU Mobile Application Processor Chip, in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers,* 2005, pp. 136–137.
6. George, V. et al., The Design of a Low Energy FPGA, in *Proc. Int. Symp. on Low-Power Electronics and Design,* 1999, pp. 188–193.
7. Zhang, H. et al., Pleiades, Berkeley.
8. Kangmin Lee et al., A 51 mW 1.6 GHz On-Chip Network for Low-Power Heterogeneous SoC Platform, in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers,* 2004, pp. 152–153.
9. Wang, H., et al., A Technology-aware and Energy-oriented Topology Exploration for On-chip Networks, in *Proc. Conf. on Design Automation and Test in Europe,* 2005, pp. 1238–1243.
10. AMBA™ Specification, Rev. 2.0, 1999, www.arm.com.
11. Kangmin Lee, *Low-Power Network-on-Chip for High-Performance SoC Designs,* Doctoral Dissertation, http://ssl.kaist.ac.kr/.
12. Stallings, W., *Data and Computer Communications (7th ed.)* Pearson Prentice Hall, 2004.
13. KAIST BONE™ 3.0 Specifications, Semiconductor System Laboratory, KAIST, 2004.
14. AMBATM AXI Protocol Specification, http://www.arm.com, 2003.
15. Kim, K. et al., An Arbitration Look-Ahead Scheme for Reducing End-to-end Latency in Networks on Chip, *Proc. of Int. Symp. on Circuits and Systems,* May 2005, pp. 2357–2360.
16. Se-Joong Lee, *Cost-Optimized System-on-Chip Implementation with On-Chip Network,* Doctoral Dissertation, http://ssl.kaist.ac.kr/.
17. D. Bertsekas and R. Gallager, *Data Networks (2nd ed.),* Prentice-Hall International Editions, 1992, pp. 93–97.
18. Mckeown, N., The iSLIP scheduling algorithm for input-queued switches, *IEEE/ACM Transactions on Networking,* Vol. 7, Issue 2, pp. 188–201, 1999.
19. Kangmin Lee, Se-joong Lee, and Hoi-Jun Yoo, A distributed crossbar switch scheduler for on-chip networks, *IEEE Proc. Custom Integrated Circuits Conf.,* pp. 671–674, Sept., 2003.

7 Low-Power Design for NoC

7.1 INTRODUCTION

The Network on Chip (NoC) paradigm is expected to be applied not only to high-end server [1] and desktop applications but also to mobile and wireless communications [2]. The NoCs for high-end applications, such as supercomputer and home entertainment server, need low-power design because of associated thermal issues, which may require expensive package and cooling equipment. On the other hand, the NoCs for mobile and wireless applications need a more aggressive low-power strategy to guarantee a reasonable operation time with a limited battery. More powerful applications, such as 3D graphics game, car navigation, and image recording, are implemented on your hand-held devices. Furthermore, as wireless communications evolves to broadband, much higher communication data rates are required. For example, 4G-terminal is expected to guarantee a 100 Mbps data rate on a moving vehicle, and 1 Gbps bandwidth at fixed position. All of those applications need more electrical energy, but the capacity of a battery is too small to cope. Thus, low-power design, not only for the high-end applications but also for mobile wireless applications, is mandatory.

In this chapter, we are going to analyze and propose multiple low-power design techniques, especially for the NoC. We have analyzed the low-power network topology in the previous chapter. We will cover a signal and circuit level, and also a protocol and architectural level in this chapter.

7.2 LOW-POWER SIGNALING

The dynamic power consumption of the on-chip interconnection can be simply described as

$$P_{interconnect} = \alpha \times C_w \times V_{swing} \times V_{driver} \times f \tag{7.1}$$

where α is the switching probability; C_w, the wire capacitance; V_{swing}, the voltage swing on the wire; V_{driver}, the supply voltage of the driver; and f, the signaling frequency [3]. In this section, we will give a brief design guideline for low-power interconnection by minimizing each of α, C_{wire}, V_{swing}, and f. Figure 7.1 shows a basic model for the signaling on an interconnect.

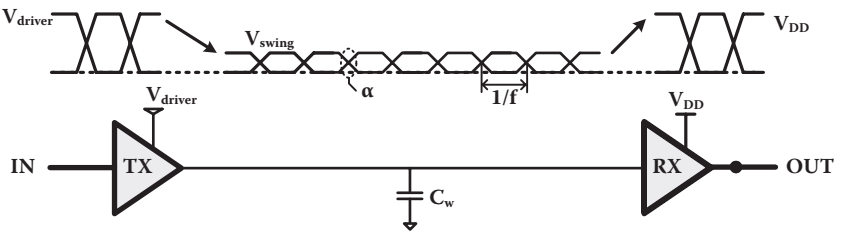


FIGURE 7.1 Interconnection model.

7.2.1 Channel Coding to Reduce the Switching Probability—α

A lot of research has been conducted on low-power bus coding to reduce the switching probability, resulting in techniques such as bus-invert (BI) coding [4], gray-code [5], T0-code [6], partial bus-invert coding [7], probability-based mapping [8], and so on. However, a report has criticized the effectiveness of those on-chip bus coding techniques because the power dissipation overhead on the (de)coder is comparable to the power saving obtained by the coding when the wire length is not longer than a few tens of millimeters [9]. Furthermore, those bus-coding schemes are effective in parallel buses and sequential data patterns. However, they are ineffective in the multiplexed channel used on packet-switched networks because the data packets are packed and not sequential.

Because the link wires connecting processing units and switches are heavily used, wiring congestion will become one of the major challenges in NoC design. To alleviate wiring congestion, a narrow channel [10] or on-chip serialized channel [11,12] is proposed. However, the switching probability on the serial link gets much higher than a parallel link, as illustrated in Figure 7.2. To solve the problem, a simple but effective transition-reducing encoding (SILENT) was reported for a serial link [13]. In this coding, the transition on parallel wires is encoded as symbol "1" before serialization, as depicted in the Figure 7.3. By using the data correlation between successive data words, the probability of 1's occurrences on the serial wire decreases, so the switching probability is also reduced. This encoding technique is more effective when the data have more correlation with multimedia applications. The details of the serial interconnection and its channel coding will be discussed further in the following chapter.

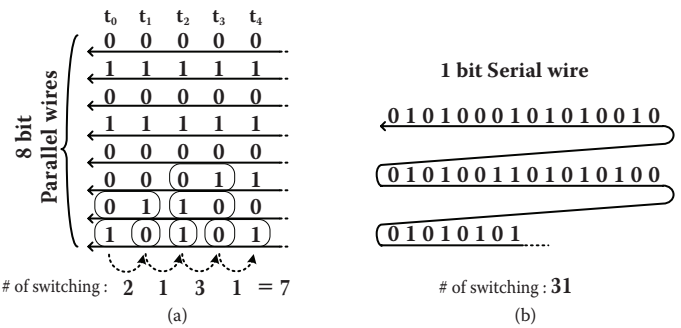


FIGURE 7.2 Signal transitions on (a) a parallel and (b) a serial link.

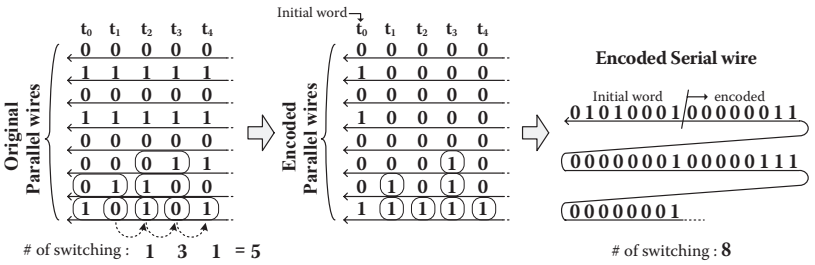


FIGURE 7.3 SILENT coding.

7.2.2 Wire Capacitance Reducing Techniques

Wire capacitance—coupling capacitance to adjacent wires—can be reduced simply by spacing the wires wider and using upper metal layers rather than lower layers. This approach needs more wiring resources horizontally and vertically. To minimize the wiring area, a serialized wire [11,12] can be used. This serialization technique will be discussed in Section 7.3 in more detail.

Another approach is to use low-k dielectric material between metal layers. Research is being conducted on it, but the manufacturing cost is still high. We will not give a detailed explanation of it, as it is out of the SoC design issue.

7.2.3 Low-Swing Signaling

Lowering the swing and driving voltage is the most effective way to reduce the power dissipation on interconnections. However, the swing voltage and receiver circuits should be carefully designed against the noise sensitivity and speed degradation.

7.2.3.1 Driver Circuits

There is a limited degree of freedom in the design of drivers. If a reduced supply voltage, V_{ddl}, is used, the power dissipation is reduced to $(V_{ddl}/V_{dd})^2$ [Figure 7.4a and Figure 7.4b]. V_{ddl} can be provided from an off-chip or generated on a chip by DC–DC conversion from the main supply, V_{dd}. Unless the additional power supply is applied, the low-swing can be obtained by exploiting a transistor V_{th} drop or pulse-enabled driving [Figure 7.4c and 7.4d]. However, these designs are susceptible to process variation and noise. In the pulse-controlled driver, V_{swing} is determined not only by the pulse duration but also by C_w, which is hard to estimate in design stage. The most widely used driver is the type of (b) [12,14,15]. By using an NMOS pull-up transistor instead of a PMOS transistor, faster rising time on the output wire is obtained with smaller transistor size.

7.2.3.2 Receiver Circuits

There are two design options in receivers with regard to the noise immunity: a single-ended level-converter [16,17,18] or differential amplifier [12,14,19,20]. Differential signaling is more immune to noise because of its high common-mode rejection,

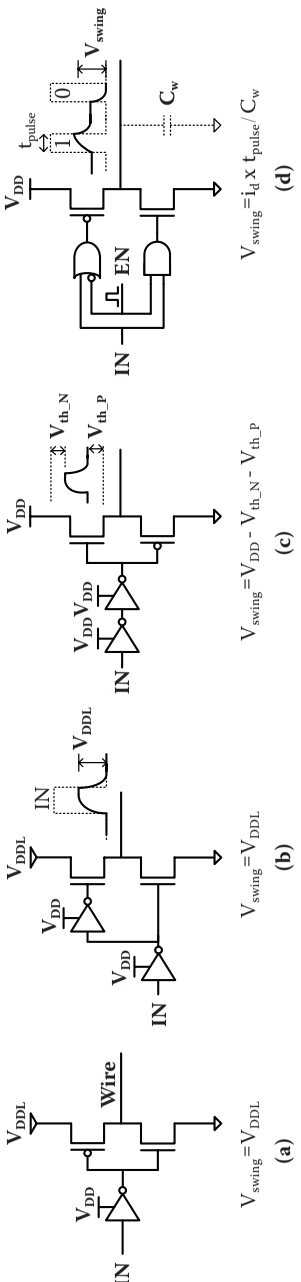


FIGURE 7.4 Low-swing drivers.

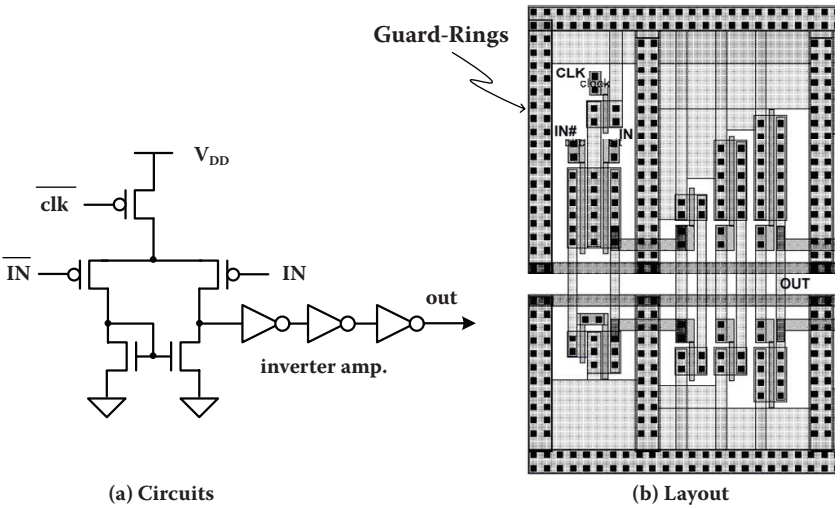


FIGURE 7.5 Clocked sense amplifier.

allowing a further reduction in the swing voltage [14,15]. Although differential signaling requires double wires, the wiring congestion can be alleviated by using on-chip serial links [21]. Zhang evaluated various receivers, including a pseudodifferential amplifier in respect of energy, delay, swing, and SNR [15].

In an on-chip interconnection network, the receiver circuits should be lightweight and occupy as small an area as possible because it is used abundantly in most of the network interfaces. Figure 7.5 shows an example of a simple clocked sense-amplifier with a three-stage CMOS inverter chain. PMOS transistors are used as input gates to receive a low common-mode input signal. The sizes of the input gates and their bias currents are chosen to amplify the desired low-swing differential input (W_P/L_P = 3 μm/0.18 μm, V_{swing} = 0.2 V, V_{DD} = 1.6 V, area = 10 × 15 μm^2 [12]).

7.2.3.3 Static and Dynamic Wires

There are two kinds of wires: static and dynamic wires. To speed up the response of wires, it is precharged to V_{DDL} through PMOS transistors before each bit transition. After the precharging phase, the wire is conditionally discharged by the pull-down transistor of a driver. This dynamic signaling is used for multidrop buses having large fan-in and fan-out. In NoC, however, the link is point-to-point, so that there is only a single fan-in and fan-out. Therefore, the dynamic wire is not a good candidate for on-chip networks, especially when the wire has long latency. Furthermore, it is susceptible to noise.

7.2.3.4 Optimal Voltage Swing

There was a report on the existence of an optimum voltage swing on long wire signaling for the lowest energy dissipation [22]. To find the optimum voltage swing, the interconnection system is simulated with extracted wire capacitance value from the real layout. Figure 7.6a shows an example of the energy consumption per bit transmission via the low-swing interconnections [12]. The required energy on the driver

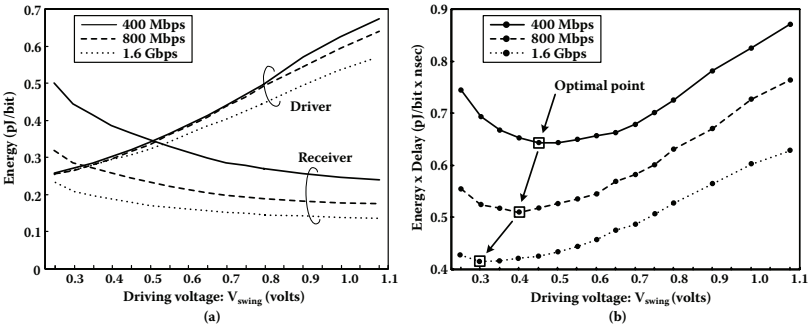


FIGURE 7.6 Energy consumption on a low-swing driver and receiver.

to create a certain voltage swing on the wires decreases linearly with the swing level. On the other hand, the energy to amplify this signal back to its normal logical swing level increases superlinearly with the decrease of V_{swing} level. The optimum V_{swing} exists because of such opposite trends of the required energy in the transmitter and the receiver. Figure 7.6b shows energy and delay product versus V_{swing}. The optimal swing voltage is not unique but varies with the signal rates, as demonstrated in the figure. As you can see, as the swing voltage decreases further below the optimal point, the energy and delay products become worse. Therefore, you should determine the swing voltage carefully in your design.

7.2.3.5 Frequency and Voltage Scaling

K. Lee presented a three-stepped frequency/voltage scaling based on the network workload [12], and F. Worm introduced a self-calibrating circuit that controls the voltage swing on the link [23] so that dynamic power consumption is minimal for the required data rate.

You can also imagine a multilevel signaling in which a single symbol represents more than binary values. Intermediate levels exist between one and zero. Therefore, you can reduce the signaling frequency to a half or a quarter. This signaling technique is currently used in advanced external memory interfaces.

7.3 ON-CHIP SERIALIZATION

An on-chip serialization (OCS) technique [21] reduces the number of interconnect wires and a switch size, also. The reduction of wires and switch size affects the overall area and energy consumption of a network. In this section, the areas and energy consumptions of the three building blocks, i.e., link, queuing buffer, and switch, are analyzed in regard to serialization ratio* (R_{SER}). After that, areas and energy consumptions of Mesh and Star networks, according to the R_{SER}, are analyzed, and an optimal R_{SER} is found. Impacts of the OCS on packet transmission latency are described with waveforms. All the analyses in this section are based on the parameters and symbols shown in Table 7.1.

* RSER is defined as (I/O bitwidth of PUs)/phit size.

TABLE 7.1
Process and Design Parameters of 0.18 μm CMOS Technology

Category	Description	Typical Value	Symbol
Design parameters	Packet size (= default phit size)	80 b	
Area (μm^2)	3-packet queuing buffer	3.30×10^4	A_{QB}
	80 b 1 × 1 crosspoint switch fabric	7.74×10^3	A_{SF}
	1-input MUX-tree [24] arbiter	8.58×10^2	A_{ARB}
	80 b 1 mm metal routing	8.80×10^4	A_{LK}
Energy (J)	1-packet write and read	2.67×10^{-11}	E_{QB}
	80 b 1 × 1 crosspoint switch fabric	9.06×10^{-12}	E_{SF}
	1-input MUX-tree arbiter	4.00×10^{-13}	E_{ARB}
	80 b 1 mm metal routing	4.78×10^{-11}	E_{LK}

7.3.1 Area and Energy-Consumption Variation Due to the OCS

Energy consumption of a link is determined by the wire capacitance and driving buffer size. When the OCS is applied, the number of wires of a link is reduced, so that the wires can be placed with larger space within the allowed routing area, which reduces coupling capacitance, and thus energy consumption. On the contrary, the OCS increases operation frequency of the link, so the driver size must be increased to support the high-speed signaling.

With the wire spaces and driver sizing factors determined in Table 7.2, normalized values of A_{LK} and E_{LK} vary according to R_{SER}, as shown in Table 7.3. It shows that E_{LK} decreases until 4:1 serialization, and it increases for higher values of R_{SER}, i.e., 8, which is due to high-frequency signaling overhead in the driving buffers.

When the OCS is applied, input and output ports of a queuing buffer need to be modified to interface the reduced bit width. Also, the modification requires addi-

TABLE 7.2
Design Parameters for On-Chip Serialization

Category	Description	Typical Value				Symbol
Switch fabric	Coupling capacitance of a SF wire (80 b 1 × 1 switch)	24.3fF				C_W
	Junction capacitance of a connector (80 b 1 × 1 switch)	1.19fF				C_J
	Gate capacitance of SF connector	1.22fF				C_C
	Port bitwidth	80 b				W
Link wire	R_{SER}	1	2	4	8	
	Driving buffer tapering factor (energy-optimized based on SPICE simulation)	16	16	8	4	
	Routing wire space (μm)	0.5	1.0	1.5	2.0	

TABLE 7.3
Area and Energy-Consumption Variations with R_{SER}

	Area				Energy-Consumption			
Item R_{SER}	Link	Queuing Buffer	Switch Fabric	Arbiter	Link	Queuing Buffer	Switch Fabric	Arbiter
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.75	1.00	0.25	1.00	0.75	1.13	0.54	1.00
4	0.50	1.00	0.06	1.00	0.64	1.30	0.32	1.00
8	0.31	1.00	0.02	1.00	0.80	0.20	0.20	1.00

tional circuitry, which increases E_{QB}. E_{QB} variation with R_{SER} is also summarized in Table 7.3. Because the area of storage cells dominates the queuing buffer area, the variation of A_{QB} is negligible.

In a switch, the arbiters are independent of the OCS, and only the switch fabric is affected by the OCS. Switch fabric area is reduced by the factor R_{SER}^2. As the switch fabric size is reduced, its energy consumption is also reduced. On basis of the energy model presented in Reference 25 and parameters defined in Table 7.2, the energy required to transfer a single packet, E_{SF}, is expressed as

$$E_{SF} = E_{SFW} + E_{SFC}$$

$$E_{SFW} = \text{No. of phits per packet} \times \text{phit bitwidth} \times \text{wire cap.} \times V^2$$

$$= R_{SER} \times \frac{W}{R_{SER}} \times \left(\frac{C_W}{R_{SER}} + C_J \right) \times V^2$$

$$E_{SFS} = \text{port bitwidth} \times \text{connector gate cap.} \times V^{2^3}$$

$$= \frac{W}{R_{SER}} \times C_C \times V^2$$

where E_{SFW} and E_{SFC} are mean energy consumption on fabric wires and connectors in a 1×1 switch, respectively. Because the C_W is much larger than C_J, as shown in Table 7.2, the preceding equations imply that E_{SF} is reduced effectively by the factor of $1/R_{SER}$.

7.3.2 Optimal Serialization Ratio

Based on the area and energy analysis shown in Section 7.3.1, the serialized networks of mesh and star topologies are analyzed. Figure 7.7 shows the energy variation of star and mesh according to R_{SER}. Both mesh and star areas are minimized when R_{SER} is 4. Further serialization is not energy-efficient because of the high-frequency operation in links and queuing buffer overhead. The areas of mesh and star decrease continuously, as shown in Figure 7.8. However, when R_{SER} exceeds 4, its effect is negligible. Therefore, in this example, a 4:1 serialization is the optimum selection.

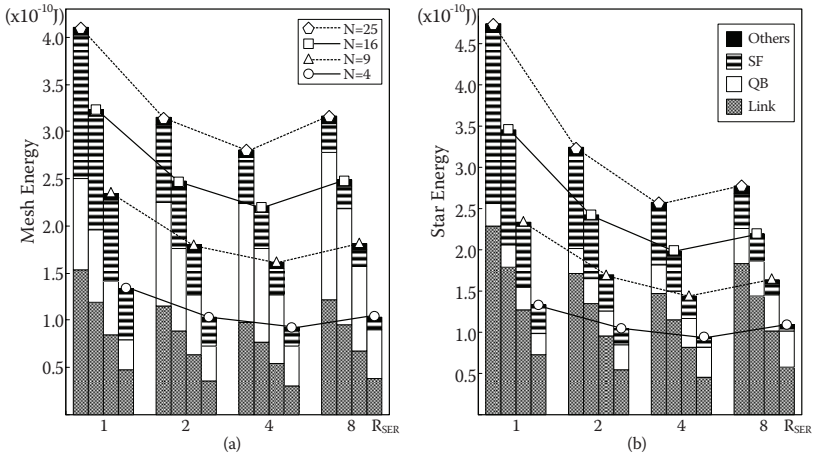


FIGURE 7.7 Energy variation of mesh and star according to R_{SER}.

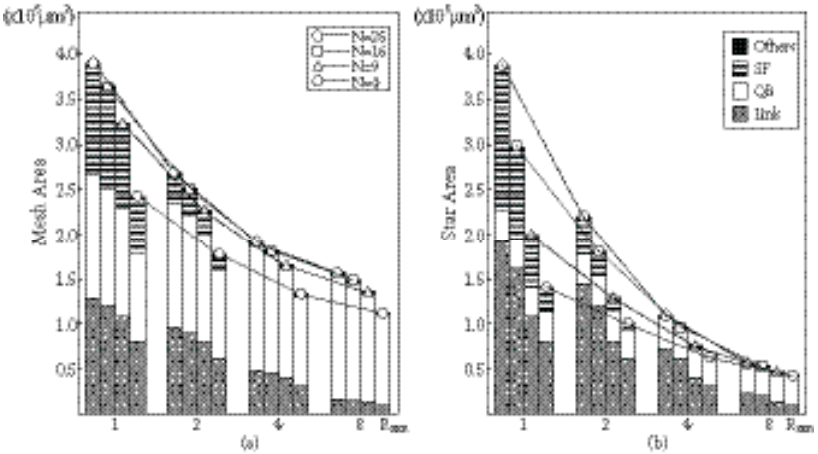


FIGURE 7.8 Area variation of mesh and star according to R_{SER}.

This implies that the abundance of wire line resources in on-chip situation does not mean allowing the use of wide channels, but that the wires must be organized by an on-chip serialization technique to optimize network area and energy consumption.

7.4 LOW-POWER CLOCKING

7.4.1 Clock Distribution inside the NoC

As the global synchronization of an SoC becomes challenging, globally asynchronous systems are becoming a practical solution. Building a globally asynchronous system, however, moves the global synchronization effort from system design to

network design. The NoC covers the entire chip area, and the issue of the global synchronization of the network is the same as that of the SoC.

Without global synchronization, a NoC can adopt one of three communication styles: mesochronous, plesiochronous, and asynchronous communication. Applying asynchronous signaling to the global interconnection or on-chip network system has been proposed in the literature [26, 27, 28]. In asynchronous communication, data is transferred from a transmitter to a receiver based on a certain handshaking protocol such as 2- or 4-phase signaling. In global wires, however, the handshaking protocol suffers from considerable performance degradation because it requires a signal to make a round trip for every signal transaction. Using the asynchronous pipeline [26], the long link can be pipelined, and throughput can be enhanced. However, it is incompatible with high-speed wave-pipelining.

In plesiochronous signaling, each network device is fed by its own clock source, and each clock source is assumed to have the same clock frequency, but not exactly the same. Plesiochronous signaling is used in large-scale networks (compared to an on-chip network), where sharing a common clock source between a transmitter and a receiver is not practical. Because the clock frequencies among network devices are not exactly the same, the receiver should have a mechanism to resolve the variable phase margin. In on-chip networks, however, plesiochronous signaling seems to be too extreme a clocking condition, because sharing a common clock source is not a big problem and does not incur much cost, whereas the effort to resolve the undetermined phase and clock frequency difference is considerable.

Mesochronous communication, in which the network devices are fed by a common clock source but the clock phase—clock skew—of functional blocks may be different from one another because of asymmetric clock tree design and difference load capacitance of the leaf cell, seems to be an optimal solution for the on-chip network. If the network has regular placement such as in mesh topology, the phase difference is deterministic [29]. If the network has an irregular shape, thus making the phase differences among the distributed network devices indeterminable, synchronizers are required between the clock domains (see Figure 7.9).

7.4.2 Synchronizers

Synchronizing mesochronous input signals requires a synchronizer. Several kinds of synchronizers, including the first in first out (FIFO) synchronizer, delay-line synchronizer, and simple pipeline synchronizer, have been proposed [30]. If an on-chip network is designed with great care, the clock skew and signal propagation delay between the clock domains can be estimated precisely, and only a single pipeline latch can synchronize mesochronous signals. In reality, the postfabrication factors, however, generate unexpected changes on the skew and the delay, which reduces the intended phase margin.

In on-chip network, the phase difference usually has a range of finite values. In this case, a simple pipeline synchronizer can be used to solve the unknown phase problem. There are three situations in which the values of the fixed phase difference can vary. The first is the variation in the physical distance due to irregular routing of the physical paths using multiplexers; the second is the variation in electrical

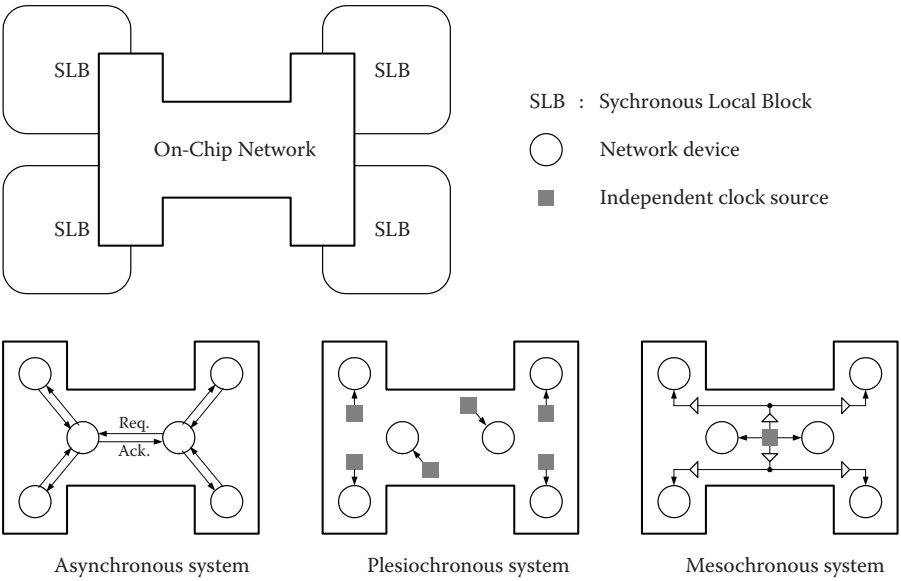


FIGURE 7.9 Various clocking schemes in the NoC.

distance due to supply voltage fluctuation; and the third is the reference timing variation due to clock frequency variation (see Figure 7.10).

A receiver can have more than one transmitter using a multiplexer. Then, the phase difference of the incoming signals at the receiver depends on which transmitter is sending these signals. Supply voltage control is used for power management in low-power application SoCs. In case of supply voltage variation, electrical distances between blocks vary, so the phase differences can be changed. The network clock frequency variation also changes the phase difference of clock signals. The variation of the clock signal is a useful technique to manage the overall power consumption

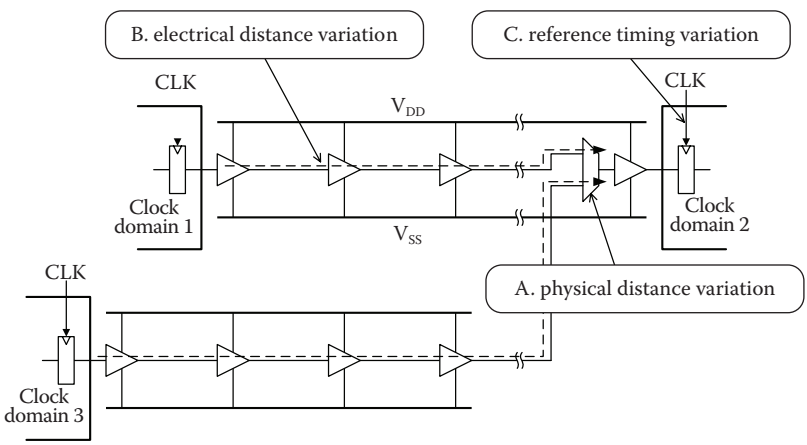


FIGURE 7.10 Timing variations on a clock signal.

