Embedded Systems

Embedded Software Design and Programming of Multiprocessor System-on-Chip

Simulink and System C Case Studies

Katalin Popovici · Frédéric Rousseau
Ahmed A. Jerraya · Marilyn Wolf

Springer

Embedded Systems

Series Editors

For further volumes:
http://www.springer.com/series/8563

Katalin Popovici · Frédéric Rousseau ·
Ahmed A. Jerraya · Marilyn Wolf

Embedded Software Design and Programming of Multiprocessor System-on-Chip

Simulink and SystemC Case Studies

Springer

Katalin Popovici
MathWorks, Inc.
3 Apple Hill Dr.
Natick MA 01760
USA
katalin.popovici@mathworks.com

Frédéric Rousseau
Laboratoire TIMA
46 av. Felix Viallet
38031 Grenoble CX
France
frederic.rousseau@imag.fr

Ahmed A. Jerraya
Laboratoire TIMA
46 av. Felix Viallet
38031 Grenoble CX
France
ahmed.jerraya@cea.fr

Marilyn Wolf
Georgia Institute of Technology
Electrical & Computer
Engineering Dept.
777 Atlantic Drive NW.
Atlanta GA 30332-0250
Mail Stop 0250
USA
marilyn.wolf@ece.gatech.edu

ISBN 978-1-4419-5566-1 e-ISBN 978-1-4419-5567-8
DOI 10.1007/978-1-4419-5567-8
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009943586

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Acknowledgments

The authors would like to thank for the very useful comments of the book reviewers, which contributed a lot to improve the book, and the remarks and suggestions of all the persons for reading parts of the manuscript. We would especially like to thank Grant Martin (Tensilica Inc., USA), Tiberiu Seceleanu (ABB Corporate Research, Sweden), Soo Kwan Eo (Samsung Electronics' SoC R&D Center, Korea), Frank Schirrmeister (Synopsys Inc., USA), Lovic Gauthier (Fukuoka Laboratory for Emerging & Enabling Technology of SoC, Japan), Jason Agron (University of Arkansas, USA), Wido Kruijtzer (NXP Semiconductors, Eindhoven, Netherlands), Felice Balarin (Cadence, San Jose CA, USA), Pierre Paulin (STMicroelectronics, Ottawa, Canada), Brian Bailey (Brian Bailey Consulting, Oregon, USA).

Finally we would like to thank Mr. Charles Glaser from Springer for his wonderful cooperation in publishing this book.

Contents

List of Figures

List of Tables

Chapter 1
Embedded Systems Design: Hardware and Software Interaction

Abstract This chapter introduces the definitions of the basic concepts used in the book. The chapter details the software and hardware organization for the heterogeneous MPSoC architectures and summarizes the main steps in programming MPSoC. The software design represents an incremental process performed at four MPSoC abstraction levels (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype). At each design step, different software components are generated and verified using hardware simulation models. The overall design flow is given in this chapter. Examples of target architectures and applications, which will be used in the remaining part of this book, are described.

1.1 Introduction

Modern system-on-chip (SoC) design shows a clear trend toward integration of multiple processor cores. Current embedded applications are migrating from single processor-based systems to intensive data communication requiring multi-processing systems. The performance demanded by these applications requires the use of multi-processor architectures in a single chip (MPSoCs), endowed with complex communication infrastructures, such as hierarchical buses or networks on chips (NoCs). Additionally, heterogeneous cores are exploited to meet the tight performance and design cost constraints. This trend of building heterogeneous multi-processor SoC will be even accelerated due to current embedded application requirements. As illustrated in Fig. 1.1, the survey conducted by *Embedded Systems Design Journal* already proves that more than 50% of multi-processor architectures are heterogeneous, integrating different types of processors [159].