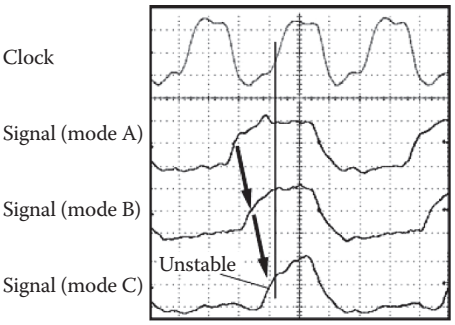


FIGURE 7.11 Measured waveforms: quantized phase variations.

in accordance with the requirements of network bandwidth. For example, K. Lee adopts 3-step network clock frequency management schemes for low-power applications [24]. When the clock frequency is changed, the reference timing is changed accordingly, which shifts the phase, too.

The variation in values of phase difference is determined by parameters such as clock frequency setting and packet path configuration; therefore, the variation is limited to several cases, and it can be quantized. The measured waveforms in Figure 7.11 show a signal's quantized phase variation in relation to the network configuration. A receiver fetches the input signal at the clock's positive edge, and a transmitter deasserts the signal if the receiver fetches it successfully. As the network configuration or mode changes, the input signal's rising edge also changes. Eventually, this rising edge lies near the rising edge of the clock in network mode C, causing an unstable synchronization. To deal with the unknown but quantized phase differences, a programmable delay synchronizer is used in Reference 31.

7.5 LOW-POWER CHANNEL CODING

7.5.1 SILENT Coding

Many parallel bus coding methods have been proposed to reduce the switching power on the address or data bus between a processor and memories [8–10]. However, such conventional parallel bus coding methods cannot be employed in the serial bus. Therefore, we proposed a serialized low-energy transmission (SILENT) coding technique to minimize the transmission energy on the serial wire [13]. We first introduce the terminology and notation that will be used throughout this section.

b$^{(t)}$[n-1:0]: The *n*-bit data word from a sender at time *t*
B$^{(t)}$[n-1:0]: The *n*-bit encoded data word at time *t*
S$^{(t)}$: Serialized data at time *t*
E: Encoding enable signal

In this regard, we ignore the latency of an encoder and a serializer for convenience. The proposed encoder Ö works as follows:

$$\mathbf{B}^{(t)}[i] = \mathbf{b}^{(t)}[i] \text{ XOR } \mathbf{b}^{(t-1)}[i] \text{ for } i = 0 \sim n-1 \quad (1)$$

The encoded words, $\mathbf{B}^{(t)}$, is equivalent to the displacement or the difference between successive data words. These encoded data words are serialized as follows:

$$\mathbf{S}^{(t)} = \mathbf{B}^{(t)}[n-1]$$

$$\mathbf{S}^{(t+1/n)} = \mathbf{B}^{(t)}[n-2]$$

$$\mathbf{S}^{(t+i/n)} = \mathbf{B}^{(t)}[n-1-i]$$

$$\cdots$$

$$\mathbf{S}^{(t+(n-1)/n} = \mathbf{B}^{(t)}[0]$$

$$\mathbf{S}^{(t+1)} = \mathbf{B}^{(t+1)}[n-1] \text{ (the next word)}$$

By serializing the encoded data words, the probability that more zeros appear on the wire increases because of the correlation between the successive data words, $\mathbf{b}^{(t)}$. Figure 7.12 shows an example of the advantage of this coding method. After encoding these data words, all bits from **B**[7] to **B**[3] become zeros because those bits do not change with time. On serializing these encoded words, as shown in Figure 7.12d, the serial wire experiences fewer transitions. So, the wire looks as though it is keeping *silent*. A conventional serial wire without SILENT coding, shown in Figure 7.12c, has three times more transitions in this example from t + 1 to t + 4. By

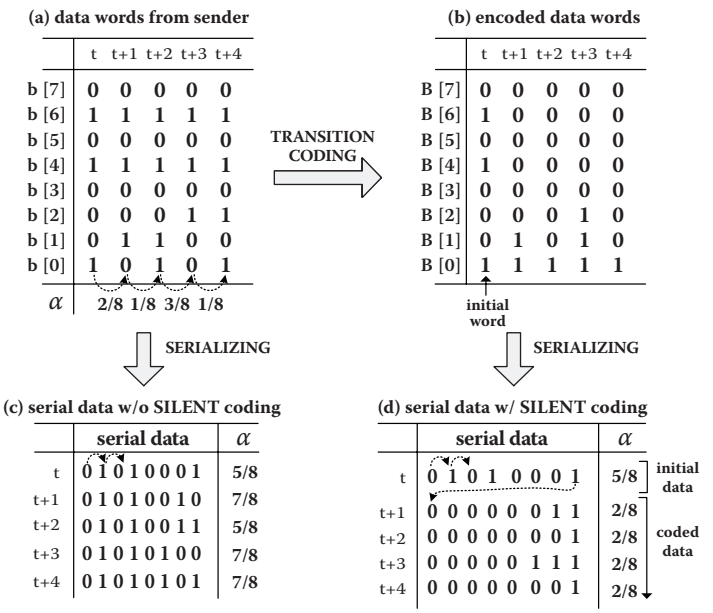


FIGURE 7.12 SILENT coding method.

reducing the number of transitions on the serial wire, the transmission energy can be proportionally saved.

The notation for deserialized data words and decoded data words is as follows:

$\mathbf{D}^{(t)}$[n−1:0]: The n-bit data word deserialized from the serial data $\mathbf{S}$ at time t
$\mathbf{d}^{(t)}$[n−1:0]: The n-bit decoded data word at time t

By assuming for convenience that there is no bit-error on the transmission channel and the latency is zero, $\mathbf{D}^{(t)}$ is identical to $\mathbf{B}^{(t)}$.

The decoder works as follows:

$$\mathbf{d}^{(t)}[i] = \mathbf{D}^{(t)}[i] \text{ XOR } \mathbf{d}^{(t-1)}[i] \text{ for } i = 0 \sim n-1 \tag{2}$$

The original data word from a sender unit, $\mathbf{b}^{(t)}$, can be recovered by the decoder, as the following proves:

Proof. From (2),

$$\mathbf{d}^{(t)} = \mathbf{D}^{(t)} \text{ XOR } \mathbf{d}^{(t-1)} = \mathbf{B}^{(t)} \text{ XOR } \mathbf{d}^{(t-1)}$$

Then, from (1), we have

$$\mathbf{d}^{(t)} = (\mathbf{b}^{(t)} \text{ XOR } \mathbf{b}^{(t-1)}) \text{ XOR } \mathbf{d}^{(t-1)} \tag{3}$$

Note that (3) is a recursive equation of $\mathbf{d}^{(t)}$. If $\mathbf{d}^{(t-1)}$ is replaced recursively by (3), we have

$$\begin{aligned}\mathbf{d}^{(t)} &= (\mathbf{b}^{(t)} \text{ XOR } \mathbf{b}^{(t-1)}) \text{ XOR } (\mathbf{b}^{(t-1)} \text{ XOR } \mathbf{b}^{(t-2)}) \text{ XOR } \mathbf{d}^{(t-2)} \\ &= \mathbf{b}^{(t)} \text{ XOR } (\mathbf{b}^{(t-1)} \text{ XOR } \mathbf{b}^{(t-1)}) \text{ XOR } \mathbf{b}^{(t-2)} \text{ XOR } \mathbf{d}^{(t-2)} \\ &= \mathbf{b}^{(t)} \text{ XOR } \mathbf{b}^{(t-2)} \text{ XOR } \mathbf{d}^{(t-2)}\end{aligned}$$

By replacing $\mathbf{d}^{(t-i)}$ terms recursively by (3) until $t-i$ becomes 0, the following expression is obtained for $\mathbf{d}^{(t)}$:

$$\mathbf{d}^{(t)} = \mathbf{b}^{(t)} \text{ XOR } \mathbf{b}^{(0)} \text{ XOR } \mathbf{d}^{(0)}$$

If we assume that $\mathbf{b}^{(0)} = \mathbf{d}^{(0)}$ at time (t = 0),

$$\mathbf{d}^{(t)} = \mathbf{b}^{(t)}$$

To guarantee the initial condition, $\mathbf{b}^{(0)} = \mathbf{d}^{(0)}$, the $\mathbf{b}^{(0)}$ at the encoder and $\mathbf{d}^{(0)}$ at the decoder are set as zeros. This can be simply done by hardware resetting the flip-flops in both of the encoder and decoder before starting transmission.

7.5.2 Performance Analysis of SILENT Coding

Figure 7.13 shows the circuit implementation of SILENT codec, and the bold line indicates a critical path in the circuits. The implementation of the SILENT encoder and decoder is so lightweight that the area, power, and latency overhead due to the codec are negligible.

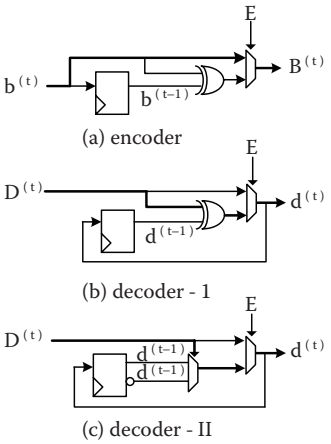


FIGURE 7.13 CODEC circuits of SILENT coding.

In the encoder, the D-flip-flop holds the previous original data, XOR generates the displacement between successive data words, and the multiplexer determines whether to enable SILENT coding or not. The area for the 1-bit encoder takes 23 × 10-μm^2 in 0.18 μm CMOS technology. The required energy for 1-bit encoding is 0.12 pJ. The power consumption for 32-bit data word encoding is about 390 μW at 100 MHz operating frequency in the *worst case* data pattern. This takes only 4% of the overall power consumption for serial communication.

There are two circuit designs of the decoder in Figure 7.13b and 7.13c. In the decoder-I, the D-flip-flop stores previous decoded data, $\mathbf{d}^{(t-1)}$. The decoder recovers the original data by XOR operation between the previous decoded data, $\mathbf{d}^{(t-1)}$, and the current encoded data, $\mathbf{D}^{(t)}$. Decoder-II uses another multiplexer instead of an XOR gate. The power consumption for 32-bit data word decoding is about 385 μW at 100 MHz operating frequency in the *worst case* data pattern.

To analyze the energy efficiency of this coding scheme, we evaluated the energy consumption with various data patterns in the communication channel, including encoders, transmitters, 8-mm serial wires with repeaters, receivers, and decoders. Figure 7.14 shows the configuration from a sender to a receiver unit.

The energy consumption in the communications depends on the data patterns to be sent. So, we evaluate the power consumption with all possible variations from a random data word. Figure 7.15 shows the comparison between the average power consumption of the serial communication with and without SILENT coding at 100 MHz operating frequency. The number of data displacements between successive 32-bit data words, $\mathbf{b}^{(t)}$, is plotted on the *x*-axis. The 0 on the *x*-axis means that $\mathbf{b}^{(t)}$

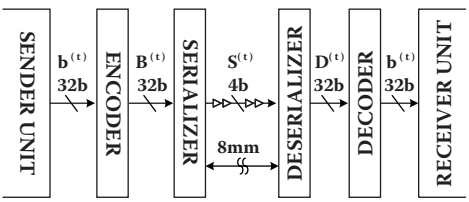


FIGURE 7.14 Configuration from a sender to a receiver unit.

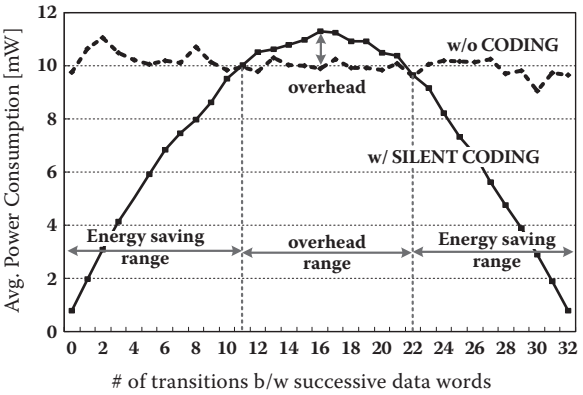


FIGURE 7.15 Power consumption in serial communications with SILENT coding.

is the same as $\mathbf{b}^{(t-1)}$, and the 16 means that on arbitrary 16 bits of 32-bit $\mathbf{b}^{(t)}$ have changed their values from $\mathbf{b}^{(t-1)}$. As a result, when the number of transitions between the successive data is less than 11, the encoded data words contain many zeros, and therefore, the serial wire has fewer transitions. Meanwhile, when the number of transitions is more than 22, the encoded data words contain many ones, and thus, the serial wire has fewer transitions. Therefore, the region below 11 or above 22 on the *x*-axis is the energy-saving region, thanks to SILENT coding. However, there is some power overhead for random data transitions of at most 14% in the region from 11 to 22. As shown here, the energy-saving range is two times wider than the overhead range, and the power saving is much larger than the overhead. Therefore, the SILENT coding will benefit most of the data traffic.

When the data pattern is sequential or highly correlated, the number of transitions between successive data words is small, as shown in Figure 7.16a, so much energy will be saved by SILENT coding. When the data pattern is quite random, the distribution of the number of transitions will resemble a Gaussian distribution with a mean value of 16, as shown in Figure 7.16b. In that case, the power overhead cannot be ignored. Therefore, we need to investigate the data pattern of a target application. In the next section, we analyze the performance of the coding method in a real application.

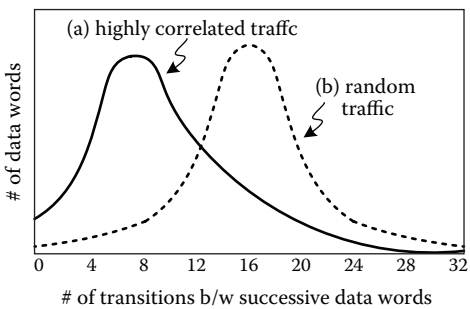


FIGURE 7.16 Distribution of signal transitions for (a) correlated traffic and (b) random traffic.

7.5.3 SILENT Coding for Multimedia Applications

To evaluate the performance of SILENT coding in a real application, we trace the on-chip traffic between a RISC processor and system memory while a 3-D graphics application is running [32]. A full 3-D graphics pipeline of geometry and rendering operations is executed for 3-D scenes with 5878 triangles. Figure 7.17 shows the distribution of the displacement of the memory address and the data for the successive memory accesses. The *instruction memory* address is so sequential that 99.5% of 6 million transactions are within SILENT's energy-saving region. The profile for the instruction address is similar to the distribution of Figure 7.17a. Although the instruction codes are quite random, their profile resembles the distribution of Figure 7.17b, with 60% being within the energy-saving region. In the case of the data memory access, 79% and 70% of 1.5 million data memory addresses and data transactions are within the energy-saving region, respectively. Those patterns are similar to the distribution of Figure 7.16a.

With this memory access pattern, we evaluated the energy consumption for serial communications in the environment, as shown in Figure 7.14. In result, Figure 7.18

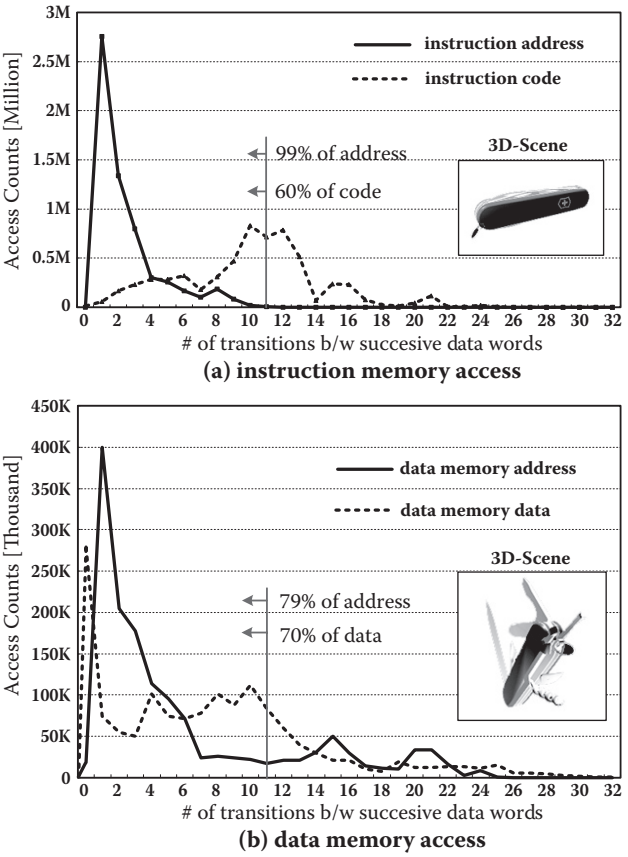


FIGURE 7.17 Distribution of signal transitions for 3-D graphics application.

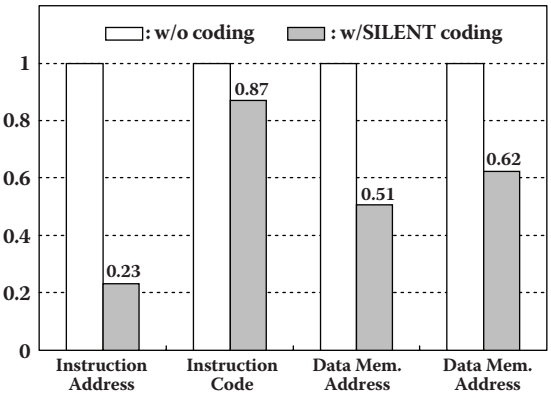


FIGURE 7.18 Normalized power consumption with and without SILENT coding.

shows the normalized average energy consumption on the serial wire with and without SILENT coding. The energy consumption with SILENT coding includes the energy dissipation in the codec circuits. SILENT coding shows the best performance for the instruction address, an energy saving of about 77%. Even with random traffic, in the case of the instruction codes, 13% energy saving is achieved. It also saves about 40% to 50% transmission energy for multimedia data traffic. In conclusion, SILENT coding reduces the energy consumption for serial communications in all kinds of on-chip data traffic in a multimedia application—3-D graphics.

7.6 LOW-POWER SWITCH

7.6.1 Low-Power Technique for Switch Fabric

Low-swing and current signaling technique can decrease the energy consumption of a crossbar because a crossbar has a large number of long and high-capacitance wires [33,34]. Another effective method is to divide the fabric into multiple segments and partially activate a selected segment [12].

7.6.1.1 Crossbar Partial Activation Technique

A conventional X–Y-based crossbar fabric is shown in Figure 7.19a. An $n \times n$ crossbar fabric comprises n^2 crossing junctions that contain NMOS pass-transistors. An NMOS-only pass-transistor is used rather than a CMOS transmission gate to reduce the voltage swing to $V_{DD}-V_{th}$ and also to reduce gate loading. In the conventional crossbar fabric, however, each input driver wastes power to charge and discharge two long wires—Row-Bar (RB) and Column-Bar (CB)—and $2n$ transistor–junction–capacitors. Therefore, the loading on the driver output becomes significant as the number of ports increases. The RB and CB should be laid out in lower metal layers, M1 or M2, to reduce the fabric area and to minimize the resistance of via.

Figure 7.19b shows a proposed crossbar switch with Crossbar Partial Activation Technique (CPAT). By splitting the $n \times n$ fabric into 4×4 fabrics (or tiles), the

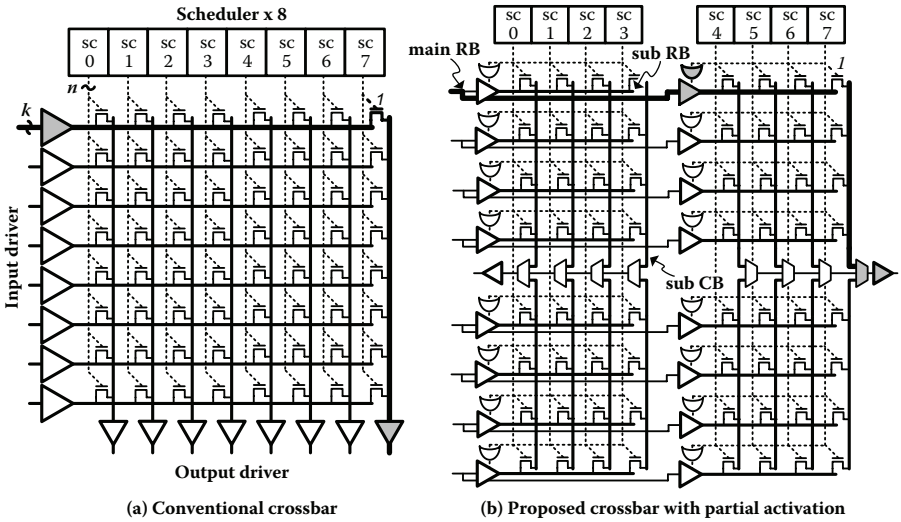


FIGURE 7.19 Schematic diagram of (a) an 8 × 8 conventional crossbar and (b) a proposed crossbar with partial activation technique.

capacitive loading activated is reduce by half. A gated input driver at each tile activates its sub-RB only when its tile gets a grant from the scheduler. Only 4 four-input OR-gates are needed additionally in each tile regardless of the depth of a channel, *k*. The output line, CB, is also divided into two sub-CBs to prevent signal propagation into other tiles. A 2:1 MUX connects one of two sub-CBs to the output port, according to the grant signals from its scheduler.

An 8 × 8 crossbar fabric with CPAT is comparatively analyzed with a conventional one. In this crossbar design, RBs and CBs are laid out in M2 and M1 layers, respectively. The area of the fabric is about 240 × 240 um^2. According to the capacitance extraction from the layout, the capacitances of an RB and a CB onto the substrate are 44 fF and 28 fF, respectively, and the coupling capacitance between adjacent bars is 13 fF. Figure 7.20 shows how the comparative power consumption varies with offered load. As the offered load increases, the power reduction becomes more significant. At 90% offered load, 22% power saving is obtained. The CPAT cuts down the capacitive loading on RBs and CBs by half. However, because the power consumption on output drivers and main RBs is not reduced, the power reduction does not exceed 25%. The additional OR-gates and MUXs consume less than 2% of the overall power. When CPAT is applied to a 16 × 16 crossbar switch that is divided into 4 × 4 tiles, 43% power saving is obtained.

7.6.2 Switch Scheduler

To arbitrate the output conflicts, a scheduler is used on each output port. The arbitration scheduling increases the latency, power, and area of the switch design. The latency of the arbiter becomes larger than that of the switch fabric as the switch size becomes bigger than 16 × 16 [21]. Furthermore, the area cost is not ignorable when we use a serialized link [12]. The scheduler occupies an area similar to the switch

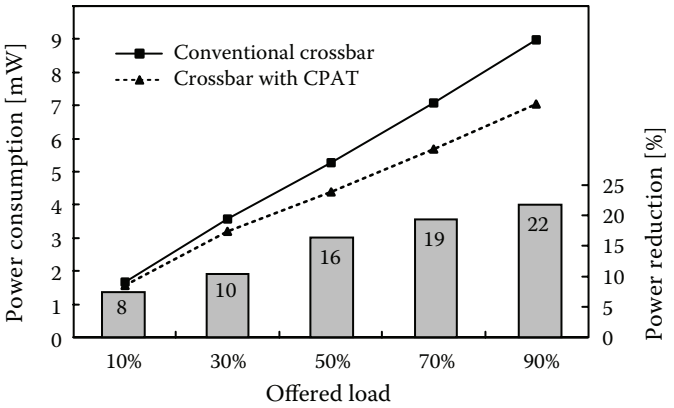


FIGURE 7.20 Power comparison of an 8 × 8 crossbar fabric with—and without—crossbar partial activation technique.

fabric when the phit width of a port is 10 bits. Therefore, the scheduler design is as important as the switch fabric design.

7.6.2.1 Low-Power Scheduler: Mux-Tree-Based Round-Robin Scheduler

A scheduler (or arbiter) is needed in a crossbar switch when more than two input packets from different input ports are destined for the same output port at the same time. Among a number of scheduling algorithms, a round-robin algorithm is most widely used in ATM switches and on-chip networks because of its fairness and lightness [Shahrier-ATM01], [35]. There are many methods of implementing the round-robin algorithm [35,36,37].

Figure 7.21 shows a Mux-Tree-based implementation whose structure is highly modulated, scalable, and power-efficient [12]. A scheduling latency is O(log n), and the required resources are O(n), where n is the number of ports in a crossbar switch.

The round-robin scheduler has a rotating pointer that indicates a recently granted port. A port next to the pointer has the highest priority to be granted, for example, the request vector <7:0> = 01100010, where underline indicates the position of the pointer. Then, the port <4> has the highest priority, and the lower group of port <4:0> has higher priority than the upper group of port <7:5>. This information is given by a thermoencoder, whose output becomes token <7:0> = 00011111. Therefore, the port <4:0> has its tokens. A request from a port having a token "1" acquires higher priority than the request from a port that has no token. These request and token vectors are inputs of the binary Mux-Tree, which consists of tiny arbiters (TAs) at each node. Each TA selects one of two ports, the upper one or the lower one, based on the table shown in Figure 7.21b. When both requests have tokens, TA selects the upper port because the pointer rotates in decreasing order. UorL, Q.req, and Q.token bits generated at each TA propagate and are inputted to its parent node. Then, one of two children's UorL bits is selected by 2:1 MUX based on their parent's UorL bit, after which the selected child-UorL bit and its parent-UorL bit are concatenated and propagate again. By doing so successively up to the root node, the granted port number is finally determined.

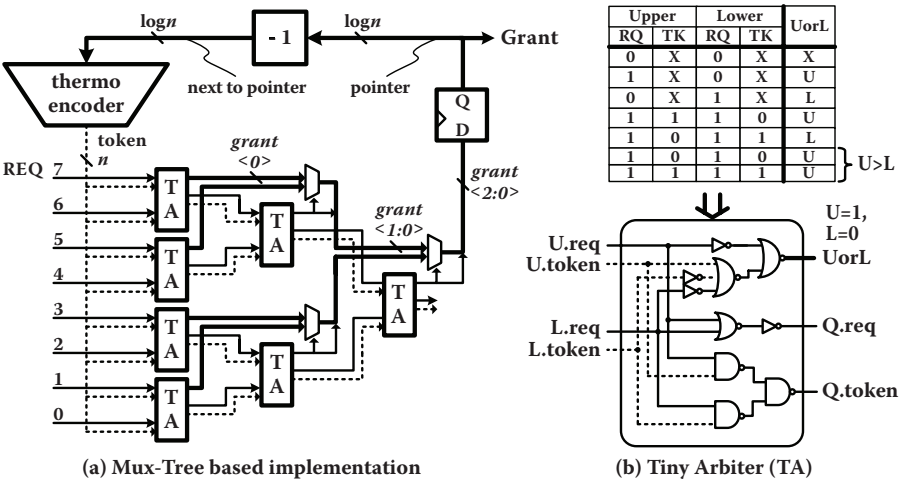


FIGURE 7.21 Mux-Tree-based Round-Robin Scheduler: (a) block diagram, (b) tiny arbiter.

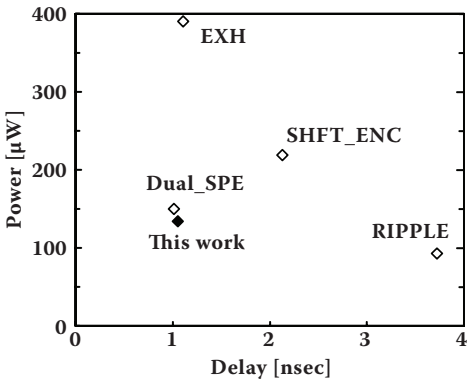


FIGURE 7.22 Power and delay comparison with other round-robin implementations.

The Mux-Tree-based implementation is compared with four other designs such as EXH, SHFT_ENC, RIPPLE, and DUAL_SPE presented in Reference 35. Figure 7.22 shows a comparison of power consumption and scheduling delay simulated with 8-input ports in 0.18 μm process technology. The Mux-Tree scheduler achieves the minimum power and delay product, 136 uW and 1.05 ns delay at 100 MHz, and offered a load of 50%. This design also requires the minimum number of transistors, i.e., area, except the RIPPLE design, as shown in Table 7.4.

TABLE 7.4
Comparison of the Number of Required Transistors

	8 ports	16 ports
RIPPLE	403	927
EXH	1435	6879
SHFT_ENC	629	1711
DUAL-SPE	573	1483
MUX-TREE	569	1203

7.7 LOW-POWER NETWORK ON CHIP PROTOCOL

7.7.1 Protocol Definition

In general terms, a *protocol* is defined as a set of rules governing communication within and among processing nodes. A protocol is a convention or standard that controls or enables connection, communication, and data transfer between two computing endpoints. In its simplest form, a protocol can be defined as the rules governing the syntax, semantics, and synchronization of communication. Protocols may be implemented by hardware, software, or a combination of the two. The most popular and simplest protocol for embedded systems is the AMBA protocol. It defines the interface signals and transaction types between a master and a slave node [38].

A NoC protocol is similar to conventional bus protocols, such as an AMBA or OCP-IP [38,39]. However, it is distributed, and thus looks more complicated than them. Many concepts of the NoC protocol originate from computer network protocols such as LAN (Ethernet) or WAN (ATM, TCP/IP).

An example of an NoC protocol is given in the appendix for your information.

7.7.2 Protocol Composition

First, it defines the interface signals between a network interface and a processing node, which might be a master or slave. For example, Figure 7.23 shows the interface signals [40]. See the appendix for the detailed protocol.

Second, it defines a packet format that consists of a header and a payload, as shown in Figure 7.24, for example. Necessary information for transactions, such as

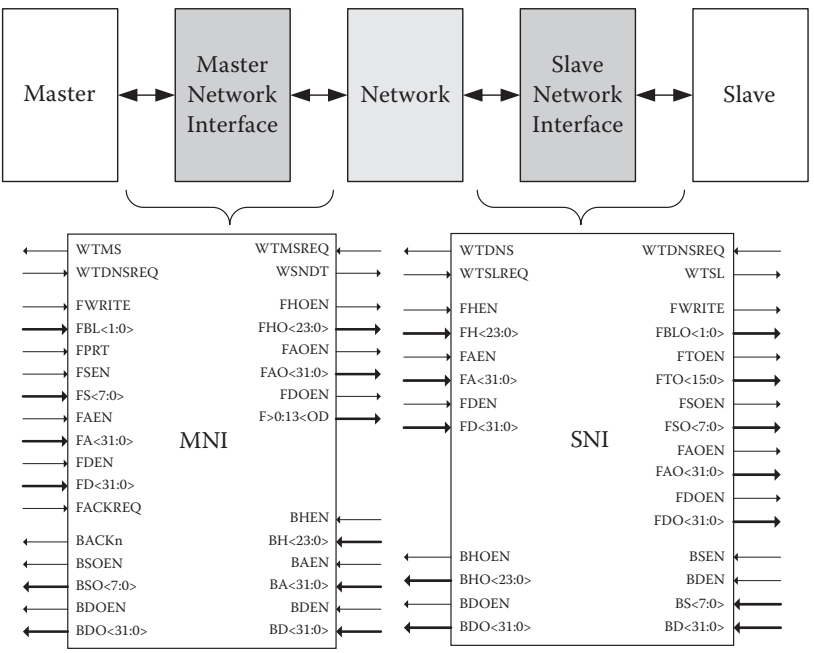


FIGURE 7.23 Network interface signals.

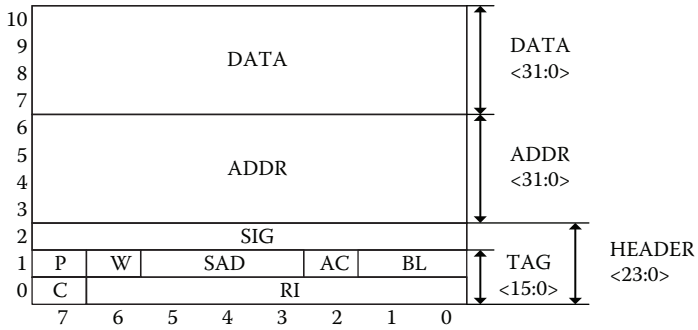


FIGURE 7.24 Packet format.

routing information (a source and destination network addresses), burst length and type, read/write indicator, acknowledgement request, and other specific fields for error handling, QoS control, flow control, and so on, is composed in a header field. A payload can be designed as fixed size or flexible size. Fixed packet length is preferable for a simpler network interface hardware.

Third, it defines a protocol stack that consists of link-level (L2), a network-level (L3), and a transport-level (L4) policies such as the following:

- Link-level policy: Low-power coding, link-level flow-control, and link-level error-control
- Network-level policy: Switching and routing policy
- Transport-level policy: End-to-end flow control, end-to-end error-control, and QoS policy

7.7.3 Low-Power Issues on the NoC Protocol

7.7.3.1 Aligned Packet Formation

An aligned-packet format helps provide an efficient hardware implementation in terms of power and area consumption. Typically, packet, flit, and phit formats are independently defined because they do not affect each other from a functional point of view. However, such independently defined formats cause a packet-processing burden for a NoC. Because a NoC's data link, network, and even its transport layers should be implemented with hardware, misalignment among packet, flit, and phit formats makes the parsing and processing operations difficult. Instead of the typical packet definition, shown in Figure 7.25a, we recommend an aligned-packet definition that defines a packet format in relation to the physical layer structure, as shown in Figure 7.25b. This packet format defines a fixed-length packet, and each packet field has dedicated link wires. This scheme has two advantages over the typical packet definition scheme: The packet parsing procedure is very simple, and the additional link wires can easily increase a field's bit width. A disadvantage of this scheme, however, is that link use is inefficient if some fields are disabled. For example, if a packet carries a payload 2 that is empty, the link wires corresponding to payload 2—that is, the link wires from the 14th to the 21st link, or simply link <21:14>—

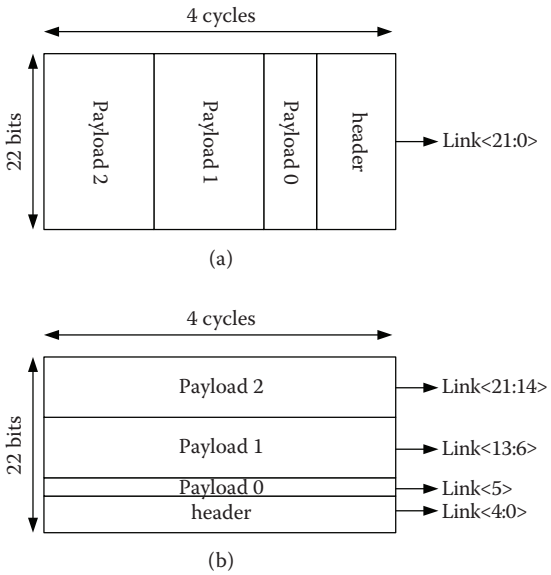


FIGURE 7.25 Packet alignment: (a) typical format and (b) aligned format.

remain idle, but other packet transactions cannot use the idle link wires. The typical packet definition, on the other hand, can use the channel resources efficiently. Even though a shorter packet is transferred, all the link wires are used and the packet's transmission time is shorter than that of a long, or aligned, packet. Nevertheless, the disadvantages of the typical packet definition scheme—a complex packet-parsing procedure and an inflexible bit width adjustment—persuaded us to use the aligned packet definition. In the implementation results for the Slim-spider chip [12], which uses a typical nonaligned-packet format, the parsing overhead occurs because of a network interface's power consumption at the destination. Control logic and data path power consumption of the network interface increase 3× and 1.3×, respectively, compared with the aligned format. As a result, the overall network interface's power consumption increases by about 50% [21].

7.7.3.2 Packet Switching versus Circuit Switching

A switching mode can be categorized into two types: packet switching and circuit switching. In a circuit-switching mode, a route is reserved before sending a data or packet. During a sending the packet through the reserved route, it doesn't arbitrated thus doesn't buffered in a switch. However, other packets that are supposed not to use the reserved route cannot share the route during the reservation and the packet transmission. Therefore, the network throughput may be degraded and underutilized. In a packet-switching mode, a packet is sent without such a prior reservation. The packet is arbitrated at every switching hop, and thus it needs to be buffered during the arbitration phase.

As you can see in Table 7.1, the buffering energy is greater than the other energy on a switching fabric and a link. Therefore, you can simply guess that the

packet-switching mode would consume more energy than the circuit-switching mode, in general. However, you should consider the energy per bandwidth, i.e., J/Mbps, to compare the two modes fairly. The circuit-switching mode provides less throughput when the offered traffic load is high.

Chang et al. compared the two modes with several multimedia applications [41]. Their results show that the packet-switching mode consumes more energy—about 2×–10× than the circuit-switching mode. Energy consumption strongly depends on the traffic locality and topology also. When the traffic is localized, the probability of blocking on switches is reduced. Thus, circuit switching is more advantageous in such a localized traffic condition.

You should investigate the traffic locality and the blocking probability on switches in your application and in your topology before you decide the switching mode. In terms of power consumption, circuit switching is more advantageous if it can provide the needed network throughput without performance degradation. Consider that the conventional bus protocols use circuit-switching mode nowadays.

REFERENCES

1. Vangal, S. et al., An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers,* 2007, pp. 98–99.
2. Hoffman, J. et al., Architecture of the Scalable Communications Core, in *ACM/IEEE Int. Symp. On Networks-on-Chip,* 2007, pp. 40–52.
3. Chandrakasan, A. et al., *Design of High-Performance Microprocessor Circuits,* IEEE Press, p. 360, 1999.
4. Stan, M. R. et al., Bus-Invert Coding for Low-Power I/O, *IEEE Trans. VLSI systems,* Vol. 3, pp. 49–58, March 1995.
5. Mehta, H. et al., Some Issues in Gray Code Addressing, in *Proc. of Great Lakes Symp. on VLSI,* Mar. 1996, pp. 178–181.
6. Benini, L. et al., Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems, in *Proc. of Great Lakes Symp. on VLSI,* March 1997, pp. 77–82.
7. Shin, Y. et al., Partial Bus-Invert Coding for Power Optimization of System Level Bus, in *Proc. of Int. Symp. on Low Power Electronics and Design,* Aug. 1998, pp. 127–129.
8. Ramprasad, S. et al., A Coding Framework for Low-Power Address and Data Busses, *IEEE Trans. VLSI systems,* Vol. 7, pp. 212–221, June 1999.
9. Kretzschmar, C. et al., Why Transition Coding for Power Minimization of on-Chip Buses does not work, in *Proc. of the Design Automation and Test Europe Conf.* (DATE), February 2004, pp. 512–517.
10. Shin, Y. et al., Narrow Bus Encoding for Low-Power DSP Systems, *IEEE Trans. VLSI systems,* Vol. 9, pp. 656–660, October 2001.
11. Lee, S. et al., An 800MHz star-connected on-chip network for application to systems on a chip, in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers,* 2003, pp. 468–469.
12. Lee, K. et al., Low-power network-on-chip for high-performance SoC design, *IEEE Trans. VLSI systems*, Vol. 14, February 2006, pp. 148–160.
13. Lee, K. et al., SILENT: serialized low energy transmission coding for on-chip interconnection networks, in *ACM/IEEE Int. Conf. on Computer-Aided Design,* 2004, pp. 448–451.
14. Ho, R. et al., Efficient On-Chip Global Interconnects, in *IEEE Symp. on VLSI Circuits Dig. Tech. Papers,* June 2003, pp. 271–274.

15. Zhang, H. et al., Low-Swing On-Chip Signaling Techniques: Effectiveness and Robustness, *IEEE Trans. VLSI systems,* Vol. 8, June 2000, pp. 264–272.
16. Moisiadis, Y. et al., High Performance Level Restoration Circuits for Low-Power Reduced-swing Interconnection Schemes, in *Proc. of Int. Conf. on Electronics Circuits and Systems*, December 2000, pp. 619–622.
17. Golshan, R. et al., A novel reduced swing CMOS BUS interface circuit for high speed low power VLSI systems, in *Proc. of IEEE Int. Symp. Circuits and Systems,* May 1994, pp. 351–354.
18. Nakagome, Y. et al., Sub-1-V Swing Internal Bus Architecture for Future Low-Power ULSI's, *IEEE J. of Solid-State Circuits,* Vol. 28, pp. 414–419, April 1993.
19. Cardarilli, G. C. et al., Low Voltage Swing Circuits for Low Dissipation Buses, in *Proc. of Int. Symp. on Circuits and Systems,* June 1997, pp. 1868–1871.
20. Hiraki, M. et al., Data-Dependent Logic Swing Internal Bus Architecture for Ultralow-Power LSI's, *IEEE J. of Solid-State Circuits,* Vol. 30, pp. 397–402, April 1995.
21. Lee, S. et al., Analysis and Implementation of Practical, Cost-Effective Networks on Chips, *IEEE Design and Test of Computers,* September 2005, pp. 422–433.
22. Svensson, C., Optimum Voltage Swing on On-Chip and Off-Chip interconnect, *IEEE J. of Solid-State Circuits,* Vol. 36, pp. 1108–1112, July 2001.
23. Worm, F. et al., A Robust Self-Calibrating Transmission Scheme for On-Chip Networks, *IEEE Trans. VLSI systems,* vol. 13, pp. 126–139, January 2005.
24. Lee, K. et al., A 51mW 1.6GHz On-Chip Network for Low-Power Heterogeneous SoC Platform, in *IEEE Int. Solid-State Circuits Conf. Dig.* Tech. Papers, 2004, pp. 152–153.
25. Wang, H. S. et al., A Power Model for Routers: Modeling Alpha 21364 and InfiniBand Routers, *IEEE Micro,* Vol. 23, No. 1, January-February 2003, pp. 26–35.
26. Villiger, T. et al., Self-timed Ring for Globally-Asynchronous Locally-Synchronous Systems, in *Proc. of Int. Symp. on Asynchronous Circuits and Systems,* 2003, pp. 141–150.
27. Chattopadhyay, A. et al., High speed asynchronous structures for inter-clocking domain communication, *Int. Conf. on Electronics, Circuits and Systems,* 2002, pp. 517–520.
28. Muttersbach, J. et al., Practical Design of Globally-Asynchronous Locally-Synchronous Systems, in *Proc. of Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* pp. 52–59, 2000.
29. Sriram Vangal et al., An 80-Tile 1.28TFLOPS Network-on-chip in 65nm CMOS, ISSCC 2007, pp. 98–99.
30. Dally, W. J. and Poulton, J. H., *Digital Systems Engineering*, Cambridge University Press, Cambridge, 1998.
31. Lee, S. et al., Adaptive Network-on-Chip with Wave-Front Train Serialization Scheme, in *IEEE Symp. on VLSI Circuits Dig.* Tech. Papers, June 2005, pp. 104–107.
32. Woo, R. et al., A 210mW Graphics LSI Implementing Full 3D Pipeline with 264Mtexels/s Texturing for Mobile Multimedia Applications, *ISSCC Dig. of Tech. Papers*, pp. 44–45, 2003.
33. Sinha, M. et al., Current-sensing for crossbars, *IEEE Int. ASIC/SOC Conf. 2001*, pp. 25–29.
34. Wijetunga, P. et al., High-performance crossbar design for system-on-chip, *IEEE Int. Workshop on SoC for Real-time Applications,* 2003, pp. 138–143.
35. Gupta, P. et al., Design and Implementing a Fast Crossbar Scheduler, *IEEE Micro,* Vol. 19, pp. 20–28, January 1999.
36. Lee, K. et al., A Variable Round-Robin Arbiter for High Speed Buses and Statistical Multiplexes, in *Proc. Int. Phoenix Conference on Computers and Communications,* March 1991, pp. 23–29.

37. Shin, E. et al., Round-robin Arbiter Design and Generation, in *Proc. IEEE Int. Symp. System Synthesis,* pp. 243–248, October 2002.
38. AMBA™ Specification, Rev. 2.0, 1999, www.arm.com.
39. OCP 2.0 Protocol Specification, www.ocpip.org.
40. BONE: KAIST NoC protocol, http://ssl.kaist.ac.kr/ocn.
41. KueiChung Chang, JihSheng Shen, and TienFu Chen, Evaluation and Design Trade-Offs Between Circuit Switched and Packet Switched NOCs for Application-Specific SOCs, in *Proc. Design Automation Conference,* 2006, pp. 143–148.

8 Real Chip Implementation

8.1 INTRODUCTION

Network on Chip (NoC) architectures are emerging as a strong candidate for highly scalable, reliable, and modular on-chip communication infrastructure platform. There have been many architectural and theoretical studies on NoCs such as design methodology, topology exploration, quality-of-service (QoS) guarantee, and low-power design. All of these have been discussed in the previous chapters in this book. In this last chapter, we introduce silicon chip implementation trials for NoC-based SoCs. They can be grouped into two categories: academic research and industrial approaches. Academic research shows complete chip implementations and demonstrations for specific applications. On the other hand, industrial approaches are mainly regarding the new protocol specifications, EDA tool chain, and IP library support for the NoC developers. In this chapter, both works from the academies and industries will be discussed in detail.

8.2 BONE SERIES

For the unique purpose of realizing the new NoC technology through implementation, the BONE (Basic On-chip Network) project was launched in 2002 at KAIST (Korea Advanced Institute of Science and Technology, Daejeon, Korea). The project is proving to be fruitful; new NoC techniques and implementations are published and demonstrated every year, as summarized in Table 8.1.

This project covers circuit-level design, architectural research, and system integration on a NoC platform. In this section, each generation will be overviewed, and for more detailed information on BONE, refer to the Web site http://ssl.kaist.ac.kr/ocn/.

8.2.1 BONE 1: Prototype of On-Chip Network (PROTON)

To demonstrate feasibility of the on-chip network (OCN) architecture, a test chip is implemented using 0.38 μm CMOS technology. The BONE 1 is designed with two physical-layer features: high-speed (800 MHz) mesochronous communication, and on-chip serialization (OCS). Using a 4:1 serialization, 80b packets are transferred through 20b links. The 4:1 serialization reduces the network area of BONE 1 by 57%, which enables it to be used in SoC design. The distributed NoC building blocks are not globally synchronized, and the packet transfer is performed with mesochronous communication. Because mesochronous communication eliminates the burden of global synchronization, high-speed clocking, 800 MHz, is possible. The implementation and successful measurement demonstrates that high-performance on-chip

TABLE 8.1
BONE Series

Generation	Code name	Features	Chip specification	Publications
BONE 1	PROTON (The first prototype of BONE)	Star topology Plesiochronous clocking Serialization Off-chip gateway	0.16 μm DRAM Tech. 10.8 × 6 mm^2 800 MHz 264 mW	ISSCC 03 [1] TCAS 05 [2]
BONE 2	SlimSpider (Low-power NoC for multimedia)	Multiprocessors Small-swing link Partial activation tech. SiLENT serial coding Frequency scaling	0.18 μm CMOS Tech. 5 × 5 mm^2 1.6 GHz 51 mW	CICC 03 [3] ISSCC 04 [4] ICCAD 04 [5] SOCC 04 [6] A-SSCC 05 [7] TVLSI 06 [8]
BONE 3	IIS (Intelligent interconnection system)	Serial wave-front-train Adaptive bandwidth ctrl. Programmable delay synchronizer Adaptive circuit/ packet switching mode	0.18 μm CMOS Tech. 5 × 5 mm^2 400 MHz, 3.0 Gbps	S.VLSI 05 [9] D&T 05 [10]
BONE 4	FONE (Flexible NoC platform)	NoC evaluation board Adaptive bandwidth ctrl. Dynamic traffic monitoring		ISCAS 05 [11] ISCAS 05 [12]
BONE V1	MC-NoC (Memory centric NoC)	Robot visionary application Ten homogeneous MPCores Distributed shared memory	0.18 μm CMOS Tech. 7.7 × 5.0 mm^2 400 MHz	NOCS 07 [15] CICC 07

serialized networking with mesochronous communication is practically feasible. The chip size is 10.8 mm × 6.0 mm, the number of transistors is 81,000, and power consumption is 264 mW at 800 MHz in 2.3 V operations.

8.2.1.1 Overall Architecture

Figure 8.1 shows an overall block diagram of the OCN with BONE 1. It consists of network interface (NI), Up_Sampler (UPS), link wires, first in first out (FIFO) synchronizer with a queuing buffer (SYNC), switch, Down_Sampler (DNS), and off-chip gateway (OGW). Using an address and/or a data output of a processing unit (PU), a corresponding NI generates a packet. A packet is an 80b bit stream consisting of a 16b header and a 64b payload. In the header, the routing information is

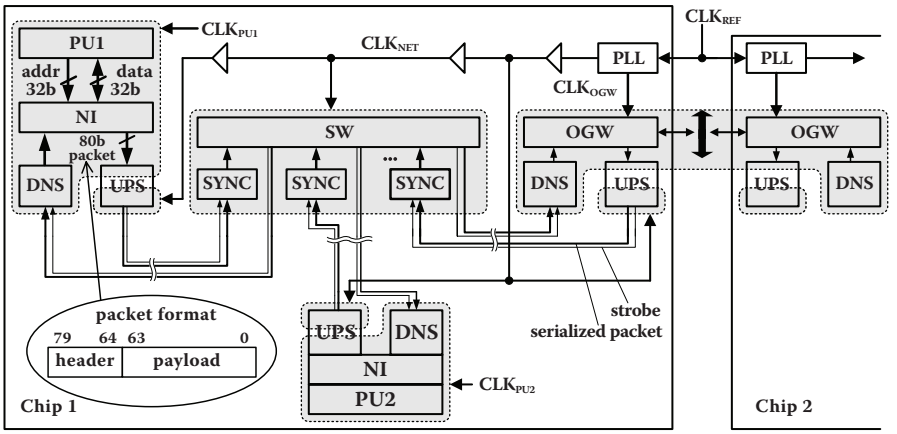


FIGURE 8.1 Overall architecture of the NoC for BONE 1.

encoded into 16b, and the switches route the packet according to the header information. The packet also can be transferred to another chip through the OGW. An NI has an address map that defines a destination PU according to an address, and a header of a packet is generated based on the address map. The packet header size is fixed to reduce latency and hardware complexity of the header's parsing logic.

The UPS, link wires, SYNC, switch, and DNS are called the *core network*. The core network delivers a packet to a correct destination PU. It is serialized to lower the wiring complexity by reducing the number of link wires. Instead, it uses its own high-frequency clock to sustain bandwidth of the network system. The UPS and the DNS interface the high-speed serialized core network to the NI.

Clock domains of the OCN are illustrated as dotted gray boxes in Figure 8.1. Even without global synchronization, each PU can use its dedicated clock source constituting a distinct clock domain. The clock for the OGWs (CLK_{OGW}) is slow enough to do chip-to-chip signaling, and it is synchronized with the reference clock (CLK_{REF}) so that all the OGWs connected together are synchronized with one another. The CLK_{NET} is distributed without any skew compensation, and the packet is transmitted using mesochronous communication. When a packet traverses from one clock domain to another, a strobe signal (STB) is sent together with the packet as a timing reference; the DNS and SYNC compensate phase differences between the STB signal and the local clock. This architecture eliminates the burden of global synchronization, thus enabling the use of high-speed clocks, i.e., 800 MHz in this case, which facilitates performance enhancement of an on-chip system.

8.2.1.2 Packet Routing Scheme

A packet is transferred to a destination according to the RI field. A response packet from the destination, if necessary, returns to the source along the reverse path. The RI contains a series of port indexes, which are called output port indexes (OPIs). An OPI_n is an index of an output port to which a packet must be routed at the nth switch. For a $k \times k$ switch, k–1 port indices are required, because the port where the packet

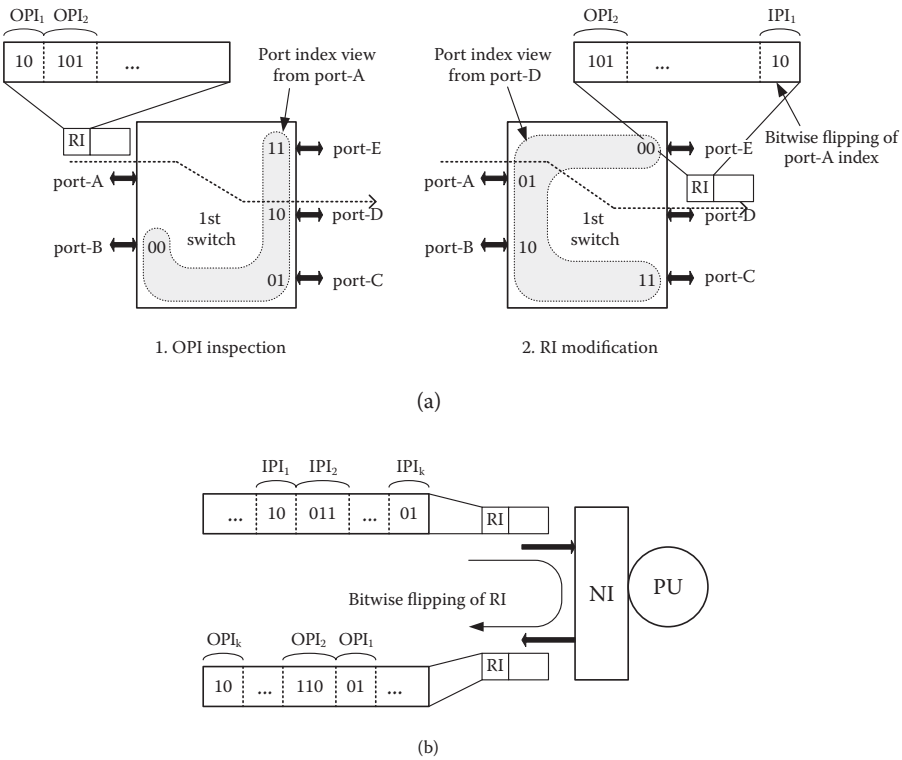


FIGURE 8.2 (a) Packet routing using RI and (b) RI generation for reverse path.

has entered does not use port indexes. Figure 8.2 shows the packet routing process using the RI field. When a packet arrives at a switch, the switch checks an OPI at the head of the packet header. Then, the header is left-shifted to locate the next OPI at the most-significant-bit part of the packet header, and the port index of the input port (IPI) is attached to the tail. The IPI is a port index that is viewed by the output port. Through this RI modification, each switch can find its OPI at the head of the RI field, which enables cut-through switching. Also, an NI of a destination PU obtains a header for the return path by bitwise flipping of the header. An IPI is a flipped index of an input port, so the bitwise flipping outputs correct the OPIs.

This scheme enables switches that do not have routing tables to hold topology-dependent data. Therefore, network design is decoupled from system design, and the network can be easily ported to a system. In conventional routing schemes described in most of the literature, the routing information header contains source and destination addresses, and each switch has its own routing table. In such a typical scheme, the configuration of the routing table depends on the OCN topology and, therefore, the overall SoC architecture should be determined to fill out the routing table. In other words, the routing table must be redesigned at every use. This means that network design is not completely decoupled from system design. A programmable routing table may be a solution for this issue. However, it requires a kind of content-addressable memory, whose area and power overheads are considerable. It also

requires additional protocol burden to program the table. A disadvantage of the proposed routing method is the difficulty in supporting switch-level adaptive routing. If, however, NIs gather traffic information, adaptive routing at the packet-transaction level is possible.

The number of PUs that can be covered by a 14b RI field depends on the network topology, whereas in the conventional scheme it is fixed at 2^7. In a hierarchical star topology, the proposed routing method covers more than 2^8 PUs, regardless of the levels of hierarchy. In a mesh topology, the RI field can cover up to seven switch hops or a 4×4 mesh network, which is still reasonable for on-chip networking.

8.2.1.3 Off-Chip Connectivity

OGWs provide chip-to-chip packet transactions without any additional external component. And the interchip communication does not require a specific header type, but uses the same format employed in the core network. Figure 8.3 shows a configuration of a multichip interconnection using OGWs. The OGWs are connected together, constituting a large virtual switch, and the interchip packet transaction is performed through the virtual switch. An OGW acts similar to a switch port, and the external bus functions as a switch fabric. Each OGW has its own identification index (ID), which is used like the port index of a switch. Access to the bus is arbitrated based on token-ring methodology.

If an OGW having a packet to transmit gets a token, it outputs the packet onto the bus, with assertion of the OEN signal. Other OGWs monitor the OEN, and if it is asserted, they check whether the OPI of the packet matches their IDs. If they match, the OGW accepts the packet and performs header modification.

Using the bus architecture with token-based arbitration, chip-to-chip interconnection can be accomplished with only wires, without any external switch or arbiter.

8.2.2 BONE 2: Low-Power Network on Chip and Network in Package (Slim Spider)

In large-scale SoCs, power consumption of the communication infrastructure should be minimized for reliable, feasible, and cost-efficient chip implementations. In the

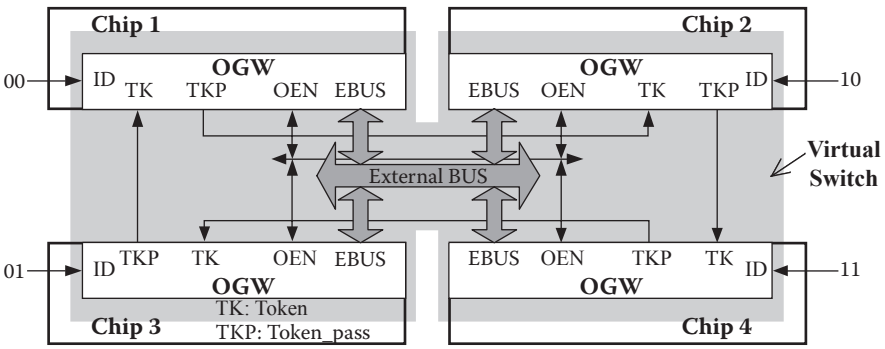


FIGURE 8.3 Chip-to-chip connection using off-chip gateways.

BONE 2 project, a hierarchically star-connected NoC is designed and implemented with various low-power techniques. The fabricated chip [4] contains heterogeneous IPs such as two RISC processors, multiple memory arrays, FPGA, off-chip network interfaces, and 1.6 GHz PLL. The integrated OCN provides 89.6 Gbps aggregate bandwidth and consumes less than 51 mW at full traffic condition. On the other hand, the previous project, BONE 1, consumes 264 mW with 51 Gbps bandwidth. The ratio of power consumption to provided bandwidth in BONE 2 is ten times less than that for BONE 1. In this section, the various low-power techniques proposed for on-chip networks will be explained briefly.

Complicated SoCs with large chip size, such as embedded memory logic systems, often suffer from problems of their low yield and high cost. The NoCs have such a modular structure that larger systems can be divided into several parts or several chips to mitigate such yield and cost problems. In this project, four NoCs are mounted on a single package for larger system integration to form the Network in Package (NiP). The chip-to-chip interconnections in a package exploit low-resistive PCB-printed wires for low-latency and low-jitter off-chip communications [13].

For the design of NoC, there are many levels of design choices to be decided, such as communication protocol, network topology, switching style, buffer depth (in a router), clock synchronization method, signaling scheme, and so on. In this section, a brief summary of design decisions at each stage, and their rationale, will be presented on the basis of the *low-power consumption of the SoC*.

8.2.2.1 NoC Architecture

The first phase of NoC architecture design is choosing the most suitable NoC topology. In this work, the power and area cost of the most popular topologies such as the multilayer bus, 2D-mesh, and the newly proposed hierarchical star (H-star) topology [8] are examined. According to the analytical evaluation shown in Figure 8.4, the H-star topology shows the lowest energy consumption not only under uniform traffic but also under nonuniform localized traffic conditions. The area cost of the H-star is much lower than the mesh and comparable to that of the bus topology. Therefore, H-star topology is chosen for the NoC platform. It is found to be cost-efficient and has fewer switching hops (see Chapter 6 for the energy analysis of the various hierarchical topologies).