In fact, the literature relates mainly two kinds of organizations for multi-processor architectures. These are called shared memory and message passing [42]. This classification fixes both hardware and software organizations for each class. The shared memory organization generally assumes a multi-tasking application organized as a single software stack, and a hardware architecture made of several identical processors (CPUs), also called homogeneous symmetrical

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_1,

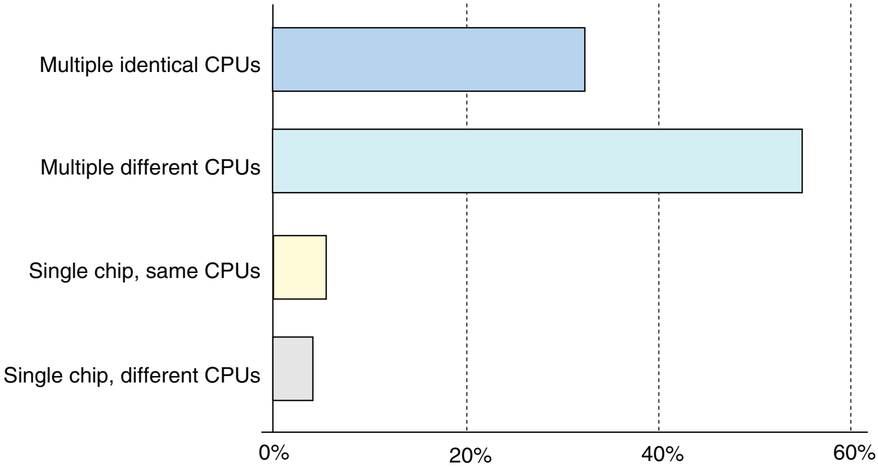


Fig. 1.1 Types of processors in SoC

multi-processing (SMP) architecture. The communication between the different CPUs is made through global shared memory. The message-passing organization assumes in most cases multiple software stacks which may run either on an SMP architecture or on non-identical processing subsystems, which may include different CPUs and/or different I/O systems, in addition to specific local memory architecture. The communication between the different subsystems is generally made through message passing. Heterogeneous MPSoCs generally combine both models, and integrate a massive number of processors on a single chip [122]. Future heterogeneous MPSoC will be made of few heterogeneous subsystems, where each subsystem includes a massive number of the same processor to run a specific software stack [87].

Nowadays multimedia and telecom applications such as MPEG 2/4, H.263/4, CDMA 2000, WCDMA, and MP3 contain heterogeneous functions that require different kinds of processing units (digital signal processor, shortly DSP, for complex computation, microcontroller for control functions, etc.) and different communication schemes (fast links, non-standard memory organization, and access). To achieve the required computation and communication performances, heterogeneous MPSoC architecture with specific communication components seems to be a promising solution [101]. Heterogeneous MPSoC includes different kinds of processors (DSP, microcontroller, ASIP, etc.) and different communication schemes. This type of heterogeneous architecture provides highly concurrent computation and flexible programmability.

Typical heterogeneous platforms already used in industry are TI OMAP [156] and ST Nomadik [114] for cellular phones, Philips Viper Nexperia [113] for consumer products, or the Atmel Diopsis D940 architecture [44]. They incorporate a DSP processor and a microcontroller, communicating via efficient, but sophisticated infrastructure.

The evolution of cell phones is a good illustration of the evolution and heterogeneity of MPSoCs. Modern cell phones may have four to eight processors, including one or more RISC processors for user interfaces, protocol stack processing, and other control functions; a DSP for video encoding and decoding and radio interface; an audio processor for music playback; a picture processor for camera options; and even a video processor for new video-on-phone capabilities. In addition, there may be other deeply embedded processors substituting for other functions traditionally designed as hardware blocks [96]. Extensible processors are proving to be flexible substitutes for hardware blocks, achieving acceptable performance and power consumption. Thus, these devices are a good example of heterogeneous MPSoC, and their demanding requirements for low cost, reasonable performance, and minimal energy consumption illustrate the advantages of using highly application-specific processors for various functions.

Heterogeneous MPSoC architectures may be represented as a set of software and hardware processing subsystems which interact via a communication network (Fig. 1.2) [42].