A state-of-the-art SoC is a heterogeneous multiprocessing system with multiple timing references; the difficulty of global synchronization as well as third-party PUs is that they use independent clock-frequency scaling. To cope with multiple clock domains, in this implementation, each PU operates with its own clock, CLK_{Pui}, but it communicates with a single clock, CLK_{NET}. The network interface (NI) changes the timing reference from the CLK_{Pui} clock to CLK_{NET} and vice versa. CLK_{NET} is not synchronized over the chip, so the communications become mesochronous—having the same frequency but different skew. For mesochronous communications, a source-synchronous scheme is adopted, in which a strobe signal goes along with the packet data. The strobe signal is used as a timing reference to latch the packet data to the receiver terminal. A packet consisting of a 16-bit header, 32-bit address, and 32-bit data fields is serialized into 8-bit channels to reduce the network area and power consumption.

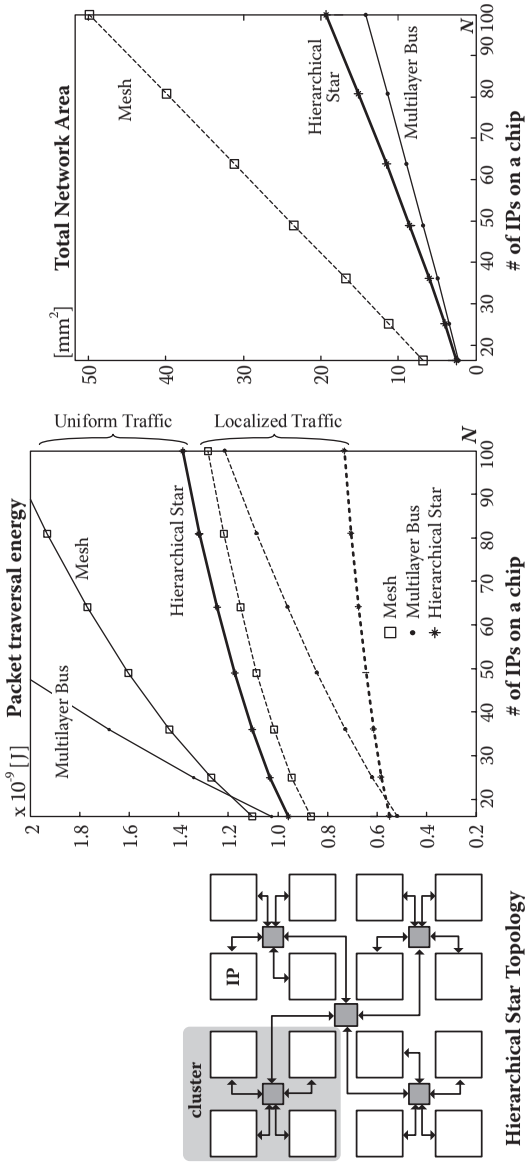
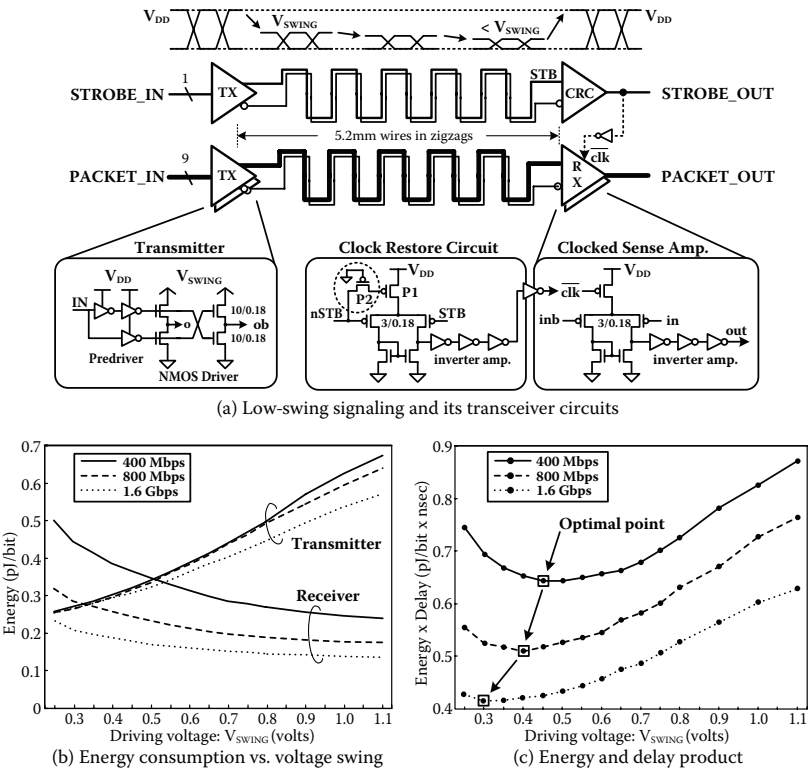


FIGURE 8.4 Energy and area cost analysis of multilayer bus, mesh, and hierarchical star topologies.



(a) Low-swing signaling and its transceiver circuits

(b) Energy consumption vs. voltage swing

(c) Energy and delay product

FIGURE 8.5 Low-swing signaling: circuits and V_{SWING} optimization.

8.2.2.2 Low-Power Techniques

The global link connecting switches are usually a few millimeters long. Therefore, it suffers from longer latency and higher power consumption than a local link and makes cross-chip communication increasingly expensive. Low-swing signaling can alleviate energy consumption significantly, and overdriving signaling improves delay.

Figure 8.5a shows the implemented differential low-swing signaling. Global wires are laid out in zigzags to emulate a long link, as long as 5.2 mm, without repeaters. To find out the optimum voltage swing, postlayout simulation is performed with a precise capacitance and resistance wire model [4]. Figure 8.5b shows how power consumption on a transmitter and a receiver varies with V_{SWING}. Figure 8.5c shows the energy and delay product. The delay from a transmitter to a receiver is about 0.9 ns, and its variations with V_{SWING} or signal rates are as small as ± 40 ps. As shown in the figure, the optimal swing voltage is 0.45, 0.40, and 0.30 V at 400 Mbps, 800 Mbps, and 1.6 Gbps signal rates, respectively. At each operating mode, the driving voltage scales to the aforesaid optimal voltage level. Due to low-swing signaling, the power dissipation on the global link is reduced to one third of that on a full-swing repeated link. In addition, there are no area-consuming repeaters on the wires.

Crossbar is widely used in the router as the switching fabric. In BONE 2, Crossbar Partial Activation Technique (CPAT) is used to reduce the power consumption on

the crossbar [4]. The CPAT removes unnecessary activation by using tri-state buffers and multiplexers. As a result, 22% power saving is obtained on an 8 × 8 crossbar.

On-chip source-synchronous serial communications has many advantages over multibit parallel communication in the areas of skew, crosstalk, area cost, wiring efficiency, and clock synchronization. However, the serial wire tends to dissipate more energy than the parallel bus owing to bit multiplexing. In BONE 2, a novel coding method, SILENT [5], is proposed to reduce the transmission energy of the serial communications by minimizing the number of transitions on the serial wire. The transition reduction coding can save significant amounts of the communication energy for multimedia applications. It reduces a maximum of 77% energy for instruction memory access and 40 to 50% energy for data memory access in a 3-D graphics application.

PLL generates internal clocks, such as a 100 MHz clock for the main cluster PUs, a 50 MHz clock for peripheral cluster units, and a 1.6 GHz network clock for the switches and network interfaces. The clock frequencies are scalable for power management modes, i.e., 100/50/1600 MHz for the FAST mode, 50/25/800 MHz for NORMAL mode, and 25/12.5/400 MHz for SLOW mode.

8.2.2.3 Design Methodology and Chip Implementation

The NoC was designed by a semicustom method. Integrated processors are synthesized, memories are compiled by the SRAM compiler, and the on-chip networks are fully custom-designed for low power and high performance. The processors and memories are obtained from vendors as IPs and reused by attaching the network interface and wrappers. Figure 8.6 shows the design flow; EDA tools, design stage, and the output deliverables at each stage. It took 6 months from architecture sketch to tape-out with seven engineers, which is a short development time for complicated SoC, by using NoC.

With the proposed NoC architecture, protocol, and low-power techniques, a multimedia SoC is implemented as a prototype. The chip photograph and a block diagram are shown in Figure 8.7. The chip integrates two clusters, a main cluster and a peripheral cluster. The main cluster contains two RISC processors, on-chip FPGA, two 64 kb SRAMs, and an OGW. The two RISC processors emulate multiprocessor systems. The OGW [1] enables seamless off-chip communications with other NoCs on the same package or boards to compose larger-scale systems. By using the OGW, a PU on a die can communicate with other PUs the other dies without protocol conversion. The peripheral cluster contains three memories to emulate peripheral slave units. The two clusters are interconnected with each other through a 5.2 mm global link that uses low-swing signaling to reduce power consumption and also differential signaling for higher SNRs. PLL generates scalable clocks for PUs and networks. PU clocks are not synchronized with one another for the emulation of systems with multiple timing references. Therefore, no effort is needed for clock-skew minimization. The OCN supports a 3.2 Gbps communication bandwidth for each PU and 11.2 Gbps aggregate bandwidth at FAST mode. The chip is implemented using a 0.18 μm CMOS process with 6-Al metal layers, and its die area takes 5 × 5 mm^2. The OCN power dissipates 51 mW at FAST mode with full traffic condition. By adopting the

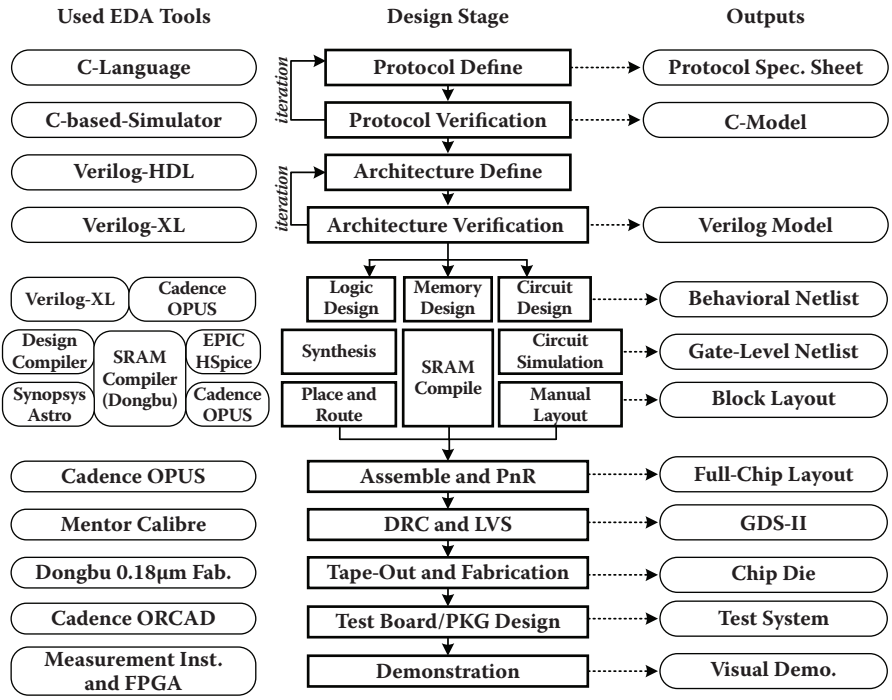


FIGURE 8.6 Chip and system design flow.

proposed techniques such as low-swing signaling, crossbar-partial activation, and serial-link coding, the overall power consumption is reduced by 43%.

8.2.2.4 Networks in Package and Measurement

Four NoCs are mounted on a single 676-BGA package as shown in Figure 8.8d. The NiP needs four isolated supply voltages: 1.8 V for digital logic, Quiet 1.8 V for analog circuits, 3.3 V for I/O, and sub-0.6 V for low-swing links. The operating frequencies are different for each module: 50 MHz for the peripheral logic, 100 MHz for processors, 800 MHz for the scheduler, and 1.6 GHz for OCNs.

The important issue for the package design is the power integrity, i.e., a design for power and ground (P/G) networks. No significant resonance should occur on the P/G plane of the package at the operating frequency. Small signal noise or the external system causes such a significant P/G noise that the P/G network becomes unstable. To analyze the power integrity, the Transmission Line Matrix (TLM) method and Simultaneous Switching Noise (SSN) analysis are performed (see Figure 8.8a, 8,8b). Decoupling capacitors for each power voltage are integrated at the proper positions: five for logic power, two for I/O power, and one for analog power. Each capacitor has 10 nF capacitance and 600 pH effective series inductance properties. Figure 8.8a shows the self-impedance of the power plane. The target value of the self-impedance of the power plane is 1 Ω. The solid line is for the bare P/G plane, and the dotted line is for the proposed insertion of decoupling capacitors. The impedance of the bare

FIGURE 8.7 BONE 2 (a) chip microphotography and (b) its block organization diagram.

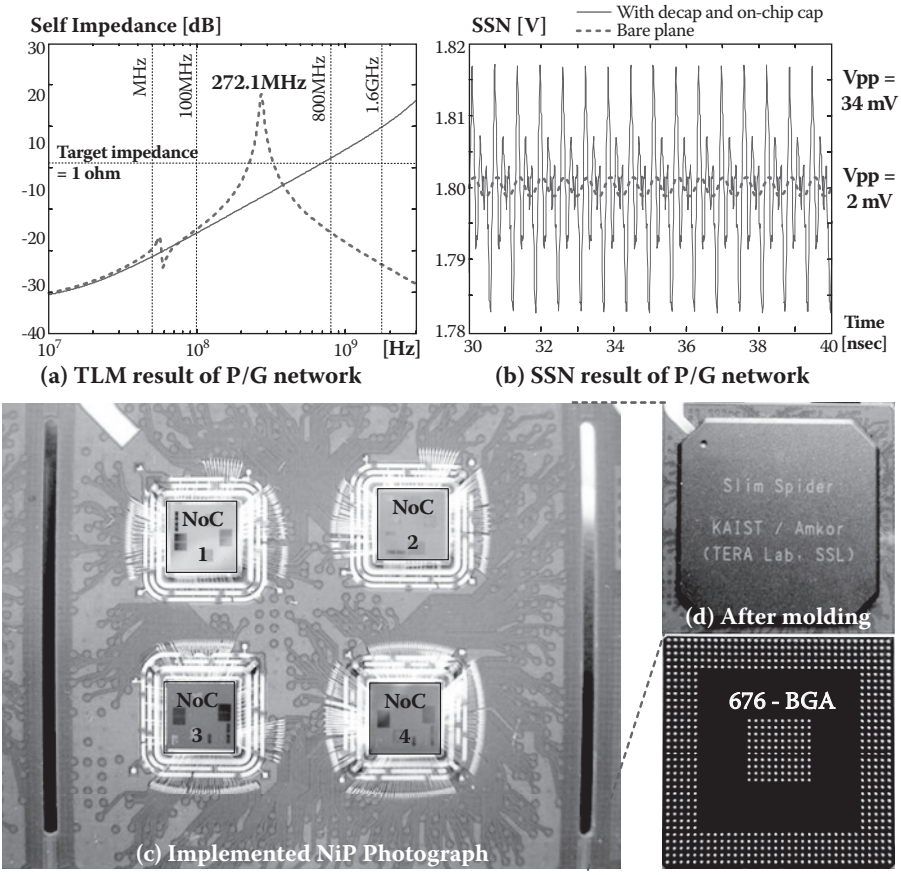


FIGURE 8.8 NiP simulation (TLM, SSN) and NiP photograph.

plane shows inductance characteristics and exceeds the target impedance at 800 MHz and 1.6 GHz. After the insertion of the decoupling capacitance, the impedance resonance occurs at 272.1 MHz, which comes from the L of the bare plane and C of the inserted capacitors. As a result, the self-impedance at the target frequency becomes lower than 1 Ω. Figure 8.8b shows the SSN analysis results when 1.6 GHz input signals are input. The switching noise is reduced from 34 mV to 2 mV because of the decoupling-capacitor insertion on the package and on the die. Ground lines are inserted between high-frequency signal lines to eliminate crosstalk.

Figure 8.9a shows the measured packet signals on the network at FAST mode and Figure 8.9b shows the measured packets with and without the proposed SILENT coding, respectively, while a 3-D graphics application is running on the system [14]. Transitions on a channel are reduced from 134 to 79 after the coding.

A demonstration system is developed with the implemented NiP for multimedia applications. Figure 8.10a shows the demonstration system, which consists of a NiP board on the top layer, video board on the bottom layer, and an LCD panel module. The system demonstrates image processing and animation processing on the LCD

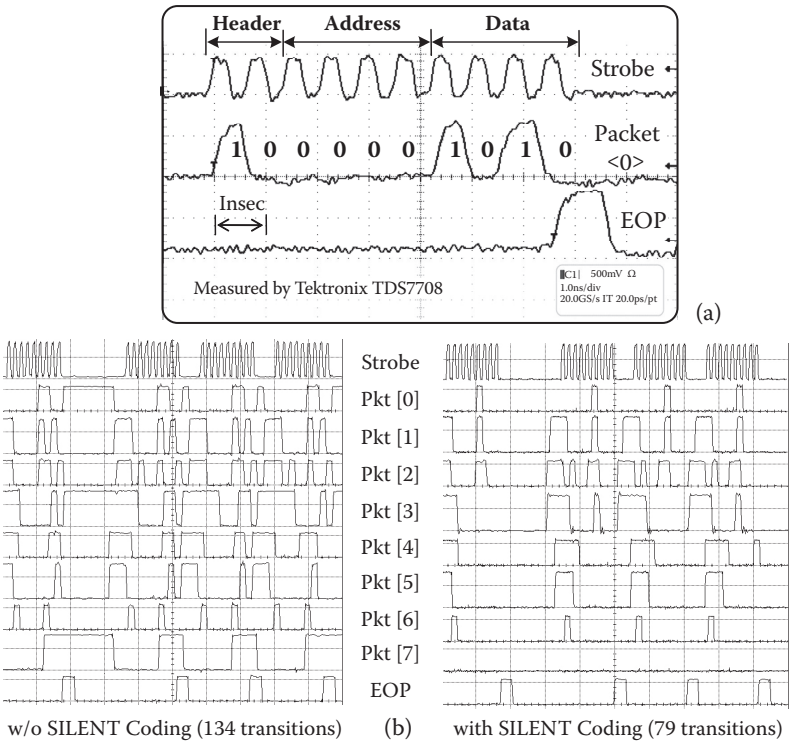
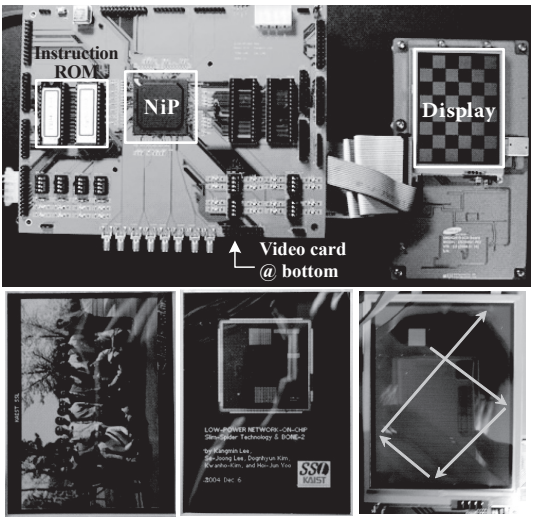


FIGURE 8.9 Measured packet signals and the effect of SILENT coding.

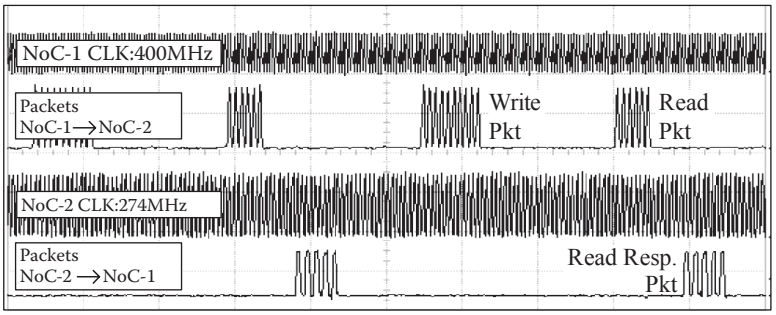
through networks on the chip and in the package. Figure 8.10b shows packet transactions between two NoCs on the NiP where the two NoCs are running at different clock frequencies, e.g., 400 and 274 MHz.

8.2.2.5 BONE 2 Chip Summary

A low-power packet-switched NoC and a NiP with hierarchical star topology are designed and implemented for a high-performance SoC platform. The chip contains two RISC processors for multiprocessor emulation, two 64 kb SRAMs, one on-chip FPGA, one OGW for off-chip network interface, three 4 kb SRAMs for peripheral logic emulation, 1.6 GHz PLL for internal clock generation, and OCNs connecting these processing units. The OCN channel is serialized from 80 bits onto 8 bits to reduce the network area significantly. Source-synchronous signaling enables plesiochronous communications between PUs running at different clock frequencies. Low-power consumption is achieved by applying various techniques such as a lower-swing signaling link, crossbar partial activation, low-energy serial-link coding, and clock-frequency scaling. The chip integrates 2.5 million transistors and consumes less than 160 mW, and the OCN consumes less than 51 mW, delivering 11.2 Gbps aggregated network bandwidth. The 5×5 mm^2 chip is fabricated with the 0.18 μm CMOS process and successfully measured and demonstrated on a NiP system evaluation board running real-time multimedia applications.



(a) Demonstration System



(b) Network in Package Signals between two NoC dies

FIGURE 8.10 Demonstration system: Board, Snapshots, and NiP signals.

8.2.3 BONE 3 (Intelligent Interconnect System)

The uniqueness of BONE 3 comes from the adaptive control schemes for high-speed flexible OCN design. Basically, it utilizes the wave-front-train (WAFT) scheme for high-speed serialization [9]. To stabilize the WAFT operation against power supply voltage variation, an adaptive reference-voltage generation according to the supply voltage variation is realized. Programmable delay synchronizers are used for run-time calibration of phase difference in mesochronous communication links. Adaptive bandwidth control schemes are also adopted for effective energy reduction in the global link wires. Figure 8.11 shows the BONE 3 microphotograph.

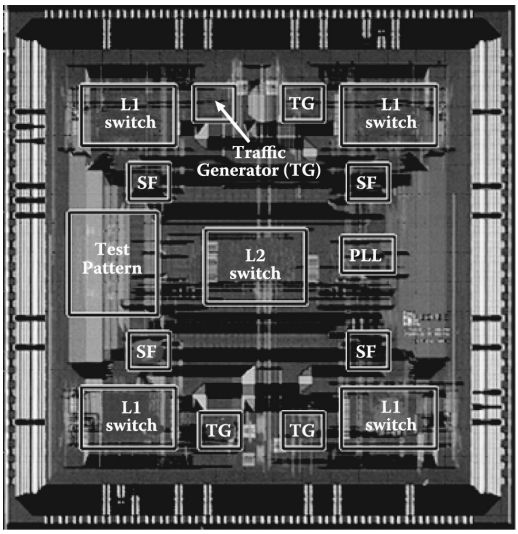


FIGURE 8.11 Die photograph of BONE 3.

8.2.3.1 Supply-Voltage-Dependent Reference Voltage

The supply voltage difference between a sender and a receiver can cause unit delay time difference, so jitter at the receiver can occur. To resolve this problem, a delay element (DE) with a current-starving inverter chain and a reference voltage generator are adopted for supply-voltage-independent delay time.

For supply-voltage-independent delay, the bias current of the current-starving inverter is controlled adaptively according the values of the supply voltage. Using this scheme, the receiver jitter is reduced from 30% to 11% for a 10% supply voltage variation.

8.2.3.2 Self-Calibrating Phase Difference

For mesochronous communication, many works used FIFO synchronizers or single-pipeline synchronizers. However, all these solutions suffer from considerable additional power consumption or area. Otherwise, the success or failure of the synchronization is random.

Instead of using such passive schemes, BONE 3 performs self-calibration for the skew between two clock domains so that it can actively adjust the phases of input signals. As shown in Section 5.3.6, a variable delay (VD) is connected with a simple pipeline synchronizer, and the VD is controlled according to the network condition. The appropriate VD setting for a certain circumstance is obtained through the calibration process. When the NoC turns on, it starts initialization, which includes the self-calibration routine. When the NoC changes its configuration and the skew between the clock domains changes, the NoC also changes the VD setting according to the NoC configuration or environment.

8.2.3.3 Adaptive-Link Bandwidth Control

The bandwidth of a link is a function of the value of the supply voltage. When a network transfers packet signals through a link, the output bandwidth of a transmitter (B_{OUT}) must be lower than the maximum bandwidth of the link (B_{MAX}). In other words, B_{MAX} has to be slightly higher than B_{OUT}. Because B_{OUT} varies according to the traffic demands of the PUs, B_{MAX} is set to the highest B_{OUT} in conventional NoCs. The BONE 3 NoC, however, adaptively controls the supply voltage of a link to lower the B_{MAX} to the current requirement of B_{OUT}. Therefore, the power consumption in the links is effectively reduced. The NoC link uses two kinds of supply voltages: 1.2 and 1.8 V. In the normal mode, a serializer and a deserializer use 1.2 V while supporting the maximum bandwidth of 1.1 Gbps. At this mode, the data rate is bounded to 0.8 Gbps. When the traffic is congested, the link changes the supply voltage to 1.8V, so the link supports the maximum bandwidth of 1.9 Gbps. In this mode, the data rate is bounded to 1.6 Gbps. In the normal mode, the power consumption is reduced by 57% compared to the enhanced bandwidth mode.

8.2.4 BONE 4 Flexible On-Chip Network (FONE)

8.2.4.1 NoC Evaluation Platform

NoC involves a complex design process such as the selection of suitable topology, switch parameters, and communication protocol. Evaluation and optimization of such design parameters are essential for the efficient design of an application-specific SoC. An efficient NoC emulation platform is required to verify, evaluate, and optimize a variety of application-specific NoC solutions. Moreover, an FPGA-based emulation platform gives opportunities to offer and to test a sufficient range of choices of NoC design parameters as well as IPs for various applications with a very fast execution time.

The NoC evaluation board is implemented on three Altera Stratix EP1S60 series FPGAs to explore and evaluate a wide range of NoC solutions, as shown in Figure 8.12. The implemented system has various IPs: two masters (RISC CPU and

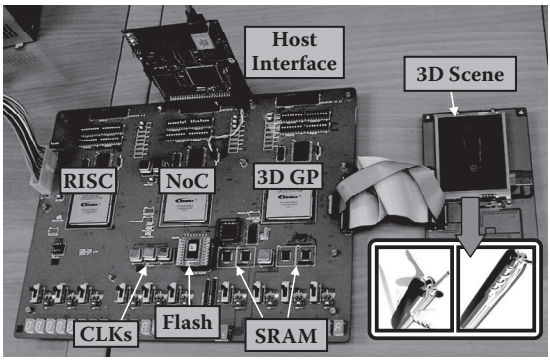


FIGURE 8.12 Implemented NoC evaluation board.

LCD controller) and four slaves (3-D graphics processor, SRAM, Flash, and UART). The integrated NoC uses OCS techniques to reduce the network area significantly and also supports plesiochronous communications among IPs for globally asynchronous locally synchronous (GALS) systems. All implemented modules are designed in RTL level, and six different clocks are used for the GALS operation.

The operation of this system is as follows: The 3-D graphics processor (3D-GP) is initialized by the RISC, and then its instructions are fetched from the SRAM. After rendering the calculation of the 3D-GP, the 3-D scenes are stored in the SRAM. All the transactions between the 3D-GP and SRAM are conducted by the RISC because the 3D-GP is designed as a slave IP. The LCD controller continuously reads the SRAM frame memory and displays the 3D scenes on an LCD screen. As a result, 3D graphics applications are demonstrated on the NoC evaluation board, which shows the feasibility of NoC in the implementation of a real system.

8.2.4.2 NoC Run-Time Traffic-Monitoring System

The NoC evaluation board contains a run-time traffic-monitoring system for the accurate evaluation and optimization of an application-specific NoC. Three traffic parameters—(1) an end-to-end latency of each transaction, (2) a backlog in each queuing buffer, and (3) output-conflict rate at each switch output port—are traced at run-time as the network performance indicators (see Figure 8.13a). In addition, a wide range of dynamic statistics such as a communication bandwidth between integrated IPs, an evaluation time of the application, and link/switch/buffer utilization can also be measured.

The main requirements of NoC traffic monitoring are nonintrusiveness, scalability, real-time tracing, and low cost. Figure 8.13b shows an example of how a NoC traffic-monitoring system meets the requirements. It consists of three subsystems: host interface, central controller, and traffic probes. The host interface, a bridge between the central controller and a host PC, transfers the traffic-monitoring results to the host PC via an Ethernet line. The central controller enables/disables each traffic probe based on the requested monitoring regional scope and time interval. A traffic probe is connected to a switch or a network interface to trace the real-time traffic parameters such as an end-to-end latency, a queuing buffer usage, and an output conflict rate on a switch. Then, the traffic traces are stored in its local trace memory, which is accessible to the host interface's ARM via the central controller. Because all the monitoring processes do not have any influence on the NoC packet flow, nonintrusive probing is achieved.

The traffic monitoring system has a modular architecture. Thus, a traffic probe can be attached to any NoC component, which is a design-time choice. Because the internal traffic behavior of the NoC core can be measured and analyzed through the monitoring system, the potential bottleneck of the NoC can be examined in the early stages of NoC design and, also, its design can be optimized by the suitable selection of various design parameters, such as queuing buffer size, packet priority assignment, and IP mapping. It will provide energy-efficient and performance-optimized communication structures for a large range of applications.

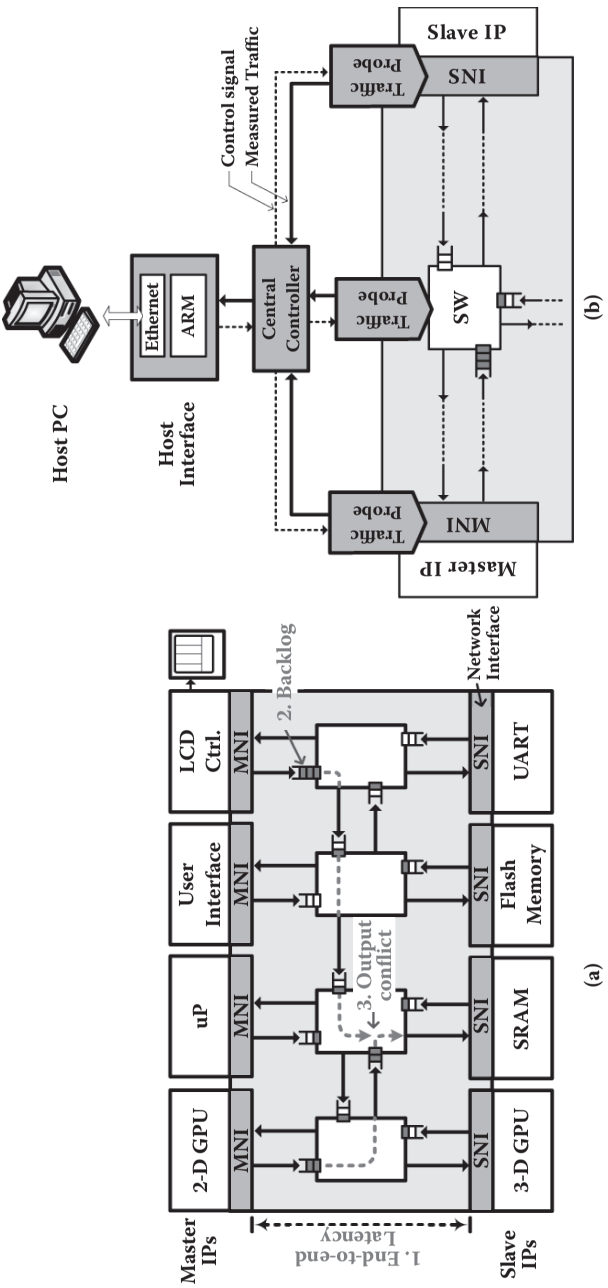


FIGURE 8.13 (a) Measurable traffic parameters and (b) NoC traffic-monitoring system.

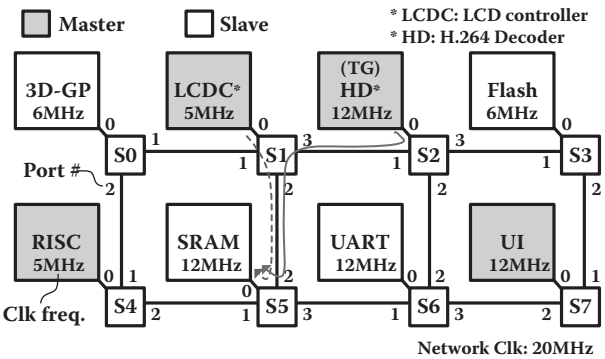


FIGURE 8.14 Portable multimedia system in mesh topology.

8.2.4.3 Case Study: Portable Multimedia System

8.2.4.3.1 Target System Description

In this section, a case study in NoC evaluation and optimization is described to demonstrate the effectiveness of the NoC evaluation framework. A portable multimedia system is implemented on the NoC evaluation board with a H.264 decoder (HD) traffic generator, as shown in Figure 8.14. The traffic generator produces real traces, assuming that it decodes CIF (352*288) H.264 baseline profile at level 2 with 30 frames/s. The SRAM is used as a display frame buffer for HD and the 3D-GP. The LCD controller directly reads the frame data from the SRAM with burst operation (burst length = 8) continuously. This transaction has a hard real-time requirement to guarantee the display frame rate. The user interface generates burst packets with burst length of 16b, once per 10,000 cycles.

The HD decodes the encoded video stream that is already downloaded in the Flash memory. During the decoding process, the HD accesses the frame data in the SRAM with a bandwidth of 10 Mbps. After the decoding process is completed, the HD writes the decoded 2-D scenes into the SRAM frame memory with 145 Mbps bandwidth.

8.2.4.3.2 NoC Evaluation

Figure 8.15 shows the latency distribution of three selected flows. The first two traffic flows from the HD and LCD controller to the SRAM experience relatively long latency and also large variations. After the decoding process of the HD, the traffic from the HD to the SRAM increases abruptly. Thus, the flow from the LCD controller to the SRAM is also affected seriously because the two flows share a link between SW1 and SW5. Meanwhile, the third traffic flow from the 3D-GP to the RISC shows a smaller and more constant latency because it does not share the network resource with others.

Figure 8.16a shows the backlog distribution on the three flows. Bursty traffic from the HD and the LCD controller to the SRAM cause the SW5 input#2 queue to be in the full state. On the contrary, the backlog of SW4 input#1 queue is almost 0 or 1.

Figure 8.16b presents the output conflict counts/1,000 cycles on the congested link between SW1 and SW5. After the decoding process is completed in the HD,

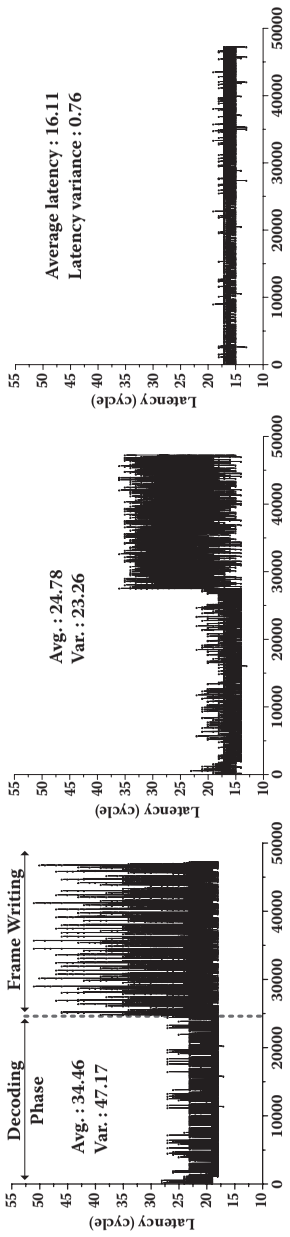


FIGURE 8.15 Latency distribution, average latency, and its variations.

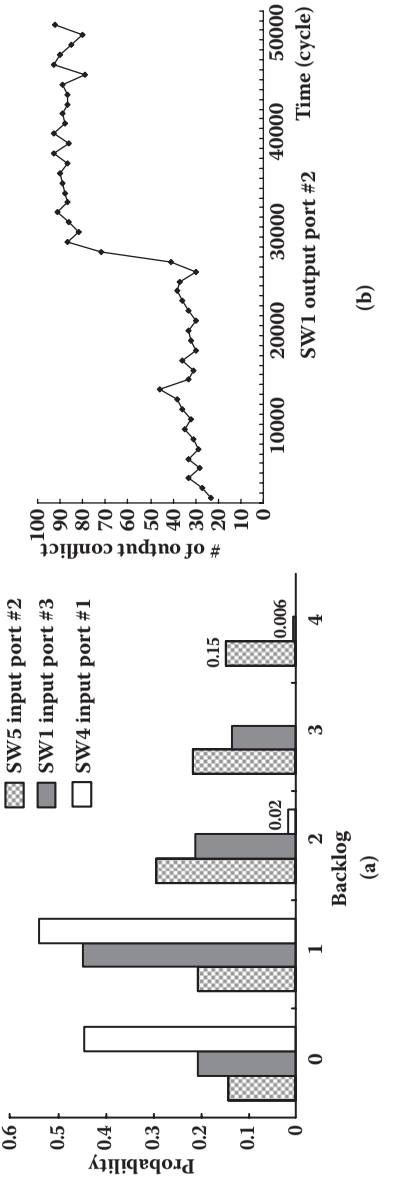


FIGURE 8.16 (a) Backlog distribution and (b) output conflict status.

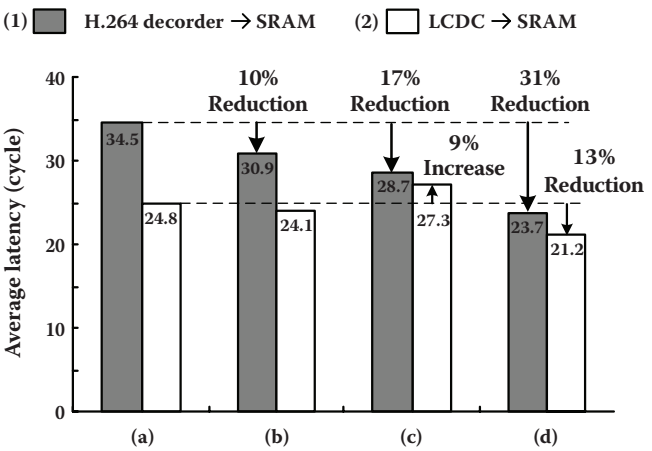
the output conflict on the shared link increases rapidly (from the 25,000th cycle) and remains highly congested at around 90 conflicts/1000 cycles.

8.2.4.3.3 NoC Optimization

In this section, the target system is optimized in three ways: buffer size optimization, packet priority assignment, and topology remapping.

The input queuing buffers in a switch take a significant portion of the chip area of the NoC; so their size should be minimized without significant performance degradation. The initial NoC design has all input buffers of 4-packet capacity uniformly. Although using the same buffer size for all the input buffers is straightforward and widely used in current NoC designs, it may lead to excessive use of silicon area or poor performance. Based on the backlog monitoring results, the optimum buffer capacity of each input queue can be obtained. After the buffer size optimization, 36% of total buffer size is reduced (Figure 8.17a and 8.17b). In addition, 10% latency reduction and 17% latency-variance reduction are also obtained on the congested flow.

Each packet has a priority field in its header. When more than two packets are destined for the same output port in a switch—an occurrence of an output conflict—a packet with higher priority gets a grant to the output port. Therefore, higher packet priority can be assigned to the latency-critical flow for NoC optimization. To reduce the latency of the most critical flow (from HD to SRAM) in our application, a higher priority is given to the packets generated by the HD right after the decoding process. Figure 8.17c shows that 17% average latency reduction is obtained and its variance is also diminished significantly.



	(a)	(b)	(c)	(d)
(1)	47.17	39.27	27.73	23.48
(2)	23.26	23.17	29.17	24.62

< Latency Variance >

(a) : Before Opt.
(b) : Buffer Size Opt.
(c) : Priority Modification
(d) : IP Mapping Opt.

FIGURE 8.17 Average latency and its variance after NoC optimization.

If the position of the HD is interchanged with the UART in a given mesh topology, the HD and the LCD controller do not share a link anymore. Moreover, the hop count between the HD and the SRAM is also minimized. As a result, the average latency of the HD and the LCD controller flows are reduced by 31% and 13%, respectively, as shown in Figure 8.17d.

8.2.4.4 FONE Platform Summary

The NoC emulation board is implemented on FPGAs to evaluate and optimize a variety of application-specific NoC designs. It provides dynamic network status such as backlog, output conflict on a switch, and end-to-end communication latency for each packet flow, using a traffic-monitoring system. A portable multimedia system is implemented to demonstrate the effectiveness of the NoC evaluation framework. The target system is evaluated and optimized in three ways: buffer size optimization, packet priority assignment, and topology remapping. As a result, buffering cost and latency reduction are obtained up to 36% and 31%, respectively.

8.2.5 BONE V1: Vision Application-1

8.2.5.1 Introduction

This section describes memory-centric NoC (MC-NoC), which facilitates flexible and traffic-insensitive mapping of tasks on the homogeneous multiprocessor SoC (MP-SoC) [15]. The MC-NoC features the hierarchical star topology network and memory management scheme, which supports unidirectional interprocessor communication. The MC-NoC incorporates distributed and fine-grained shared memory for simultaneous data transactions among processing elements (PEs), whereas a hierarchical star topology network is adopted for providing PEs with area-efficient external memory interconnections. The MC-NoC improves feasibility and flexibility in mapping a series of tasks into the homogeneous MP-SoC. The details of the MC-NoC are explained in the following subsections.

8.2.5.2 Architecture and Operation

Figure 8.18 shows the architecture and operation of the MC-NoC. MC-NoC is applied to the homogeneous MP-SoC, which incorporates 10 RISC processors. The building blocks of the MC-NoC are the dual-port SRAM, crossbar switch, NI, and channel controller. In the MC-NoC, dual-port SRAMs are dynamically assigned to the subset of the RISC processors involved in data communication. Then, shared data are exchanged by accessing the assigned dual-port SRAM. Crossbar switches of the MC-NoC provide nonblocking concurrent interconnections between dual-port SRAMs and RISC processors. The operational frequency of the crossbar switches is decided to be twice the other part of the MC-NoC to reduce the overhead of packet-switching latency. The NI performs packet processing and clock synchronization between the crossbar switch and other building blocks of the MC-NoC. The key building block is the channel controller. The channel controller automatically manages communication channels between RISC processors to facilitate mapping

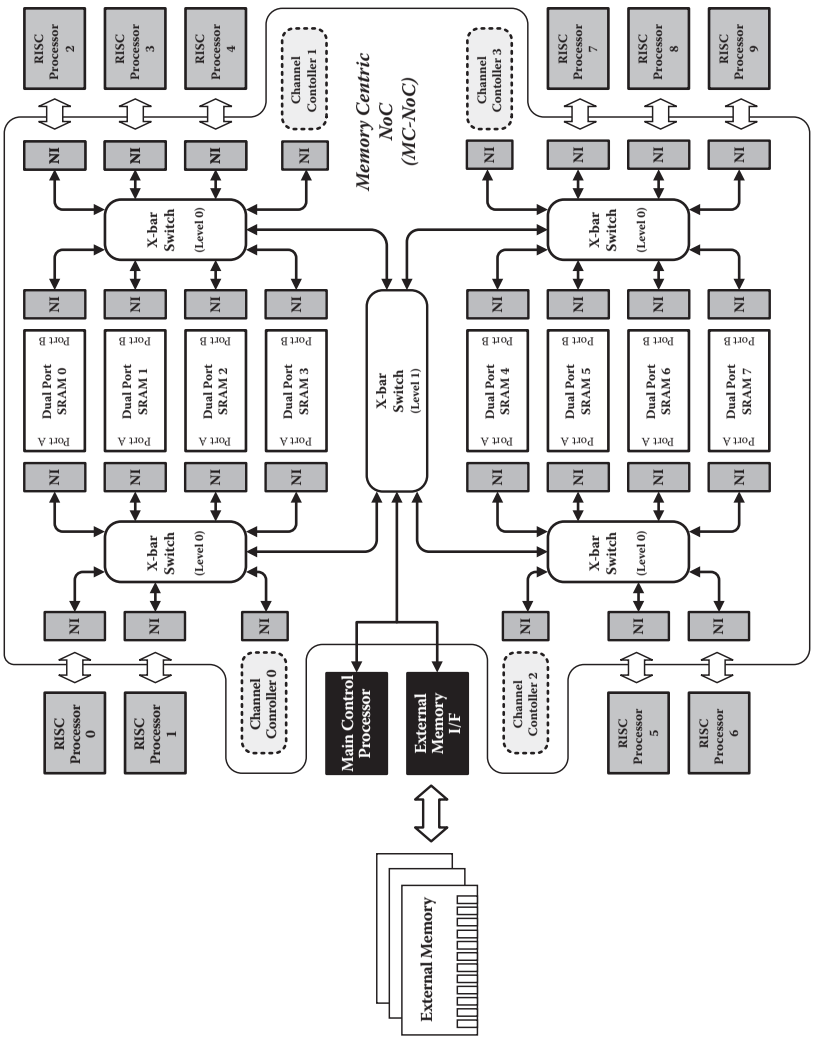


FIGURE 8.18 Overall architecture of the MC-NoC.

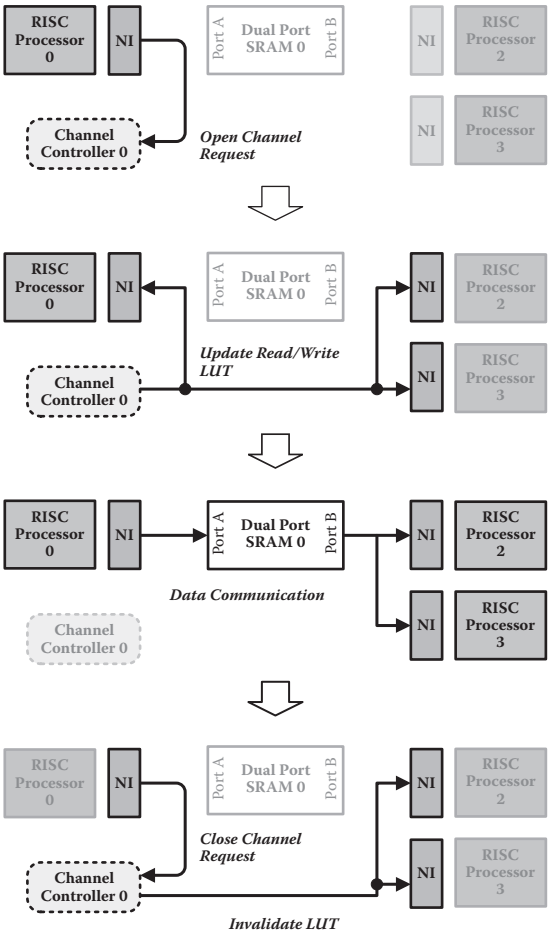


FIGURE 8.19 Important steps of the MC-NoC operation.

of the task on the homogeneous SoC. The role of channel controller is described in more detail with the operation of the MC-NoC.

Figure 8.19 briefly represents important steps of the MC-NoC operation. In this figure, crossbar switches are not drawn for simplicity of the description. Although the operation is explained, we will assume that RISC processor 0 wants to pass the processed results into RISC processors 2 and 3. The MC-NoC operation is initiated by RISC processor 0 sending an Open Channel request to the channel controller. The information about source and destination RISC processors is also included in the Open Channel request. After that, the channel controller assigns one dual-port SRAM as a data communication channel if any of the SRAMs are available. By updating the routing look up tables (LUTs) in the NIs of corresponding processors, the SRAM assignment is completed. In this way, the assigned SRAM is made accessible only to the RISC processors involved in data communication. At the end of data transfer through the dual-port SRAM, the source RISC processors send a Close

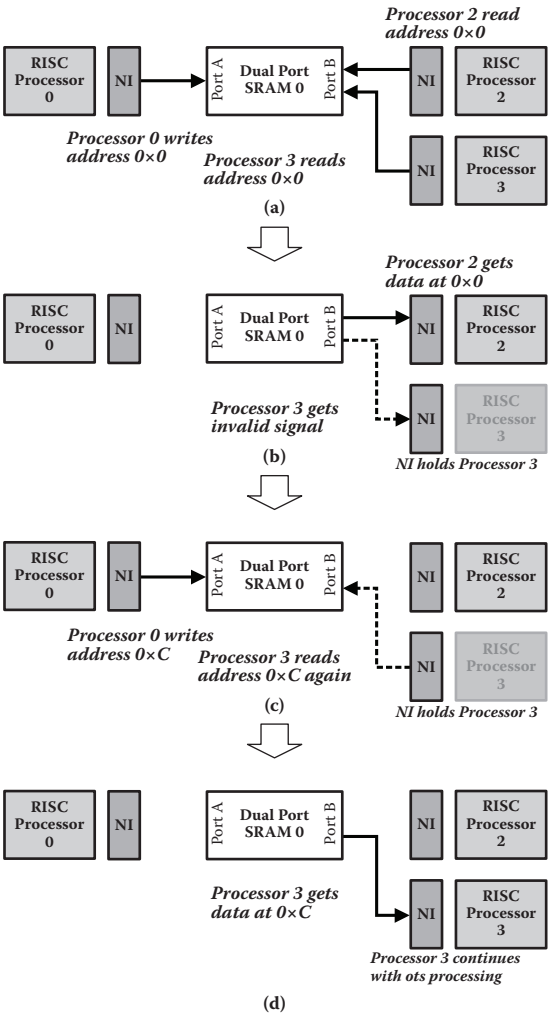


FIGURE 8.20 Data synchronization scheme of the MC-NoC.

Channel request to the channel controller. Then, the channel controller invalidates the updated LUTs after checking for completion of data transfer. In the proposed MC-NoC, each processor is able to send as many Open Channel requests as required. If all the SRAMs are used by other processors, the data transfer should be stalled until one of the SRAMs becomes available. In the MC-NoC, Open/Close Channel requests and LUT updates are performed by sending special packets that are not visible to any processor or memory. Controlling MC-NoC operations using special packets has the advantage of removing additional control signal wires.

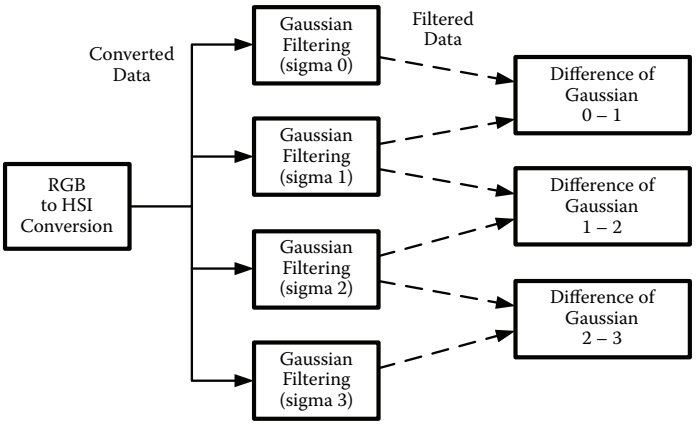
Although data communication is performed through the dual-port SRAM that is assigned by the channel controller, progress of data access from the destination processors may differ from one another. To improve programming feasibility of the multiple RISC processors, the MC-NoC provides a data synchronization scheme

to resolve consistency problems that occur because of the different data access sequences of the destination processors. Figure 8.20 illustrates this. Processor 2 reads data from address 0 × 0 while Processor 3 accesses address 0 × C. Until this moment, Processor 0 has written valid data only at the address 0 × 0. The next step is shown in Figure 8.20b. Here, only Processor 2 gets valid data from the dual-port SRAM, and Processor 3 receives the invalidate signal from the valid check logic inside the dual-port SRAM. After that, the NI of Processor 3 holds the processor and retries read after specified wait cycles, as shown in Figure 8.20c. Once Processor 0 writes a valid data at address 0 × C, Processor 3 also gets valid data and continues processing (Figure 8.20d). In the MC-NoC operation, the retry procedure described in Figure 8.20c is transparent to the RISC processors because the NI module of the MC-NoC automatically manages the procedure. Similar to the Open/Close Channel requests, the invalid signal from the valid check logic of the dual-port SRAM is also transferred as a special packet. In our implementation, the valid check logic takes 5% of the dual-port SRAM area.

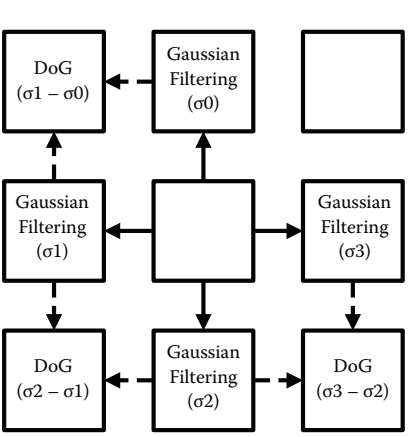
8.2.5.3 Benefits of the MC-NoC

The main advantage of the MC-NoC is its flexibility of task mapping on the homogeneous SoC. In this section, the benefits of the MC-NoC are discussed through a comparison with the conventional 2-D mesh topology NoC. As a task-mapping example, the edge detection operation is shown in Figure 8.21a [15]. In the figure, rectangular boxes represent processors performing tasks, and solid/dotted lines depict dataflow between tasks. In this operation, the input image is first converted from the RGB color space to HSI color space. The converted image is processed by Gaussian filters with varying coefficients (sigma), and subtractions between filtered results are calculated to detect edges in different scales. Figure 8.21a, Figure 8.21b and 8.21c show the mapping of the edge detection operation on the homogeneous SoC with a conventional 2-D mesh NoC. Because, there is no contention in dataflow for the task mapping shown in Figure 8.21b, it will outperform task the mapping shown in Figure 8.21c even though all other conditions are equally given. The contention of dataflow in Figure 8.21c is visualized by the number of arrows in the same locations. The drawback of the conventional 2-D mesh NoC is the dependence of overall SoC performance on the mapping of the task. Further, finding the optimal task mapping may be very difficult for applications with complex data dependencies. Longer average hop counts and possibilities of deadlock in finding the bandwidth-optimized task mapping are additional drawbacks of the 2-D mesh NoC.

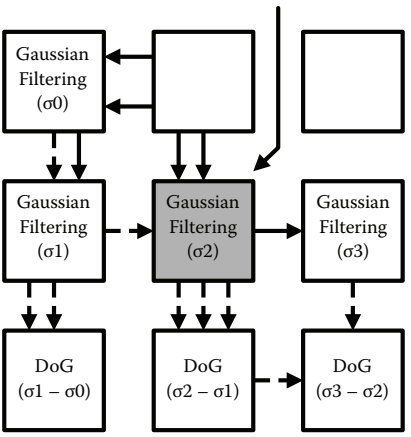
The feasibility of task mapping on the MC-NoC is shown in Figure 8.22. For simplicity, crossbar switches are not drawn, and only a portion of the MC-NoC is depicted in this figure. In the MC-NoC, processors and dual-port SRAMs are interconnected through the crossbar switches, which provide full nonblocking connections. Therefore, interchanging task mappings just inside the left side or right side of the MC-NoC does not affect the dataflow characteristic and resulting overall performance. For example, interchanging the task mapping of the difference of Gaussian (DoG) 0–1 and RGB-to-HSI conversion in Figure 8.22 has no impact on the contentions in dataflow. This attribute of the MC-NoC improves the flexibility of task



(a) Edge Detection Operation Flow



(b) Task Mapping on 2D Mesh NoC w/o Contention



(c) Contended Task Mapping on 2D Mesh NoC

FIGURE 8.21 Mapping of edge detection task into conventional Mesh NoC.

mapping on the homogeneous SoC, because the key decision of the task mapping reduces to whether the given task is mapped on the left or the right side. The dual-port SRAM is adopted to remove performance loss when SRAM accesses come from both the left and the right sides of the MC-NoC.

In addition, variations on the required bandwidth between coworking tasks are also successfully supported by the MC-NoC. If a large bandwidth is required for some tasks, multiple numbers of SRAMs can be dynamically assigned for the demanding task. For small amounts of data transfer, only one dual-port SRAM is assigned. In regard to traffic characteristic, the MC-NoC improves locality, because most of the packet transactions between processors and memories are confined into a single crossbar switch to which the involved processors and SRAMs are connected.

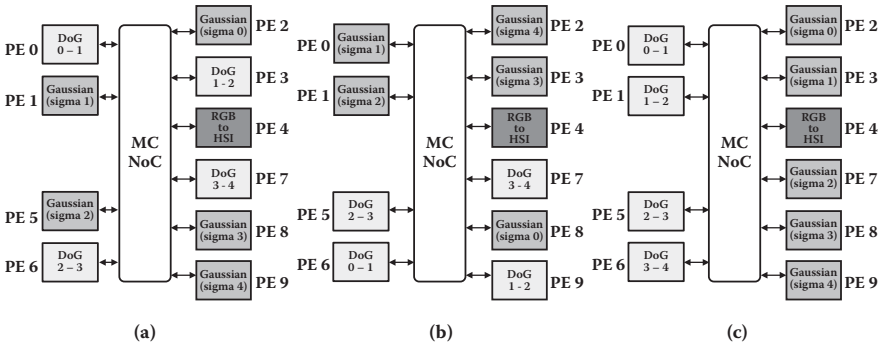


FIGURE 8.22 Task mapping of edge detection on the MC-NoC.

8.2.5.4 Evaluation of the MC-NoC

To demonstrate feasibility and flexibility of task mapping on the MC-NoC, this section briefly reports experimental results, showing how the overall performance is affected by different task mappings on the MC-NoC. For comparison, tasks of edge detection operations are mapped into the homogeneous SoC as shown in Figure 8.18.

The different mapping configurations are depicted in Figure 8.22. Although the MC-NoC is drawn as a single rectangular black box for simplicity, the architecture shown in Figure 8.22a is the one that is applied for the performance comparison. At first, the RGB-to-HSI conversion, Gaussian filter operations, and DoG calculation tasks are mapped randomly on the given architecture (Figure 8.22a). In the second, Gaussian filter operations are mapped on the upper half of the SoC, whereas DoG calculations are mapped on the lower half (Figure 8.22b). Similarly, the DoG and Gaussian filtering tasks are separated into left and right, respectively, in the third task-mapping configuration (Figure 8.23b). The results of performance comparison are given in Table 8.2. In performance comparison, verilog HDL description is used for the MC-NoC and other parts of the SoC. Therefore, the performance comparison result derived from the simulation is cycle-accurate. In the table, the numbers in the "cycle count" column give the required clock cycles to perform edge detection for 320 × 240 pixels of image. The number in the rightmost column shows the cycle-count ratio compared to the task-mapping configuration in Figure 8.23a. The results prove the flexibility of task mapping on the proposed MC-NoC. In the MC-NoC-based homogeneous SoC, the difference in overall performance according to the various task mappings is less than 3%. This feature of the MC-NoC also facilitates software-level optimization of the NoC.

8.3 INDUSTRIAL IMPLEMENTATIONS

8.3.1 Intel's Tera-FLOP 80-Core NoC

Intel Corporation launched a terascale computing research program a few years ago to handle tomorrow's advanced applications, which would need a thousand times

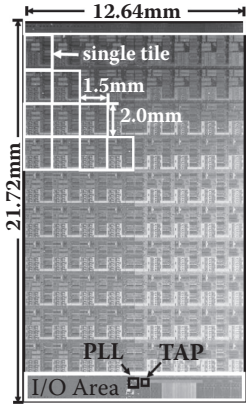


FIGURE 8.23 Intel's Tera-FLOP 80-core NoC chip.

TABLE 8.2
Performance Comparisons of Different Task Mappings on the MC-NoC

Task Mapping	Cycle Count	Ratio to Mapping (a)
(a)	8,202,900	1
(b)	8,086,580	0.986
(c)	8,021,820	0.978

more compute capability than is available in today's giga-scale devices. For example, there is real-time data mining across the teraflops of data; artificial intelligence (AI) for smarter cars and appliances; and virtual reality (VR) for modeling, visualization, physics simulation, and medical training [17].