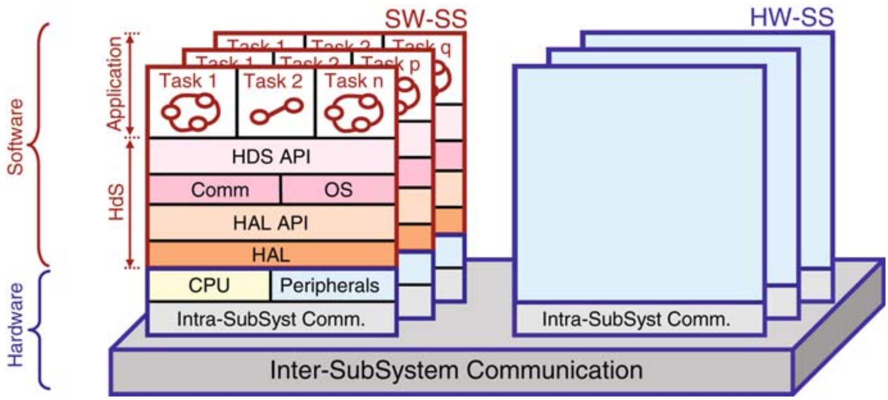


Fig. 1.2 MPSoC hardware–software architecture

A software subsystem is a programmable subsystem, namely, a processor subsystem. This integrates different hardware components including a processing unit for computation (CPU), specific local components such as local memory, data and control registers, hardware accelerators, interrupt controller, DMA engine, synchronization components such as mailbox or semaphores, and specific I/O components or other peripherals.

Each processor subsystem executes a specific software stack organized in two main layers: the application and the hardware-dependent software (HdS) layers. The application layer is associated with the high-level behavior of the heterogeneous functions composing the target application. The HdS layer is associated with the hardware-dependent low-level software behavior, such as interrupt service routines, context switch, specific I/O control, and tasks scheduling. In fact, the HdS layer includes three components: operating system (OS), specific I/O communication

(Comm), and the hardware abstraction layer (HAL). These different components are based on well-defined primitives or application programming interfaces (APIs) in order to pass from one software layer to another.

A hardware subsystem represents specific hardware component that implements specific functionalities of the application or a global memory subsystem accessible by the processing units.

The shift from the single processor to an increasingly processor- and multi-processor-centric design style poses many challenges for system architects, software and hardware designers, verification specialists, and system integrators. The main design challenges for MPSoC are as follows: programming models that are required to map application software into effective implementations, the synchronization and control of multiple concurrent tasks on multiple processor cores, debugging across multiple models of computation of MPSoC and the interaction between the system, applications, and the software views, and the processor configuration and extension [96].

Current ASIC design approaches are hard to scale to a highly parallel multi-processor SoC [88]. Designing these new systems by means of classical methods gives unacceptable realization costs and delays. This is mainly because different teams contributing to SoC design used to work separately. Traditional ASIC designers have a hardware-centric view of the system design problem. Similarly, software designers have a software-centric view. System-on-chip designs require the creation and use of radical new design methodologies because some of the key problems in SoC design lie at the boundary between hardware and software. Current SoC design process uses in most cases two separate teams working in a serial methodology to achieve hardware and software designs, while some SoC designers already adopted a process involving mixed hardware–software teams, and others try to move slowly in this direction.

The use of heterogeneous ASIPs makes heterogeneous MPSoC architectures fundamentally different from classic general-purpose multi-processor architectures. For the design of classic computers, the parallel programming concept (e.g., MPI) is used as an application programming interface (API) to abstract hardware/software interfaces during high-level specification of software applications. The application software can be simulated using an execution platform of the API (e.g., MPICH) or executed on existing multi-processor architectures that include a low-level software layer to implement the programming model. In this case, the overall performances obtained after hardware/software integration cannot be guaranteed and will depend on the match between the application and the platform.

Unlike classic computers, the design of MPSoC requires a better matching between hardware and software in order to meet performance requirements. In this case, the hardware/software interfaces implementation is not standard; it needs to be customized for a specific application in order to get the required performances. This includes customizing the CPUs and all the peripherals required to accelerate communication and computation. In most cases, even the lower software layers need to be customized to reach the required cost and performance constraints. Applying the classical design schemes for those architectures leads to inefficient designs.

Additionally, classic SoC design flows imply a long design cycle. Most of these flows rely on a sequential approach where complete hardware architecture should first be developed before software could be designed on top of it. This long design cycle is not acceptable because of time to market constraints. There is an increasing use of early system-level modeling, even if it would not contain the entire hardware architecture, but only a subset of components which are sufficient to allow some level of software verification on the hardware before the full hardware is available, thus reducing the sequential nature of the design methodology. The use of high-level programming model to abstract hardware/software interfaces is the key enabler for concurrent hardware and software designs. This abstraction allows to separate low-level implementation issues from high-level application programming. It also smoothes the design flow and eases the interaction between hardware and software designers. It acts as a contract between hardware and software teams that may work concurrently. Additionally, this scheme eases the integration phase since both hardware and software have been developed to comply with a well-defined interface. The use of a parallel programming model allows reducing the overall system design time and cost in addition to a better handling of complexity.