The terascale research program consists of three research categories—teraflop of performance, terabytes per second of memory bandwidth, and terabits per second of I/O performance [17]. In this chapter, we focus on the "teraflop of performance" research relating to NoC development and implementation (see Figure 8.23). Eighty processing cores are interconnected through a 2D mesh packet-switched OCN. It performs up to 1 teraflop at 4 GHz clock speed and consumes less than 100 W [18].

8.3.1.1 Key Enablers for Tera-FLOP on a Chip [18]

- 80 Processing elements (PE), 160 single-precision floating point units (FPUs) designed for 4GHz operation
- Fast single-cycle accumulate loop
- Sustained FPU throughput: 2 FLOPS/cycle
- 80 Gbps router, operating at 4 GHz
- Shared and double-pumped crossbar switch
- 2-D mesh topology, 256 Gbps bisection bandwidth
- A 15 FO4 balanced core and router pipeline
- Robust, scalable mesochronous clock distribution
- 65 nm eight-metal CMOS

8.3.1.2 NoC Architecture Overview [18]

The NoC architecture contains 80 tiles arranged as a 10 × 8 2-D mesh network and operating at 4 GHz (see Figure 8.24). Each tile consists of a processing engine (PE) connected to a 5-port router with mesochronous interfaces, which forwards packets between tiles. The 80-tile OCN enables a bisection bandwidth of 256 Gbps. The PE contains two independent fully pipelined, single-precision, floating-point

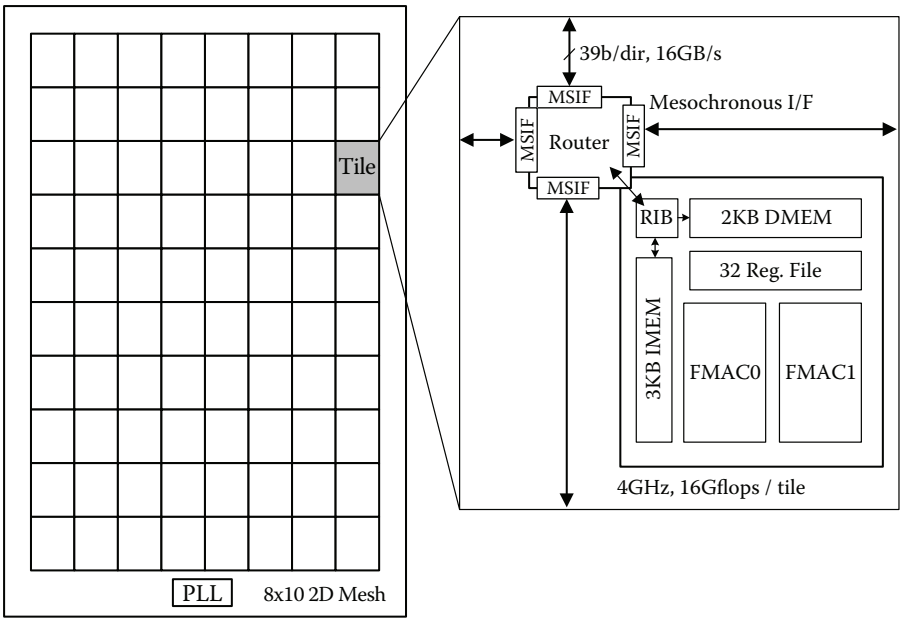


FIGURE 8.24 Chip architecture.

multiply-accumulator (FPMAC) units with 3 KB single-cycle instruction memory (IMEM) and 2 KB data memory (DMEM). A 96-bit VLIW encodes up to eight operations per cycle. With a 10-port (6-read, 4-write) register file, the architecture allows scheduling to both FPMACs, simultaneous DMEM load and stores, packet send/receive from mesh network, program control, and dynamic sleep instructions. A router interface block (RIB) handles packet encapsulation between the PE and router. The fully symmetric architecture allows any PE to send (receive) instruction and data packets to (from) any other tile.

The 4 GHz 5-port wormhole-switched router uses two logical lanes: virtual channels for deadlock-free routing, and a fully nonblocking crossbar switch with a total bandwidth of 80 Gbps. Each lane has a 16 FLIT (FLow control unIT) queue, arbiter, and flow control logic. The router uses a five-stage pipeline with a two-stage round-robin arbitration scheme that first binds an input port to an output port in each lane and then selects a pending FLIT from one of the two lanes.

Figure 8.25 shows a NoC packet format. Each packet is subdivided into multiple FLITs. Each packet has minimum two FLITs, and there is no maximum size limit. Each FLIT consists of a 6-bit control field and a 32-bit data field. The control field includes two flow control bits for each lane, a valid indication bit for the FLIT, and packet header/tail indication bits. There are three kinds of FLITs—header FLIT, PE control FLIT, and data FLIT. The header FLIT has a 3-bit destination ID (DID), which represents the out-port direction on each switching hop. Because of the data field size limit, the maximum hop count is limited to 10 hops. However, a chained header seems to support larger hop counts. The PE control FLIT includes an address filed and PE control information such as PE power management signals [18].

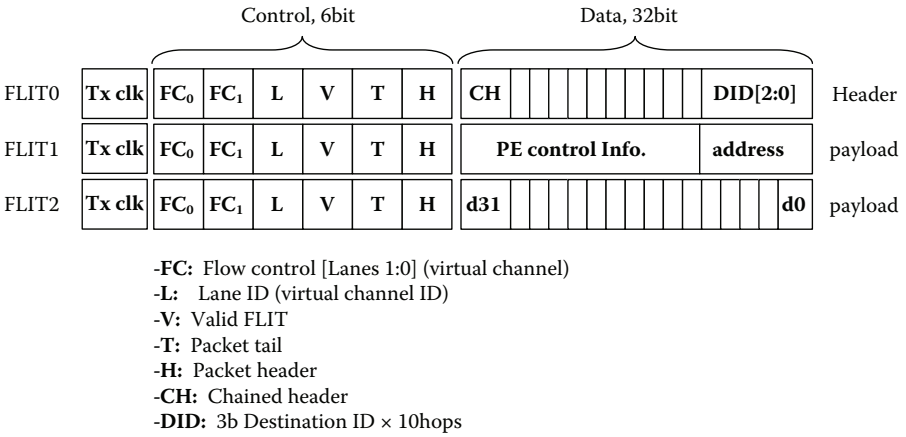


FIGURE 8.25 Packet format and protocols.

8.3.1.3 Double-Pumped Crossbar Router and Mesochronous Interface

Each 36-bit crossbar data bus is double-pumped at the fourth pipe stage by interleaving alternate data bits using dual edge-triggered flip-flops, reducing crossbar area by 50% [19]. The double-pumped technique is the same as the 2:1 serialization within a switch fabric line, although it maintains the clock frequency by using a double-edge technique similar to a double-data-rate SDRAM (DDR-SDRAM) interface.

The point-to-point router links implement a phase-tolerant mesochronous interface with FIFO-based synchronization. A 4-deep circular FIFO—Push-In Mux-Out Register type in Figure 5.8—captures data using a delayed link strobe at the receiver.

The chip uses scalable global mesochronous clocking, which allows for clock-phase-insensitive communication across the tiles and synchronous operation within each tile, i.e., globally mesochronous and locally synchronous (GMLS) (see Figure 8.26). The on-chip PLL output is routed using horizontal M8 and vertical M7 spines. Each spine consists of differential clocks for low duty-cycle variation along the worst-case clock route of 26 mm. An op-amp at each tile converts the differential clocks to a single-ended clock with 50% duty cycle. The worst-case simulated global duty-cycle variation is 3 ps, and local clock skew within the tile is 4 ps. The systematic clock skews inherent in the distribution help spread clock power because of simultaneous clock switching over the entire cycle. The estimated global clock distribution power at 4 GHz, 1.2 V supply, is 2.2 W.

8.3.1.4 Fine-Grained Power Management

Fine-grained clock gating, sleep transistor, and body bias circuits [20] are used to reduce active and standby leakage power, and are controlled at full-chip, tile-slice, and individual tile levels based on workload. Each tile is partitioned into 21 smaller sleep regions with dynamic control of individual blocks in the PE and router units, based on the instruction type. The router is enabled on a per-port basis, depending on network traffic patterns. The design uses NMOS sleep transistors to reduce the

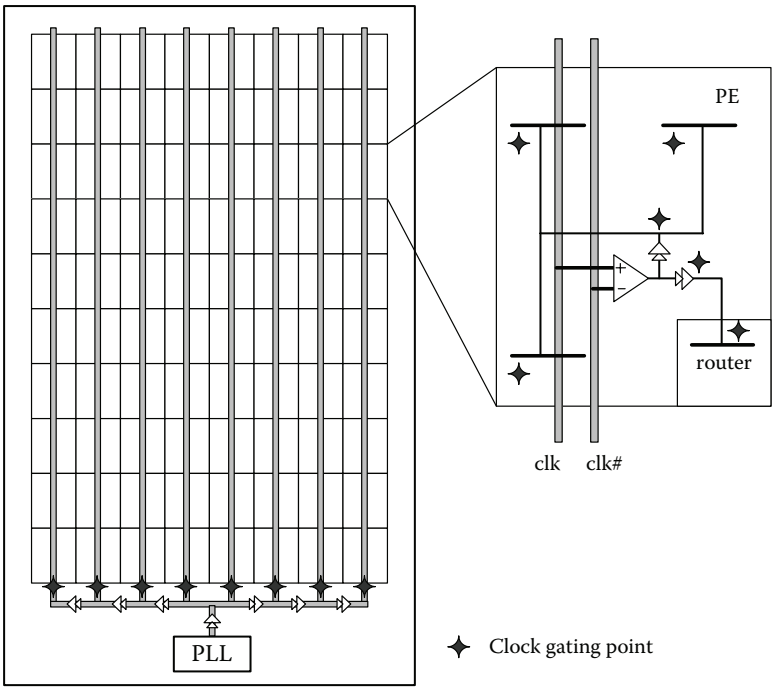


FIGURE 8.26 Clock distribution.

frequency penalty and area overhead. Memory arrays use an active clamped sleep transistor [21] that ensures data retention and minimizes standby leakage power. The average sleep transistor area overhead is 5.4%, with a 4% frequency penalty. About 90% of FPMAC logic and 74% of each PE is sleep enabled. Forward body bias can be applied to NMOS devices during the active mode to increase the operating frequency, and reverse body bias can be applied during the idle mode for further leakage savings. Figure 8.27 shows the power breakdown of a tile and NoC building blocks. As you can see, clocking and buffering are the major power consumers in the NoC. Table 8.3 shows the power and performance summary of the Tera-FLOP 80-Core NoC.

8.3.2 Intel's Scalable Communication Architecture [22]

Intel Corporation initiated another NoC project for the application of wireless communications such as WiFi (IEEE 802.11a,g,n), WiMAX(IEEE 802.16e), Digital TV (DVB-H), 3GPP, and 4G standards [22,23]. This section introduces their research directions and achievements.

8.3.2.1 Scalable Communication Core

Intel proposed a scalable communication core (SCC), which is a power- and area-efficient solution for the physical layer (PHY) and lower MAC processing of concurrent multiple wireless protocols. The architecture consists of coarse-grained,

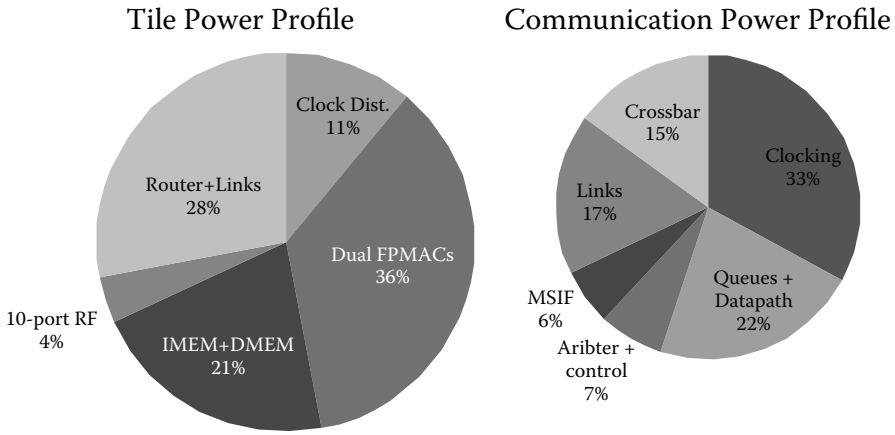


FIGURE 8.27 Power breakdown of a tile and NoC building blocks.

TABLE 8.3
Power and Performance Summary

Frequency	Voltage	Power	Aggregated BW	Performance
3.16 GHz	0.95 V	62 W	1.62 terabits/s	1.01 teraflops
5.1 GHz	1.2 V	175 W	2.61 terabits/s	1.63 teraflops
5.7 GHz	1.35 V	256 W	2.92 terabits/s	1.81 teraflops

heterogeneous, programmable accelerators connected via a packet-based 3-ary 2-cube NoC. The combination of the accelerators, which were developed for key communications operations, and the NoC results in an architecture that is flexible for multiple protocols, extensible for future standards, and scalable to support multiple simultaneous streams.

8.3.2.2 Prototype Architecture

The SCC system architecture, shown in Figure 8.28, consists of a 3-ary 2-cube (3 × 3 2-D mesh) interconnect and a heterogeneous set of coarse-grained, highly optimized, and parameter-reconfigurable baseband processing elements (PEs). This NoC architecture was selected to fulfill system flexibility and scalability constraints.

The SCC interfaces to separate RFIC via a digital front end (DFE). Currently, it is assumed that a different RFIC is required for each standard. The SCC also has network interfaces to protocol-specific MACs. It is also assumed that a separate MAC is required for each standard.

Although much of recent SDR research has employed reconfigurable logic, the resulting cost in area and power far exceeds the constraints of mobile wireless client platforms. Instead, Intel has defined the PEs after analyzing the PHYs of several wireless protocols and determining a common set of kernels [23]. These PEs or accelerators can be configured to achieve the required flexibility using corresponding

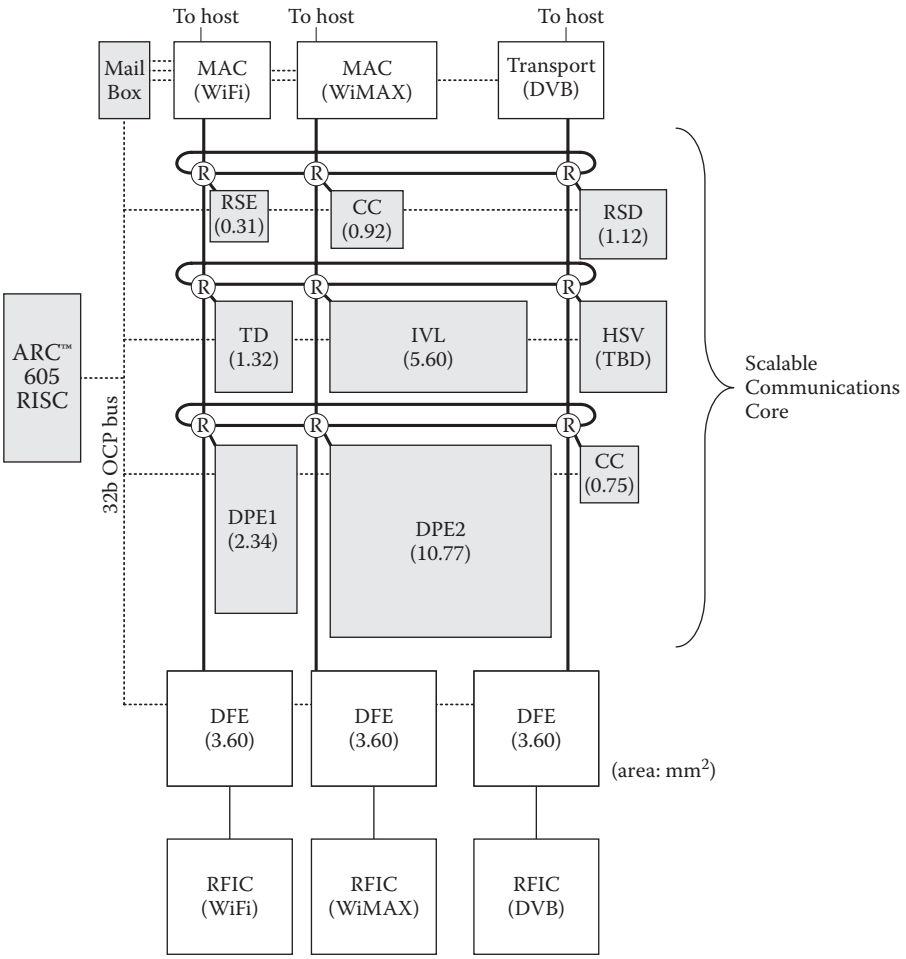


FIGURE 8.28 SCC system architecture.

programming technology that they are developing to enable developers to program, debug, and validate new protocols.

The PEs are connected in a low-latency 3-ary 2-cube that provides for flexible data routing and enables the architecture to scale for future requirements. Each PE injects packets into the network through a router's local/client port via an asynchronous data interface that allows the router and PE to operate at different clocks.

This architecture retains a high degree of parallelism and low-energy characteristics of a traditional ASIC PHY pipeline, while providing complete flexibility in routing data between highly configurable PEs. In addition, the computational resources are time-multiplexed to support multiple streams, and resource context switching is data driven. Finally, memory is distributed and local to the PEs.

Scalability is a key attribute of the architecture. The number and types of the PEs that are instantiated in a particular SCC implementation is dependent on the expected workload. For example, the designer can instantiate the appropriate mix

of PEs that is necessary to meet the computational requirements of the worst-case protocol scenario and attach them via the NoC. For non-worst-case scenarios, the unused PEs are clock-gated to reduce power consumption.

8.3.2.3 Control Plane (OCP-Bus)

The control plane that configures and controls the PEs consists of two components: (1) a bus based on the Open Core Protocol (OCP) [24], which enables the ARC™ processor to manage the PEs, and (2) mailboxes that enable the exchange of primitives between the ARC and the MACs. The control plane is used for very light control and status functions.

8.3.2.4 Data Plane (NoC)

The SCC solution for an efficient and flexible data plane is a regular 3-ary 2-cube directed network that combines a high-efficiency and low-overhead ASIC streaming dataflow with packetization and routing. Although, data may be routed from any source PE to any sink PE, our layout keeps the routing length to a minimum: PE pairs with a high traffic volume are placed adjacent to each other in the mesh.

8.3.2.4.1 Packet Hierarchy

Two packet sizes are defined: (1) *logical packets* that are sized to accommodate natural data blocks, such as an OFDM symbol, interleaver block, or RS codeword and (2) *physical packets* that are sized for low latency and are the atomic routing unit. The physical packets are, typically, small fragments of a logical packet.

In communications applications, logical packets could be very large. For example, the DVB-H standard requires an 8-k point FFT. At 16-bit complex precision, this logical packet is 262,144 bits–32 kB. At a transfer rate of 6.4 Gbps, a route is blocked for 40.96 s. This is an unacceptable latency for an 802.11a stream that is sharing the same interconnect segment and/or PE. Therefore, the 262,144-bit logical packet is divided into 512 × 512 bit physical packets. The worst-case latency due to contention for interconnect or PE resources is now reduced to an acceptable 80 ns.

Because of resource sharing, physical packets may arrive at a PE interleaved by stream. Each PE includes a buffer to reassemble the physical packets into logical packets, where the buffer size is PE-dependent. Once a complete logical packet has been reassembled, it is issued to the processing function within the PE.

Many protocols, such as WiFi and WiMax, have precise transmission timing requirements. Because of resource sharing, some amount of jitter through the PHY pipeline is expected. A time stamp field has been included in the logical packet header to schedule the transmission of logical packets, and, on reception, to synchronize our reference clock frequency with that of the transmitter.

8.3.2.4.2 Data-Driven Control

The packet header contains the address of the destination PE, Function ID that designates the operations to be performed on the packet, and additional control information. The Function ID enables flexibility and context switching to support simultaneous operation so that centralized control is not required for most of the data processing.

8.3.2.4.3 Router

The nine routers were synthesized in a 90 nm CMOS process. The area is estimated to be 0.8 mm^2, and the power is 18 mW when clocked at 250 MHz.

8.3.2.4.4 Flow Control

In the SCC, the physical transfer digit (phit) is of the same size as the flow control digit (flit), or 32 bits, so FC is performed on every phit. Currently, flow control is performed in a point-to-point (P2P) basis on every router port separately. A wormhole FC is implemented with a stall/go backpressure mechanism: one request line is used for forward flow control (FFC), and one stall/go line is used for the reverse flow control (RFC). No end-to-end flow control is implemented; the backpressure mechanism, which propagates the sink's buffer availability upstream through the routers up to the traffic source, stops the flit injection into the network. The flits injected, in the meantime, generate compressions along the worm (packet trajectory through the network), thus requiring extra buffers at each router port to avoid flit loss.

Wormhole flow control was selected to fulfill the NoC power constraints because it reduces buffer size compared to cut-through or packet-buffer flow control, and it allocates channel bandwidth more efficiently than bufferless flow control, which is susceptible to packet drop or packet deflection.

8.3.2.5 Data Flow and Reusability

Figure 8.29a through 8.29d show the dataflow for WiMAX Tx flow, WiFi Tx flow, WiMAX Rx flow, and DVB Rx flow, respectively. It illustrates how the NoC supports the flexibility and area efficiency of this architecture. Many of the PEs are shared across multiple protocols. The NoC is the key to efficiently reusing the PEs to reduce the overall hardware area, compared to the discrete fixed-function solutions.

It is reported that the SCC-based multiradio baseband is estimated to be 34% smaller than the same radio built from three discrete ASICs [22]. They are expected to instantiate a silicon prototype soon.

8.4 ACADEMIC IMPLEMENTATIONS

8.4.1 FAUST (Flexible Architecture of Unified System for Telecom)

Eleven European industrial research institutes and universities launched a joint project named 4-MORE—4G Multicarrier CDMA multiple antenna System on Chip for Radio Enhancements. Recently, their architecture has been published for the application of multicarrier OFDM-based baseband processing, such as 802.11n, 802.16e, and 3GPP/LTE [25,26]. They proposed the asynchronous Network on Chip (ANOC) with GALS paradigm. The ANOC architecture uses virtual channels to provide low latency and QoS, which is implemented in quasi delay-insensitive (QDI) asynchronous logic [27].

The FAUST chip integrates 20 asynchronous NoC routers, 23 synchronous units including an ARM946 core, embedded memories, various IP blocks, reconfigurable

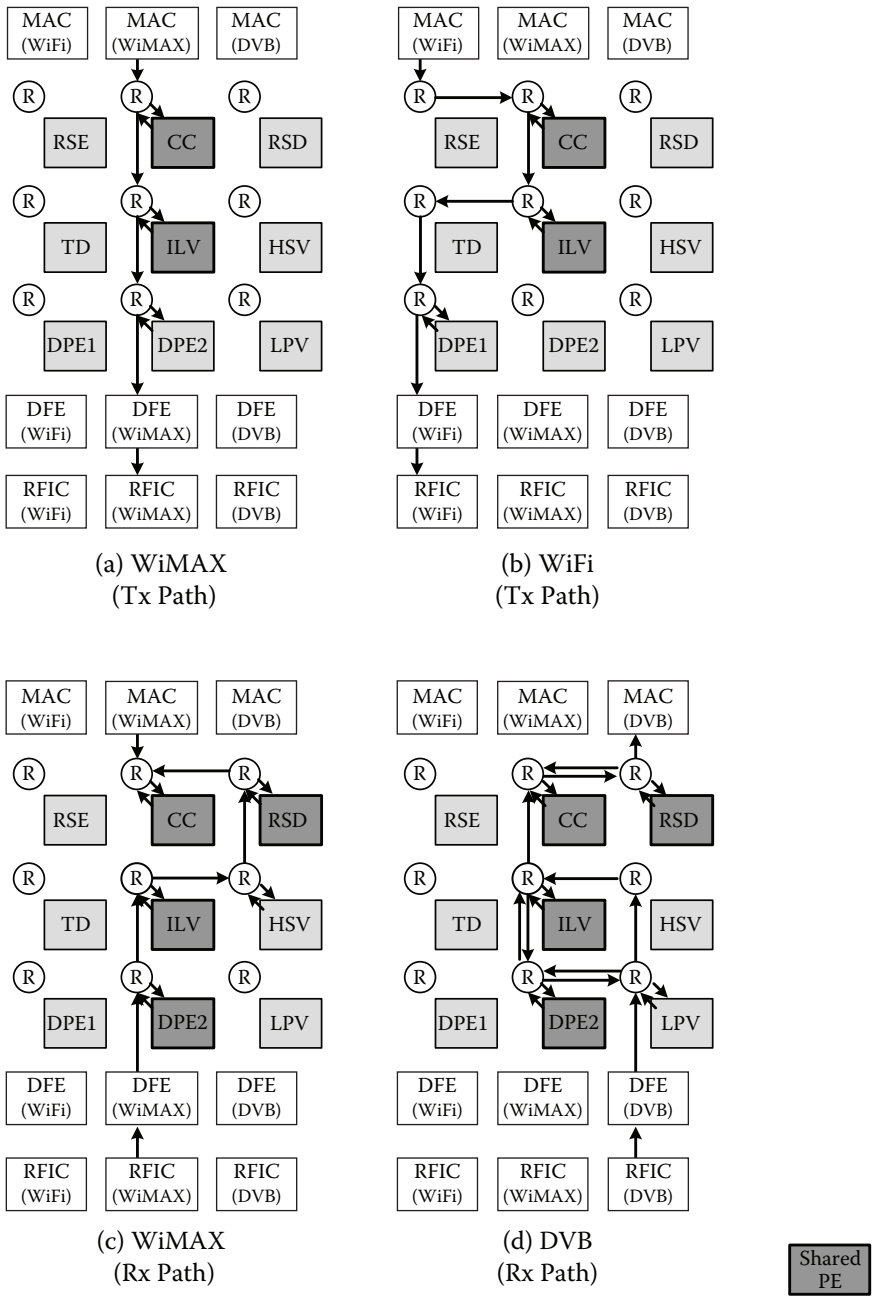


FIGURE 8.29 Data flow for (a) WiMAX Tx, (b) WiFi Tx, (c) WiMAX Rx, and (d) DVB Rx flow.

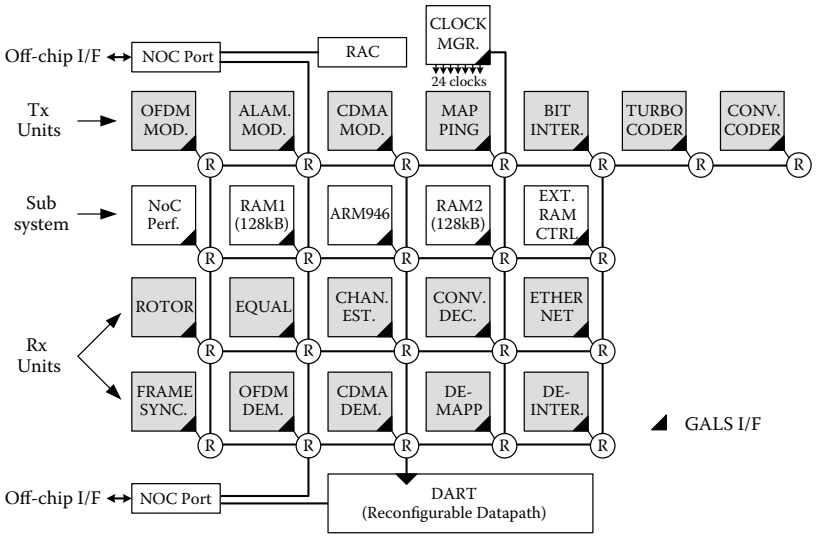


FIGURE 8.30 FAUST architecture.

datapath engines, and one clock management unit to generate the 24 distinct unit clocks (see Figure 8.30).

To integrate any synchronous IP within ANOC architecture, the dedicated NI performs two main tasks: it synchronizes the synchronous and asynchronous logic domains using ad hoc decoupling FIFOs [28], and provides all facilities to access the NoC communication infrastructure such as network routing path programming, network data packet generation, and IP core configuration.

Table 8.4 presents the main features of the FAUST chip: it is implemented in a 6 ML, 130 nm, 1.2 V, CMOS process from STMicroelectronics. The whole chip integrates more than 3M gates and 3.5 Mb of embedded RAM, which corresponds to a core area of 72.71 mm^2 and a chip area of 79.5 mm^2, respectively. The maximum NoC throughput measured between two adjacent nodes or between an IP and its connected node is 5.12 Gbps per link. The latency is about 6 ns per crossed node, 12 ns for the GALS IF, and 12 ns for the NI.

A real-time 100 Mbps SISO OFDM transceiver needs a bandwidth of 10 Gbps, which corresponds to a 10% network load, for a complexity of 1.7M gates at the TX-level and 1.9M gates at the RX-level. The integration of each IP within the NoC costs 41K gates (~ 0.45 mm^2) for the 5 × 5 asynchronous node (19K), the GALS interface (12K), and the network interface (10K without configuration registers). The 20-node NoC represents about 15% of the overall area, and the average complexity of the 23 IPs connected is close to 300K gates (including RAM). Using an optimized frequency scaling between 160 MHz and 250 MHz, the transceiver functions consume 640 mW in TX mode and 760 mW in RX mode. The NoC consumption represents only about 6% of the overall consumption for a typical traffic defined by the targeted applications (see Figure 8.31). The cost of the NoC in terms of area is similar to that of a bus-based architecture, but the properties of NoC structures are better suited to addressing the design issues associated with complex SoCs.

TABLE 8.4
Features of the FAUST Chip

	NoC Architecture
Topology and size	2D-Mesh including 20 nodes
Switching and routing modes	Packet switching and wormhole routing
Flow-control technique	Credit based
Flit size (i.e., Word size)	32 bits
QoS support	2 Virtual channels
I/O	Direct external NoC accesses
Implementation	Asynchronous logic (QDI)
Power-saving technique	Dynamic frequency scaling
	Computing and Memory Aspects
IP count	23 Units connected to the NoC
Processor core	ARM946ES
DMA engines	3 Units to manage on-chip and off-chip memories
Reconfigurable datapath	SIMD structure for channel estimation
Host computer interface	100 Mbps full duplex Ethernet unit
	Technology and Complexity
Process	130 nm CMOS 6ML (STMicroelectronics)
Logic gate count	3.124 M gates (excluding SRAM blocks)
On-chip clocks	24 Clocks
Die size	8.900 mm * 8.933 mm = 79.50 mm^2
Package	BGA420 (35 mm * 35 mm)
Signal I/O count	275
Supply voltage	1.2 V Core, 3.3V I/O
	Measured Performance
NoC throughput	5.120 Gbps per link
IP operating frequency	162 MHz
Chip power consumption	640 mW TX mode, 760 mW RX mode
NoC area	15% of the global area

8.4.2 RAW

The search for processors that can exploit the increase in instruction-level parallelism (ILP) has led to several creative choices. MIT's Raw processor has replaced the traditional bypass network with a more general interconnect for operand transport. Raw is probably the most significant example of a multiprocessor using a scalar operand network (SON), which can be defined as a set of mechanisms that join the dynamic operands and operations of a program in space to enact the computations specified by a program graph. Recent SONs incorporate point-to-point mesh interconnects; i.e., they use NoCs for internal data and instruction transfers. Multiprocessing has addressed the issue of scalability by distributing resources

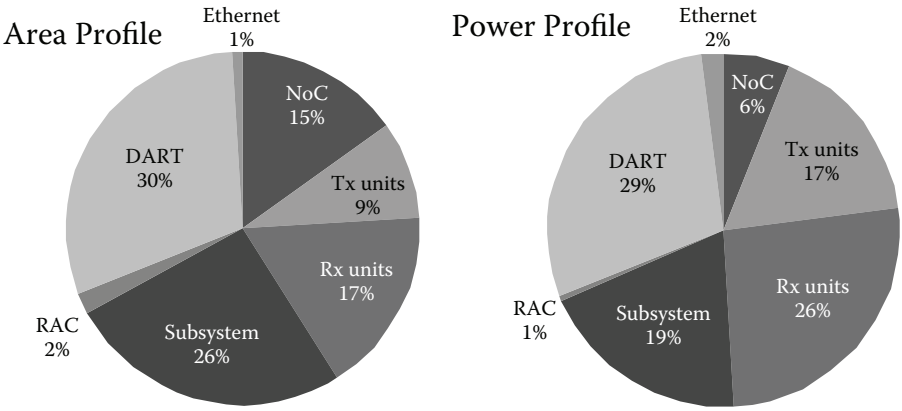


FIGURE 8.31 Area and power profile of FAUST chip.

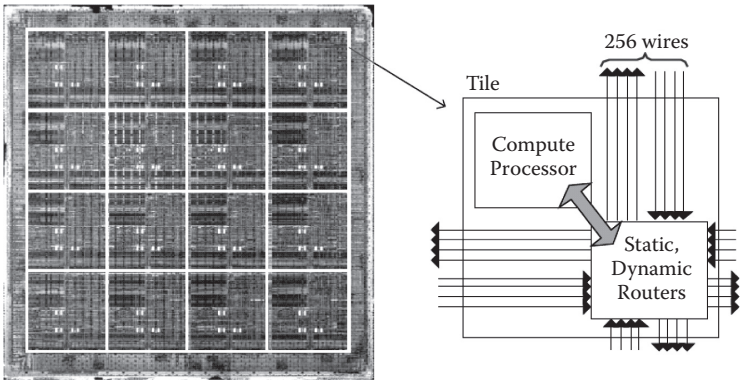


FIGURE 8.32 RAW chip photograph and a tile organization.

and pipelining paths between distant components. Such approaches do not solve completely the bandwidth scalability problem, which occur when the amount of information that needs to be transmitted and processed grows rapidly with the size of the system.

The MIT Raw processor [29] shown in Figure 8.32 addresses bandwidth scalability using an NoC SON with a mesh topology. We report here two variants of Raw, the first with static and the second with dynamic transport. Both versions are based on the Raw processor, which consists of a 2-D mesh of identical, programmable tiles connected by two types of transport networks. Each tile is sized so that the signals can travel through the tile in a clock cycle. The system can be scaled up by using more tiles, with frequency of operation being constant. Each Raw tile contains a single issue in-order processor and a number of routers. The switching portion of the tile contains two dynamic routers (for two dynamic transport networks) and one static router (for a static transport network). The static version or Raw uses the static router, whereas the dynamic version uses one of the dynamic routers. A credit system is used to prevent FIFO overflow. The additional dynamic router is used for

cache-miss traffic. Indeed, misses are turned into messages that are routed to the side of the mesh and eventually to the off-chip distributed DRAMs.

A mesh interconnect is overlaid on the tiles. Because all links can be programmed to route operands only to those tiles that need them, the bandwidth that is required is much smaller as compared to a shared-bus approach. Each of the two SONs relies on a compiler to assign operations to tiles and to program the network transports to route operands between the corresponding instructions. Thus, assignment in Raw can be defined as static. Furthermore, Raw uses in-order execution. Both versions of Raw (static and dynamic) support exceptions. Branch conditions and jump pointers are transmitted over the NoC-like data. Raw's interrupt model allows each tile to take and process interrupts individually. Cache misses stall the processor, which processes the miss only. The dynamic version of Raw uses a dynamic dimension-ordered, wormhole-based routing to route operands between tiles. The Raw dynamic router is significantly more complex because it handles deeper logic levels compared to the static router to determine the path of the incoming operands.

An experimental version, Raw, was realized with 16 tiles. Raw was implemented using 180 nm technology with six levels of metallization. The area is 330 mm^2, and the pinout is 1657. The chip operates at 425 MHz at a nominal voltage of 1.8 V. Each of the 16 tiles contains an 8-stage, in-order, single-issue MIPS-like processor, a stage-pipelined FPU, a 32-bit data cache, the three routers, and 96 kB of instruction cache. Tiles are connected to the nearest neighbors, using, for separate networks, two static and two dynamic networks. The links consist of more than 1034 wires/tile.

REFERENCES

1. Lee, S. et al., An 800MHz star-connected on-chip network for application to systems on a chip, *IEEE Int. Solid-State Circuits Conf.* Dig. Tech. Papers, 2003, pp. 468–469.
2. Lee, S. et al., Packet-Switched On-Chip Interconnection Network for System-on-Chip Applications, *IEEE Transactions on Circuits and Systems II,* Vol. 52, No. 6, pp. 308–312, June 2005.
3. Lee, K. et al., A Distributed On-Chip Crossbar Switch Scheduler for On-Chip Network, *IEEE Custom Integrated Circuits Conf.,* September 2003, pp. 671–674.
4. Lee, K. et al., A 51 mW 1.6G Hz On-Chip Network for Low-Power Heterogeneous SoC Platform, In *IEEE Int. Solid-State Circuits Conf.* Dig. Tech. Papers, 2004, pp. 152–153.
5. Lee, K. et al., SILENT: serialized low energy transmission coding for on-chip interconnection networks, In *ACM/IEEE Int. Conf. on Computer-Aided Design,* 2004, pp. 448–451.
6. Lee, K. et al., Low Energy Transmission Coding for On-Chip Serial Communications, *IEEE Int. SOC Conf.*, September 2004, pp. 177–178.
7. Lee, K. et al., Networks-on-Chip and Networks-in-Package for High-Performance SoC Platforms, *IEEE Asian Solid Stated Circuits Conf.,* Nov. 2005, pp. 485–488.
8. Lee, K. et al., Low-power network-on-chip for high-performance SoC design, *IEEE Trans. VLSI systems,* Vol. 14, Feb 2006, pp. 148–160.
9. Lee, S. et al., Adaptive Network-on-Chip with Wave-Front Train Serialization Scheme, In *IEEE Symp. on VLSI Circuits* Dig. Tech. Papers, June 2005, pp. 104–107.
10. Lee, S. et al., Analysis and Implementation of Practical, Cost-Effective Networks on Chips, *IEEE Design and Test of Computers,* September 2005, pp. 422–433.

11. Kim, D. et al., A Reconfigurable Crossbar Switch with Adaptive Bandwidth Control for Networks-on-Chip, *Int. Symposium on Circuits and Systems*, May 2005, pp. 2369–2372.
12. Kim, K. et al., An Arbitration Look-Ahead Scheme for Reducing End-to-End Latency in Networks-on-Chip, *Int. Symposium on Circuits and Systems*, May 2005, pp. 2357–2360.
13. Chung, D. et al., A Chip-Package Hybrid DLL Loop and Clock Distribution Network for Low-Jitter Clock Delivery, *IEEE Int. Solid-State Circuits Conf.*, February 2005, pp. 514–515.
14. Sohn, J.-H. et al., A 50M vertices/s Graphics Processor with Fixed-Point Programmable Vertex Shader for Mobile Applications, *IEEE Int. Solid-State Circuits Conf.,* February 2005, pp. 192–193.
15. Kim, D. et. al, Circuits, Solutions for Real Chip Implementation Issues of NoC and Their Application to Memory-Centric NoC, In *Proc. of IEEE International Symposium on Networks-on-Chip (NOCS),* pp. 30–39, May 2007.
16. Lowe, D. G., Distinctive Image Features from Scale-Invariant Keypoints, *ACM International Journal of Computer Vision*. Vol. 60, Issue 2, pp. 91–110. 2004.
17. Held, J. et al., From a Few Cores to Many: A Tera-scale Computing Research Overview, white paper, Intel Corporation, www.intel.com.
18. Vangal, S. et al., An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, In *IEEE Int. Solid-State Circuits Conf.* Dig. Tech. Papers, 2007, pp. 98–99.
19. Vangal, S., Borkar, N. Y., and Alvandpour, A., A Six-Port 57GB/s Double-Pumped Non-blocking Router Core, *Dig. Symp. VLSI Circuits*, pp. 268–269, June 2005.
20. Tschanz, J., Narendra, S. G., and Ye, Y. et al., Dynamic Sleep Transistor and Body Bias for Active Leakage Power Control of Microprocessors, *IEEE J. Solid-State Circuits*, pp. 1838–1845, November 2003.
21. Khellah, M., Kim, N. S., and Howard, J. et al., A 4.2GHz 0.3mm2 256kb Dual-Vcc SRAM Building Block in 65nm CMOS, *ISSCC Dig. Tech. Papers*, pp. 624–625, February 2006.
22. Hoffman, J. et al., Architecture of the Scalable Communications Core, *IEEE International Symposium on Networks-on-Chip,* pp. 40–49, May 2007.
23. Chun, A. et al., Application of the Intel® Reconfigurable Communications Architecture to 802.11a, 3G and 4G Standards, *Frontiers of Mobile and Wireless Communication*, May 31–June 2, 2004.
24. OCP-IP Association, *Open Core Protocol Specification 2.1*, rev 1.0.
25. Lattard, D. et al., A Telecom Baseband Circuit based on an Asynchronous Network-on-Chip, *ISSCC Dig. Tech. Papers*, pp. 258–259, February 2007.
26. Viviet, P. et al., FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications, In *Proc. Design, Automation and Test in Europe Conf.,* University Booth, 2007.
27. Beigne, E., Clermidy, F., Vivet, P., Clouard, A., and Renaudin, M., An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-level Design Framework, *Proc. ASYNC'05,* New York, pp. 54–63, March 2005.
28. Beigne, E. and Vivet, P., Design of On-chip and Off-chip Interfaces for a GALS NoC Architecture, *Proc. ASYNC'06,* Grenoble, France, pp. 172–181, March 2006.
29. Talyor, M.B. et al., A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network, *IEEE International Solid-State Circuits Conf.,* February 2003, pp. 170–171.

Appendix: BONE Protocol Specification

There are many complicated and fancy topologies and protocols in long-distance and wide-area networks (LAN and WAN), such as computer network and telephone network. Most of NoC researches just proposed one or entire features of the long-distance network onto a chip to implement SoC. Those approaches are not practical and efficient for SoC because the on-chip traffic is not so undeterministic as that of the long-distance network, and latency should be minimized more tightly than WAN and LAN.

Simple but efficient architecture and circuits should be used in the implementation of NoC. In this chapter, a detailed explanation of how such NoC architecture and circuits are designed will be given with BONE as an example. The Basic On-chip Network (BONE) will be divided into the abstract layer and the physical design layer. The abstract layer again can be classified into BONE protocol and Electrical Signaling. The physical design layer is composed of the architecture and circuits. BONE provides an on-chip communication architecture and its protocol standards for the high-performance and application-specific SoC design. The BONE uses its network architecture to interconnect and integrate multifunctional units or intellectual properties (IPs) with sufficient bandwidth and minimum latency and, especially, without the need of the global synchronization.

A.1 OVERVIEW OF BONE

BONE is based on the star topology and consists of five kinds of components: master network interface (MNI), slave network interface (SNI), Up_Sampler (UPS), Dn_Sampler (DNS), and switch (SW). The MNI connects a master to the BONE. Using the UPS and DNS, the BONE serializes and deserializes packets. The nonblocking SW routes packets. A path from MNI to SNI is called *forward path*, and the reverse path is called *backward path*. See Figure A.1.

Although it is not 1:1 compatible with the OSI seven-layer protocol, it is useful to look over the BONE protocol from the view point of the OSI protocol. The NoC usually extends from the physical layer to the fourth layer, the transport layer. In the first place, let us take a look at the physical layer of the BONE. The packet signals are composed of 32b ADDR, 32b DATA, 8b SIG, and 7b routing information (RI) for packet routing. The clock frequency in BONE is higher than those of the master and slave. Exact synchronization between different clocks is not required.

The link layer of BONE features serialization/deserialization to reduce the number of long interconnection wires while maintaining the bandwidth by increasing the clock frequency. It can support multilevel flow control.

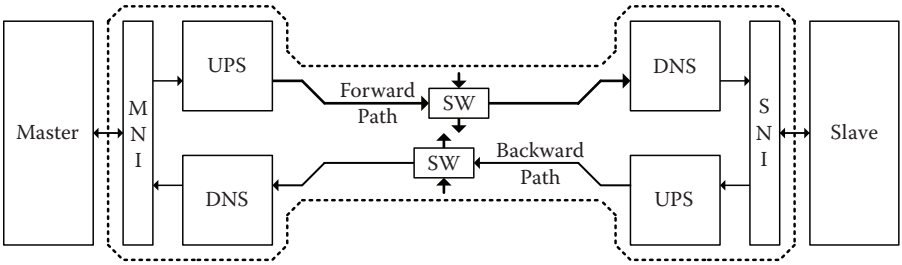


FIGURE A.1 Overall architecture.

For the network layer, the network is managed by a separate control packet. The source routing is achieved with 7b, and cut-through switching is adopted.

Although the transport layer is not supported explicitly, a few considerations on the BONE can be classified as the features of the transport layer. The header size can be reduced for the burst read/write operation to enhance the utilization of the network. In addition, the burst length can be programmed into 1, 2, 4, and 8.

- Encoding/decoding scheme to reduce packet size
- Acknowledge packet for end-to-end hand-shaking
- Direct-Signal (SIG) field for directed-mapped signal transfer

A.2 BONE PROTOCOL

A.2.1 Packet Format

BONE supports the layered (hierarchical) protocols and packet-level transaction. Its 80b packet is composed of 16b RI for packet routing, 32b ADDR, and 32b DATA. W field indicates read or write. HEADER includes TAG and the Direct-Signal (SIG) field. In the compact packet, the header is only 1 Byte which is called C_HEADER. The Header of the control packet contains the hop count (HC) field instead of the SIG field. The acknowledge packet consists of TAG. See Figure A.2 and Table A.1.

The 7b RI field is generated by MNI, and the master is assumed to access the slave by address mapping. The 7b RI contains route information from a master to a slave. The RI field can is divided into a few subfields, each of which corresponds to an output port at every switch hop. RI modification (deleting the head index and attaching a flipped source port index) is required to record the route path. RI for the reverse path can be obtained by bitwise flip operation. RI modification and flipping for reverse-path RI are depicted in Figure A.3.

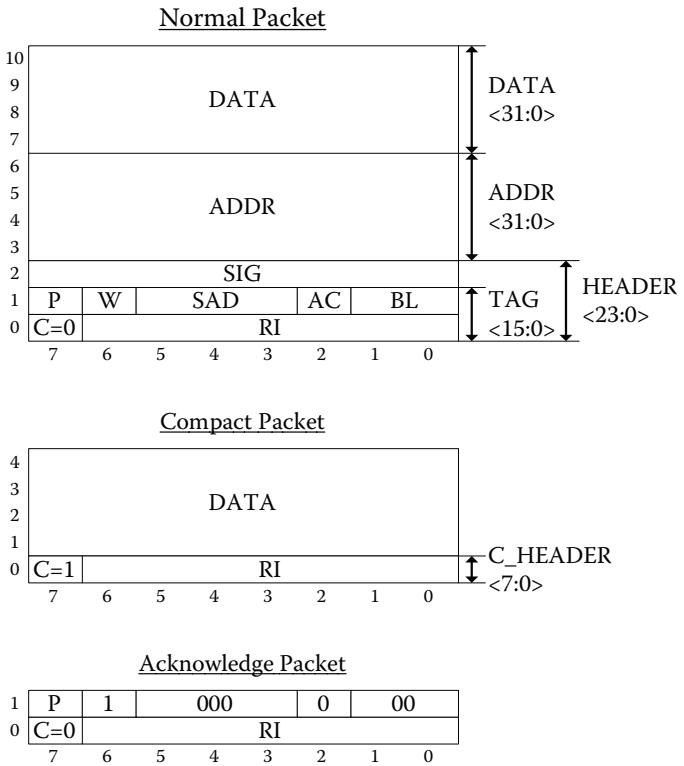


FIGURE A.2 Packet formats (BONE 2).

TABLE A.1
TAG Field Description

C	Compact. 1 for compact packet, 0 for others.
RI	Route information.
P	Priority. 1 for high priority.
W	Write.
SAD	3b bitmap encoding. Each bit indicates whether the corresponding field exists or not (e.g., 101 means direct signal and data field exist).
AC	Acknowledge Request.
BL	Burst Length.
HC	Hop Count. Every switch performs shift-right. Control packet is switched according to RI until HC becomes 0.

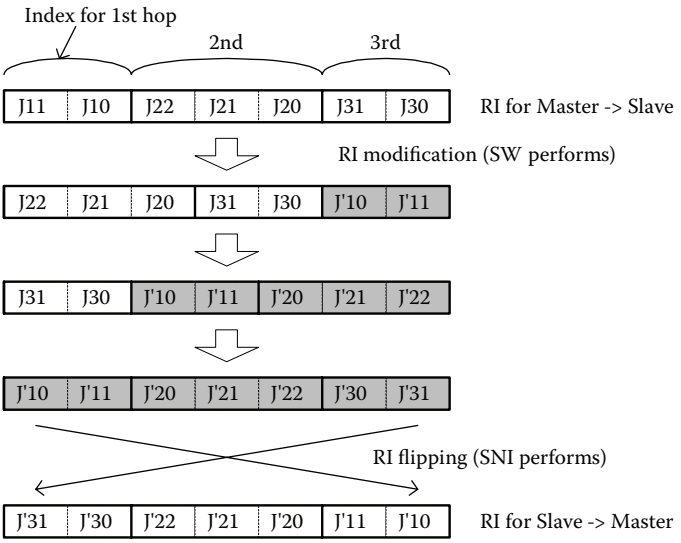


FIGURE A.3 RI modification and flipping.

A.2.2 BONE Signals

A.2.2.1 Master Network Interface (MNI)

MNI accepts data from the master and converts the data to the internal BONE format for processing by UPS. This stream of data is called *forward direction*, and the initial F is assigned to the signal names. In addition, MNI receives the data in the internal BONE format from UPS and reconverts them to the master format. This data stream is called as *backward direction* and the initial B is attached to the signal names. The details of input/output signals can be varied by the interface conditions of the master. Here, take an example of the simple MNI as shown in Figure A.4.

NI, including MNI and SNI, deals with the BONE protocol packet consisting of a 32b address, 32b data, and a 24b (or 8b) header field. Its interface to the processing element is very flexible and easily fit to the individual I/O specification of masters and slaves. For flow control, WTMSREQ (Wait Master Request) can be received from the UPS and WTDNS (Wait DNS) can be issued to UPS. MNI also outputs WTMS (Wait Master) to the master and receives WTDNSREQ (Wait DNS Request) from the master. See Table A.2 for a description of MNI signals.

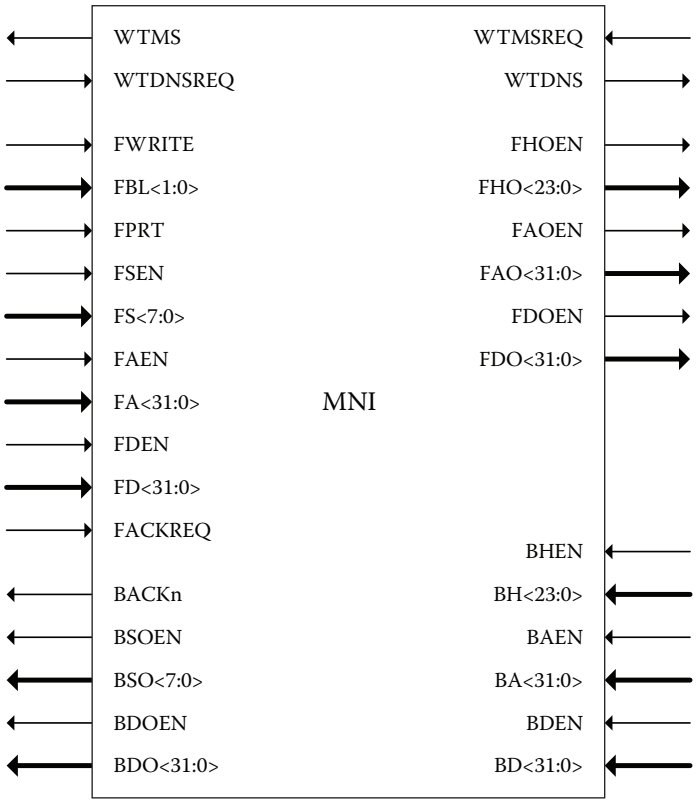


FIGURE A.4 MNI interface diagram.

TABLE A.2
MNI Signals

WTMSREQ	Wait Master Request.
WTMS	Wait Master. Bypassing WTMSREQ. Controls packet flow from a master.
WTDNSREQ	Wait Down_Sampler Request.
WTDNS	Wait Down_Sampler. Bypassing WTDNSREQ. Controls packet flow from DNS.
FWRITE	Forward path HIGH for write command, LOW for read command.
FBL<1:0>	Forward path Burst Length (00:1, 01: 2, 10:4, 11:8).
FPRT	Forward path priority (1: High priority 0: low priority).
FSEN	Forward path Direct-Signal Enable.
FS<7:0>	Forward path direct signals.
FAEN	Forward path Address Enable.
FA<31:0>	Forward path Address.
FDEN	Forward path Data Enable.
FD<31:0>	Forward path Data.
FACKREQ	Forward path Acknowledge Request. NI which receives a packet with ACKREQ must reply by an ACK packet.
FHOEN	Forward path Header Output Enable.
FHO<23:0>	Forward path Header Output.
FAOEN	Forward path Address Output Enable.
FAO<31:0>	Forward path Address Output.
FDOEN	Forward path Data Output Enable.
FDO<31:0>	Forward path Data Output.
BHEN	Backward path Header Enable.
BH<23:0>	Backward path Header.
BAEN	Backward path Address Enable.
BA<31:0>	Backward path Address.
BDEN	Backward path Data Enable.
BD<31:0>	Backward path Data.
BACKn	Backward path Acknowledge Not. Set to H when ACKREQ is asserted. Set to L when an ACK packet arrives.
BSOEN	Backward path Direct Signal Output Enable.
BSO<31:0>	Backward path Direct Signal Output.
BDOEN	Backward path Data Output Enable.
BDO<31:0>	Backward path Data Output.

A.2.2.2 Up_Sampler (UPS)

The role of UPS is to convert the traffic data into traffic signals for the on-chip network. In BONE, the main role of UPS is the serialization of the packet data. It has a 32b-wide input port for Address, 32b for Data, and 24b for Header of a packet from MNI. It has an 8b-wide serial link to transfer data to the switch. For the flow control, there are the WTI (Wait Network Interface) and the WTIREQ (Wait Network Interface Request). See Figure A.5 and Table A.3.

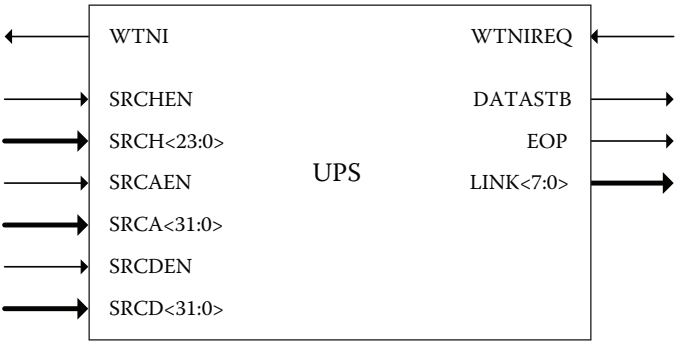


FIGURE A.5 UPS interface diagram.

TABLE A.3
UPS Signals

WTNIREQ	Wait Network Interface Request.
WTNI	Wait Network Interface. Retiming WTNIREQ.
SRCHEN	Source Header Enable. Source may be one of MNI or SNI.
SRCH<23:0>	Source Header.
SRCAEN	Source Address Enable.
SRCA<31:0>	Source Address.
SRCDEN	Source Data Enable.
SRCD<31:0>	Source Data.
DATASTB	Datastb. Destination retimes packets at rising edge.
EOP	End-of-packet.
LINK<7:0>	Link through which packets are delivered.

A.2.2.3 Switch (SW)

BONE can support various sizes of cross-point switches. Here, the 8 × 8 cross-point switch will be explained as an example. Simple CMOS transfer gates are arranged in mesh structure as shown in Figure A.6. It receives WTSWREQ (Wait Switch Request) signal from DNS to slow down or stop its packet transfer to DNS. It also generates WTUPS (Wait UPS) to inform UPS of the overflow of switch buffer. See Table A.4 for a description of SW signals.

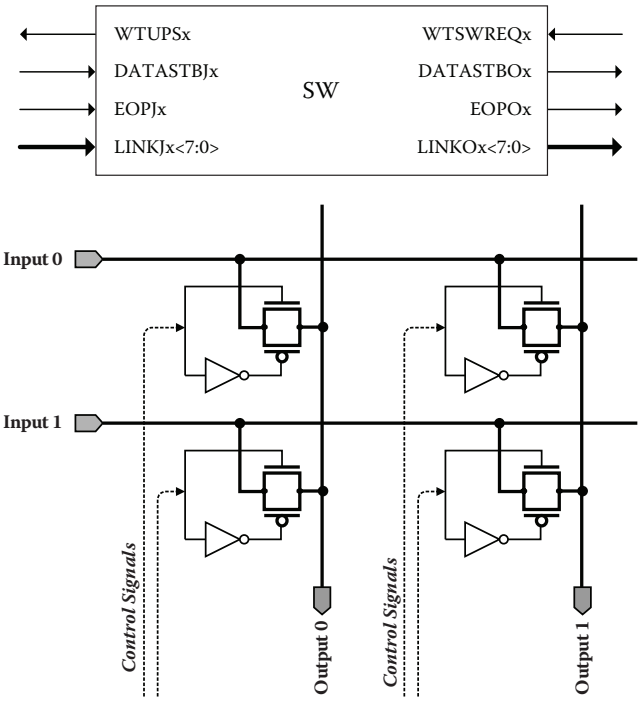


FIGURE A.6 SW circuit diagram.

TABLE A.4
SW Signals

WTSWREQx	Wait Switch Request. Switch must slow down or even stop packet transfer; x indicates port no.
WTUPSx	Wait UPS. Switch buffer is close to overflow.
DATASTBJx	Datastb Input. Switched w/o retiming.
EOPJx	End-of-packet Input. Switched w/o retiming.
LINKJx<7:0>	Link Input.
DATASTBOx	Datastb Output.
EOPOx	End-of-packet Output.
LINKOx<7:0>	Link Output.

A.2.2.4 Dn_Sampler (DNS)

DNS converts the serialized signals into packet data. It also plays the role of buffer for the packets and transfers the packets to SNI in synchronization with the BONE clock. It receives WTSWREQ from the SNI for the slow switch operation. Then, it relays the signal by generating WTSW(wait switch) to the switch. See Figure A.7 and Table A.5.

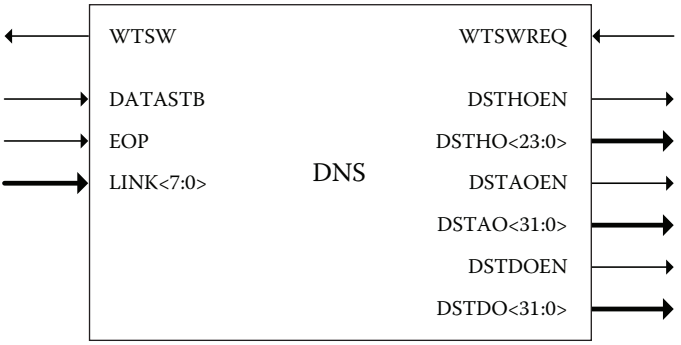


FIGURE A.7 DNS interface diagram.

TABLE A.5
DNS Signals

WTSWREQ	Wait Switch Request.
WTSW	Wait Switch. Retiming WTSWREQ.
DSTHOEN	Destination Header Output Enable.
DSTHO<23:0>	Destination Header Output.
DSTAOEN	Destination Address Output Enable.
DSTAO<31:0>	Destination Address Output.
DSTDOEN	Destination Data Output Enable.
DSTDO<31:0>	Destination Data Output.

A.2.2.5 Slave Network Interface (SNI)

SNI interfaces the DNS to the slave block. It converts the forward data in BONE packet protocol from the DNS into the format of the slave and backward data from SNI into the BONE protocol packet. Similar to MNI, its interface to the slave is quite versatile to accommodate any functional block. See Figure A.8 and Table A.6.

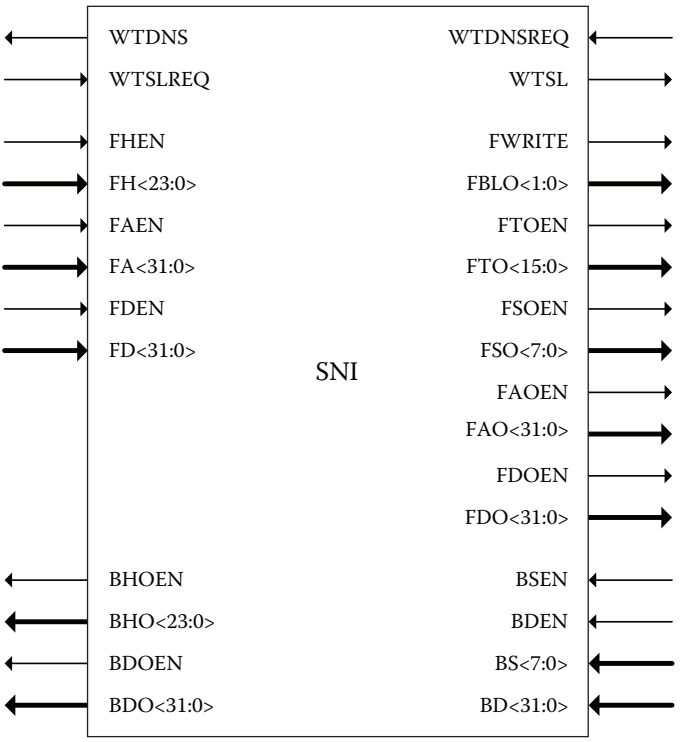


FIGURE A.8 SNI interface diagram.

TABLE A.6
SNI Signals

WTDNSREQ	Wait DNS Request. A slave can perform backpressure by this signal.
WTDNS	Wait DNS. Bypassing WTDNSREQ.
WTSLREQ	Wait Slave Request. A switch controls packet flow from a slave.
WTSL	Wait Slave. Bypassing WTDNSREQ.
FHEN	Forward path Header Enable.
FH<23:0>	Forward path Header.
FAEN	Forward path Address Enable.
FA<31:0>	Forward path Address.
FDEN	Forward path Data Enable.
FD<31:0>	Forward path Data.
FWRITE	Forward path H for write operation; L for read operation.
FBLO<1:0>	Forward path Burst Length Output.
FTOEN	Forward path Tag Output Enable.
FTO<15:0>	Forward path Tag Output. This tag information is used for backward_packet from slave to master.
FSOEN	Forward path Direct Signal Output Enable.
FSO<7:0>	Forward path Direct Signal Output.
FAOEN	Forward path Address Output Enable.
FAO<31:0>	Forward path Address Output.
FDOEN	Forward path Data Output Enable.
FDO<31:0>	Forward path Data Output.
BSEN	Backward path Direct Signal Enable.
BDEN	Backward path Data Enable.
BS<7:0>	Backward path Direct Signals.
BD<31:0>	Backward path Data.
BHOEN	Backward path Header Output Enable.
BHO<23:0>	Backward path Header Output.
BDOEN	Backward path Data Output Enable.
BDO<31:0>	Backward path Data Output.

A.2.3 Packet Transactions

Various packet transactions are explained in Figure A.9 through A.16.

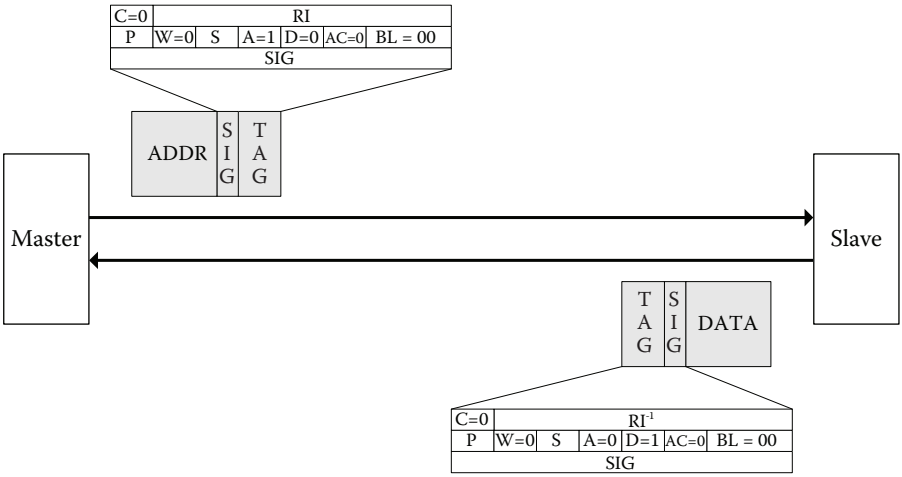


FIGURE A.9 Basic read packet transaction.

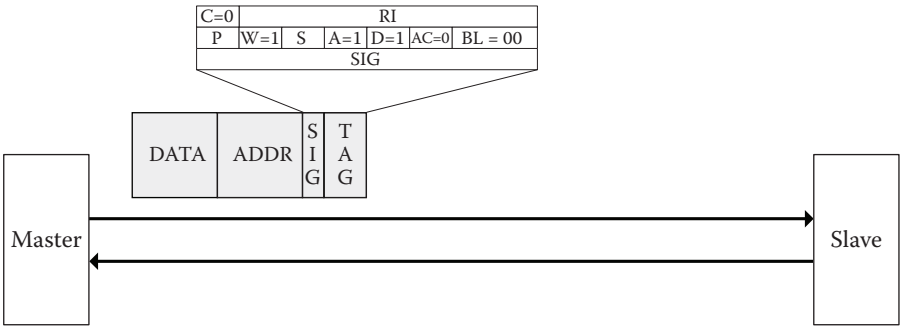


FIGURE A.10 Basic write packet transaction.

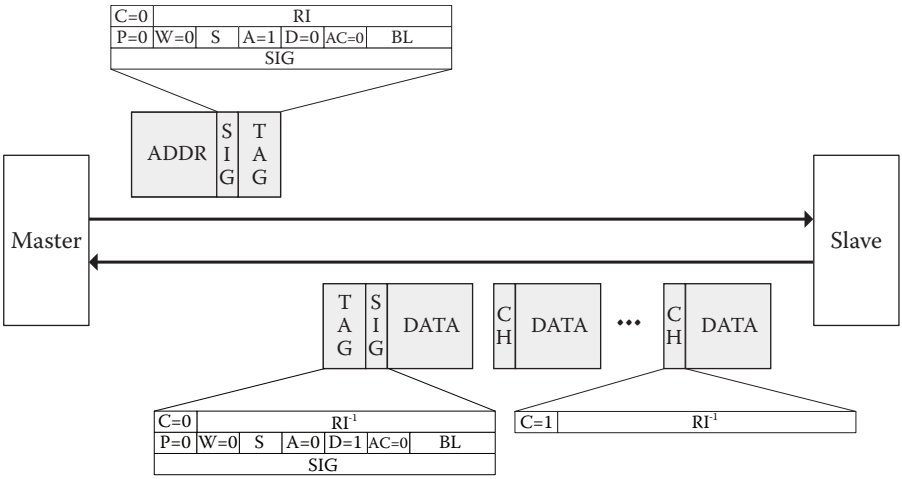


FIGURE A.11 Low-priority burst read packet transaction.

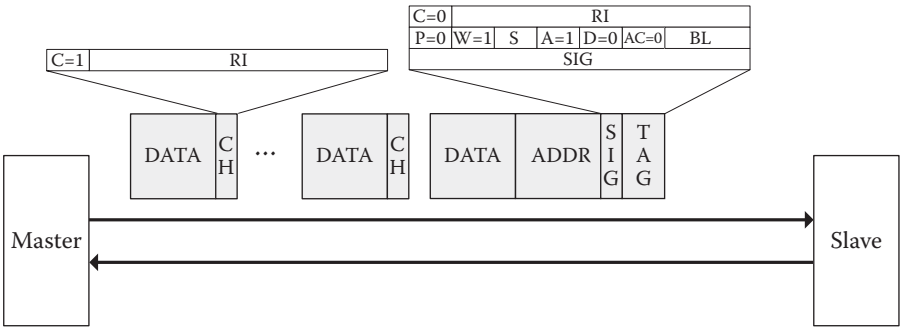


FIGURE A.12 Low-priority burst write packet transaction.

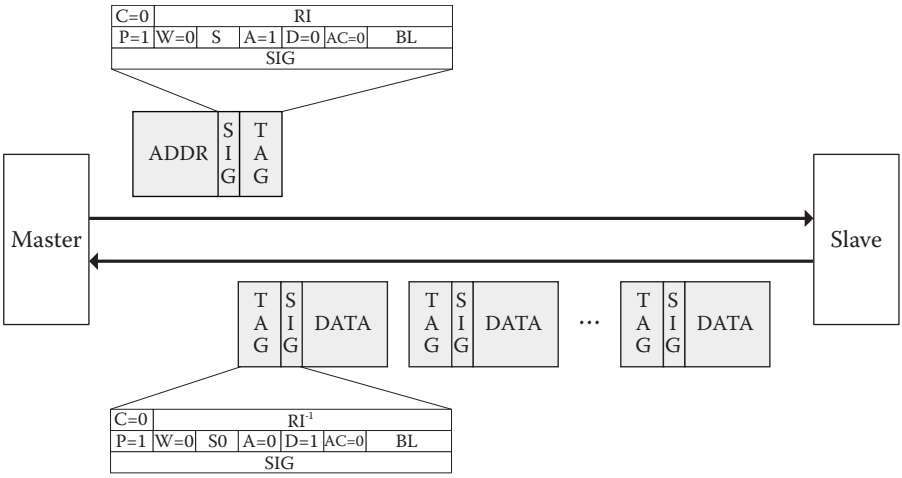


FIGURE A.13 High-priority burst read packet transaction.

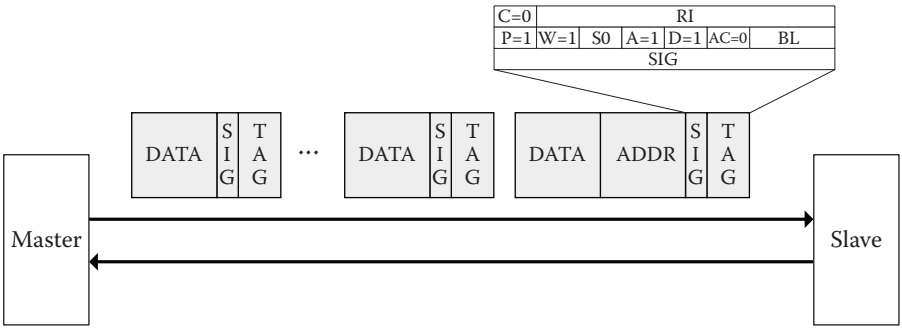


FIGURE A.14 High-priority burst write packet transaction.

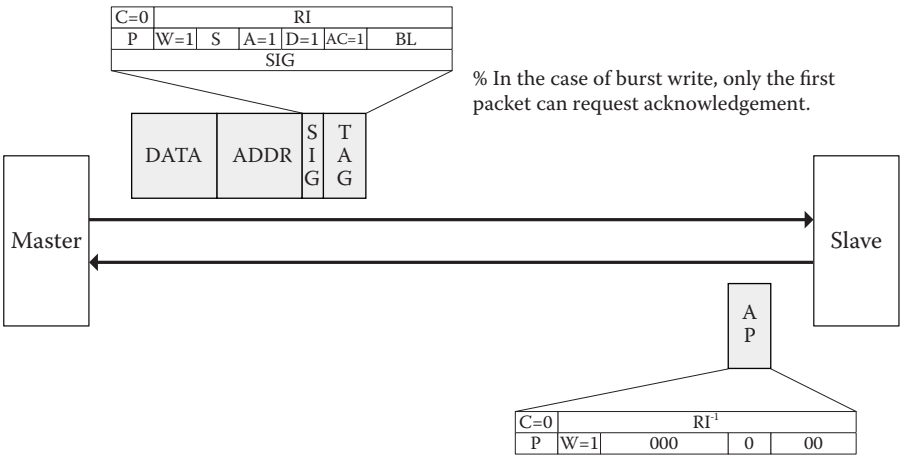


FIGURE A.15 Low-/high-priority, single/burst write w/ ackreq packet transaction.

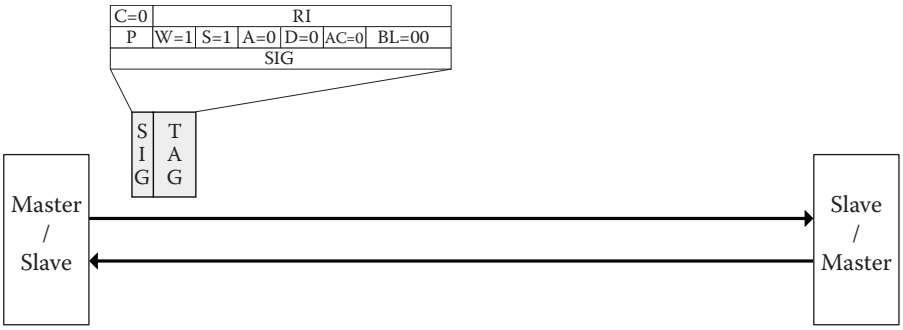


FIGURE A.16 Direct-signal packet transaction.

A.2.4 Timing Diagrams

A.2.4.1 Basic Read Packet Transaction

- Outputs of a master are latched-output. Settled in *phase_1*. Forward_path latency is zero [MNI_01].
- Before it outputs BS<7:0> and BD<31:0>, MNI examines whether a packet is destined to the MNI (i.e., the control packet) or master. If the packet is destined to the MNI, BSOEN and BDOEN are disabled [MNI_02].
- Backward_path latency is zero [MNI_03].
- SNI does not retime forward_path inputs, thus transfer them as soon as possible. In other words, forward_path latency is zero [SNI_01].
- SNI examines whether a packet is a control packet or not. If it is, FSOEN, FAOEN, and FDOEN are disabled. Then, SNI interprets the control packet [SNI_02].
- For read operation, SNI modifies FH, and thus generates FTO, to use it for a relevant backward_packet. C, P, and BL fields are not changed. RI is flipped. W is set to H. SAD is properly set. AC is set to L. The FTO must be delayed by the amount of slave latency so that it is output through BT together with BD [SNI_03].
- Backward_path inputs may be settled in *phase_2*, thus backward_path latency is 1 [SNI_04].

See Figure A.17 and A.18.

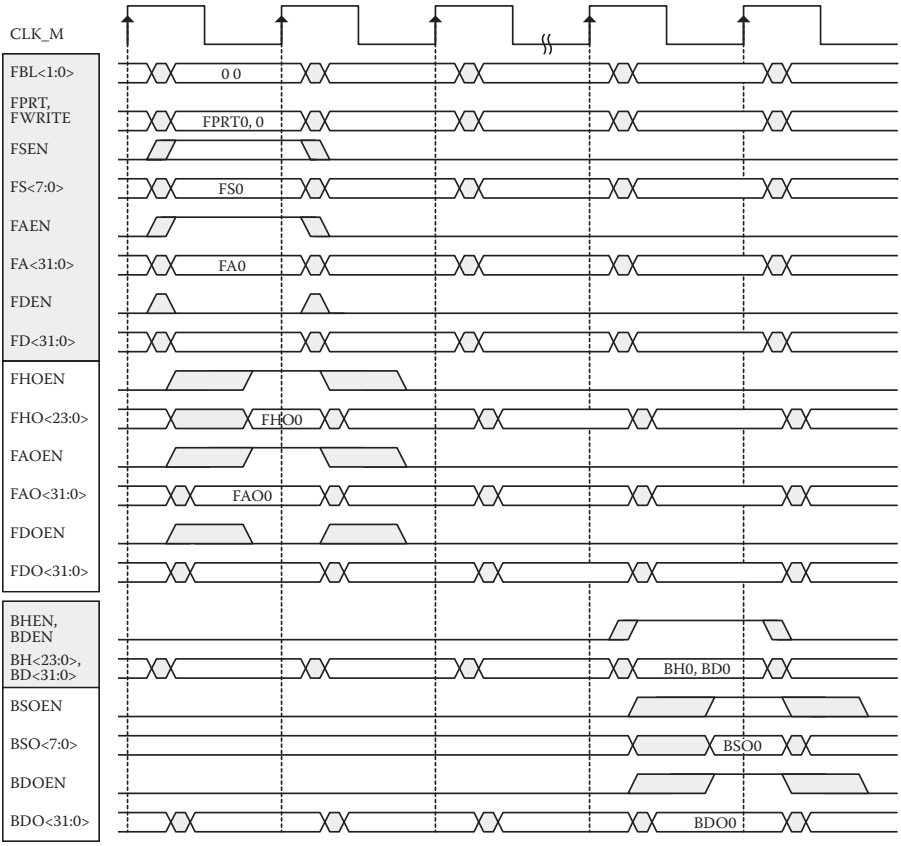


FIGURE A.17 (Master) ← MNI ← (UPS) protocol for basic read.

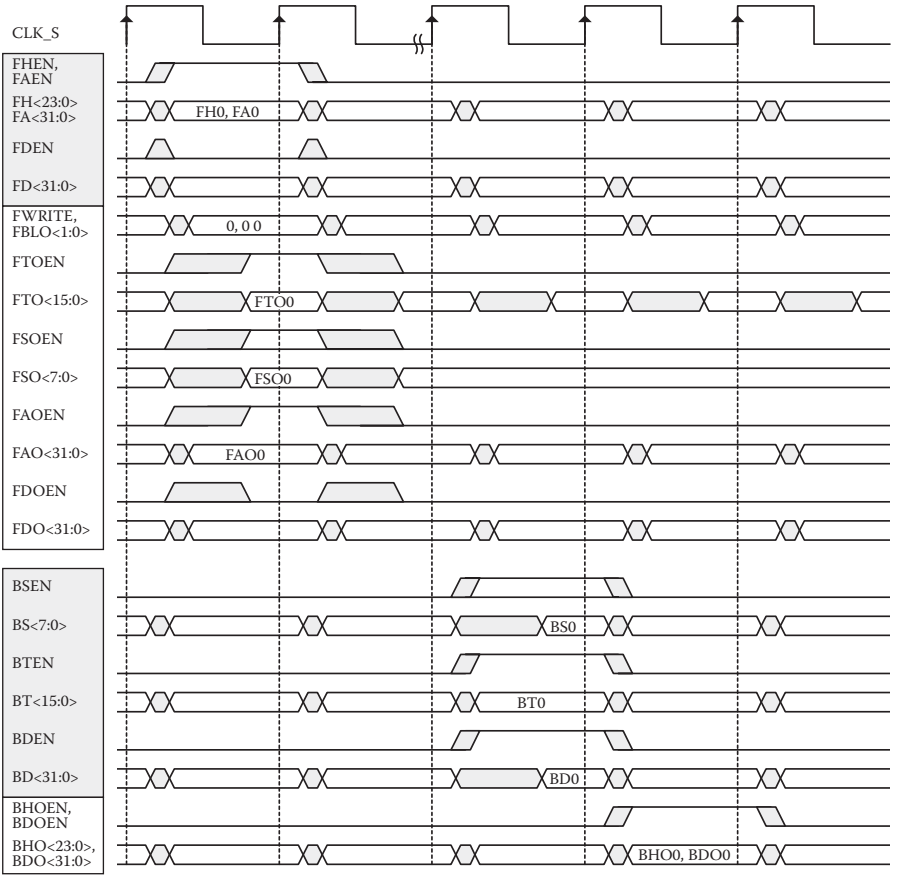


FIGURE A.18 (DNS) ← SNI ← (slave) protocol for basic read.

A.2.4.2 Basic Write Packet Transaction

- FTOEN is disabled for the write operation.

See Figure A.19 and A.20.

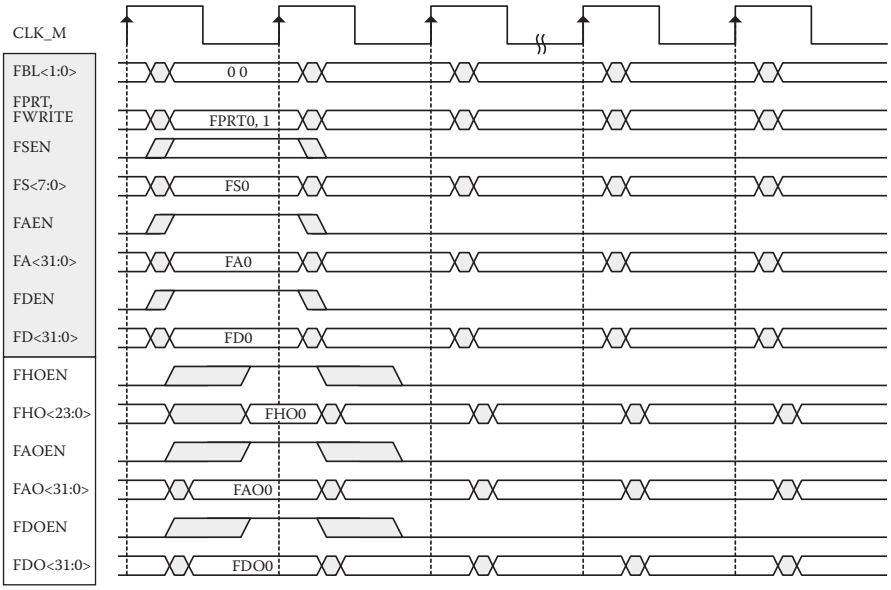


FIGURE A.19 (Master) ← MNI ← (UPS) protocol for basic write.

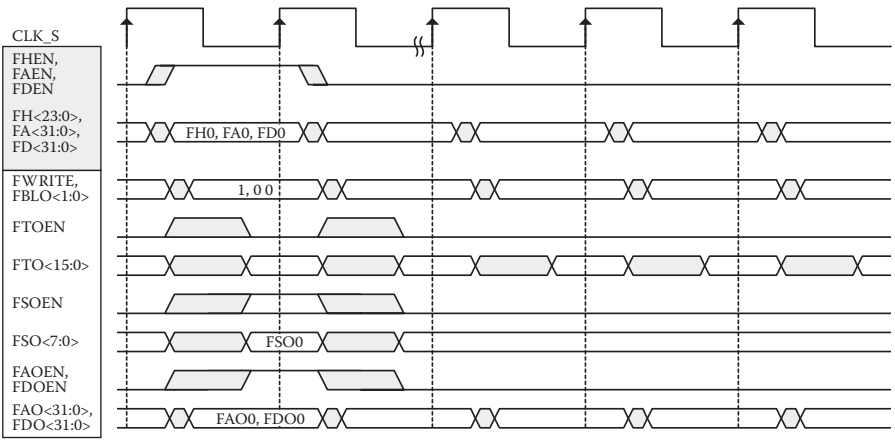


FIGURE A.20 (DNS) ← SNI ← (slave) protocol for basic write.

A.2.4.3 UPS/DNS Timing Diagram

When a UPS serializes a packet, it is serialized in the order of header, address, and data. The operation of the UPS and DNS is regardless of packet transfer types such as read/write, basic/burst, and acknowledgement. EOP signal is used to indicate the end of the packet.

- The minimum delay time, t_{UPSd}, which is required by UPS to start serialization, is open for actual design. This specification does not fix the value. See Figure A.21.
- The minimum delay time, t_{DNSd}, which is required by DNS to finish deserialization, is open for actual design. This specification does not fix the value. See Figure A.22.

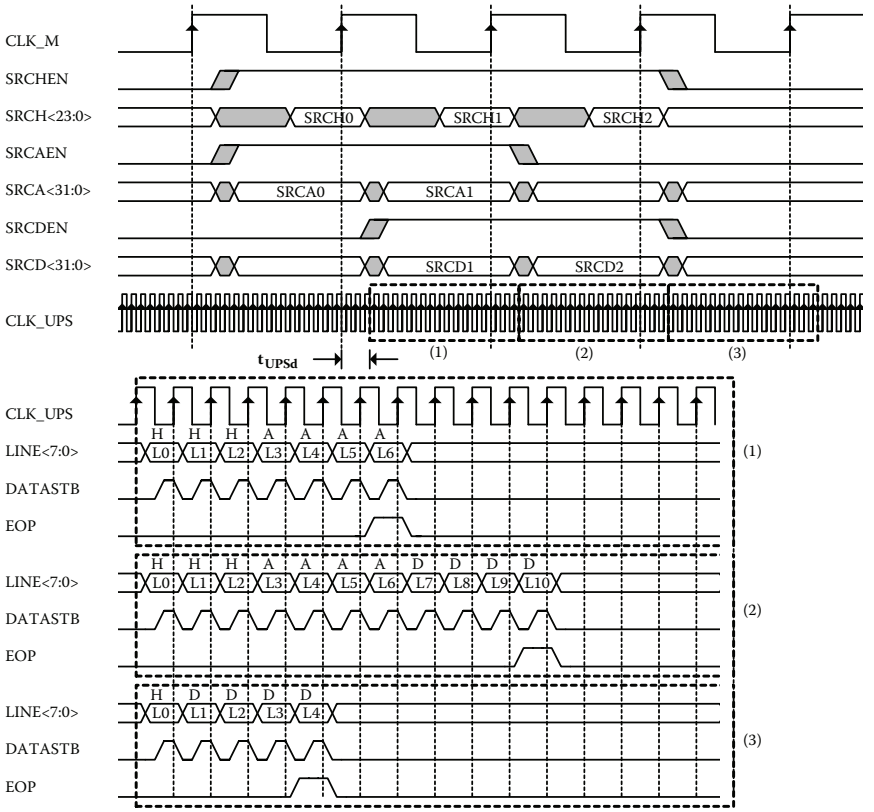


FIGURE A.21 (MNI) → UPS → (SW).

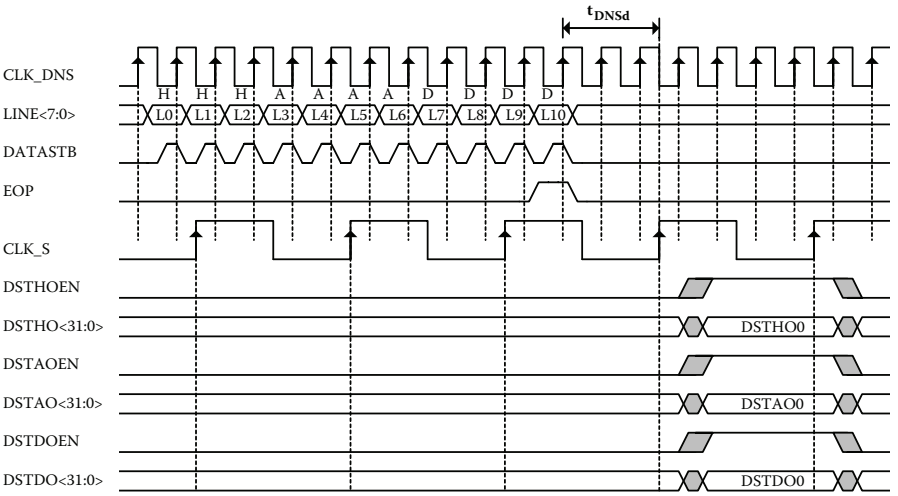


FIGURE A.22 (SW) → DNS → (SNI).

A.2.4.4 SW Timing Diagram

- The switching delay time, t_{SWd}, is open for actual design. This specification does not fix the value. See Figure A.23.
- The minimum setup time to request wait of a switch, tWT_SW_R, is 2 clock cycles [SW_01]. See Figure A.24.
- The minimum delay time from deassertion of WTSWREQ to LINEO, tWT_SW_F, is 1 clock cycle [SW_02].
- The minimum setup time of WTUPS to request wait of a UPS before the end of previous packet, is open for actual design. It may depend on the propagation delay and response time of the UPS.
- The minimum delay time from deassertion of WTUPS to LINEJ is 1 clock cycle [SW_03].

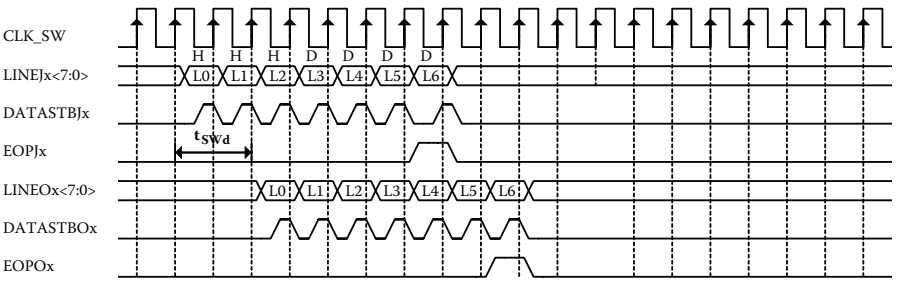


FIGURE A.23 Switch delay timing diagram.

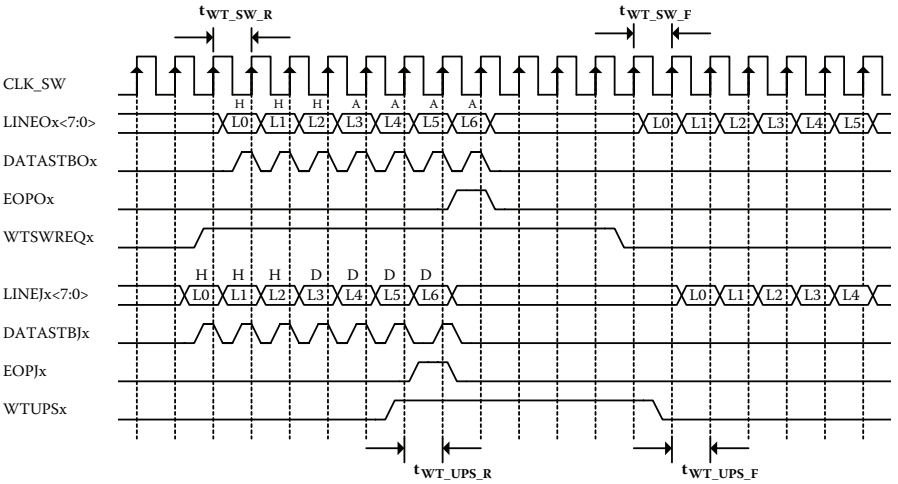


FIGURE A.24 Switch timing for flow control.