The use of programming models for the design of heterogeneous MPSoC requires the definition of new design automation methods to enable concurrent design of hardware and software. This will also require new models to deal with non-standard application-specific hardware/software interfaces at several abstraction levels.

In order to allow for concurrent hardware/software design, as shown in Fig. 1.3, we need abstract models of both software and hardware components. In general-purpose computer design, system designers must also consider both hardware and software, but the two are generally more loosely coupled than in SoC design. As a result, general-purpose computer systems generally model the hardware/software interfaces twice. Hardware designers use a hardware/software interface model to test their hardware design and software designers use a hardware/software interface model to validate the functionality of their software. Using two separate models induces a discontinuity between hardware and software. The result is not only a waste of design time but also a less efficient and lower quality hardware and software. This overhead in cost and loss in efficiency are not acceptable for SoC design. A single hardware/software interface needs to be shared between both hardware and software designers.

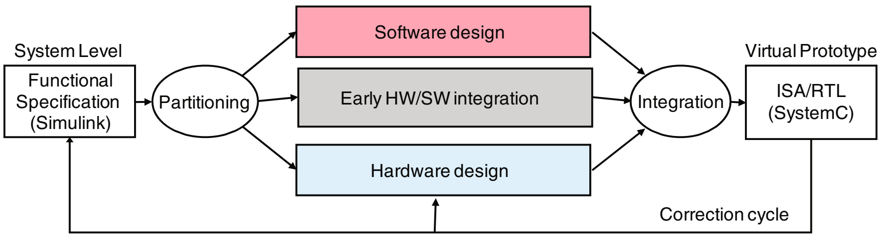


Fig. 1.3 System-level design flow

Figure 1.3 shows a simplified flow of mixed hardware/software design, where both software and hardware are designed concurrently. This flow starts with a system-level specification made of application functions using a system-level parallel programming model. This may be a Simulink functional model that can be simulated using the corresponding environment. Then, the application functions are partitioned in either hardware or software target implementations, followed by concurrent hardware and software designs. The hardware design produces RTL (register transfer level) or gate model of the hardware components often represented using SystemC language or a hardware description language like VHDL and Verilog. The software design can be performed at higher level of abstraction and it produces the binary code of the software components. The final integration step consists of verification of the whole system by co-simulating the RTL hardware model with the binary software code.

Programming the application-specific heterogeneous multi-processor architectures becomes one of the key issues for MPSoC, because of two contradictory requirements: (1) reducing software development cost and overall design time requires a higher level programming model. This reduces the amount of architecture details that need to be handled by application software designers and then speed up the design process. The use of higher level programming model will also allow concurrent software/hardware design and thus reduces the overall design time. (2) Improving the performance of the overall system requires finding the best matches between hardware and software. This is generally obtained through low-level programming.

Therefore, for this kind of architectures, classic programming environments do not fit: (i) high-level programming does not handle efficiently specific I/O and communication schemes, while (ii) low-level programming explicitly managing specific I/O and communication is a time-consuming and error-prone activity. In practice, programming these heterogeneous architectures is done by developing separate low-level codes for the different processors, with late global validation of the overall application with the hardware platform. The validation can be performed only when all the binary software is produced and can be executed on the hardware platform.

Next-generation programming environments need to combine the high-level programming models with the low-level details. The different types of processors execute different software stacks. Thus, an additional difficulty is to debug and validate the lower software layers required to fully map the high-level application code on the target heterogeneous architecture [125].

This book gives an overview of concepts, tools, and design steps to systematic embedded software design for the MPSoC architectures. The book combines Simulink for high-level programming and SystemC for the low-level software development. The software design and validation is performed gradually through four different software abstraction levels (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype). Specific software execution models or abstract architecture models are used to allow debugging the different software components with explicit hardware–software interaction at each abstraction level.

The book is organized as follows: Chapter 1 introduces the context of MPSoC design, the difficulties of programming these complex architectures, the design and validation flow of the multiple software stacks running on the different processor subsystems, the adopted MPSoC abstraction levels, and the definition of some concepts later used in this book. Chapter 2 defines first the hardware components of the MPSoC architecture, i.e., processor, memory, and interconnect and then, the components of the embedded software running on top of these architectures, i.e., operating system, communication, and middleware and hardware abstraction layers. Chapters 3, 4, 5, and 6 detail the embedded software design and validation for MPSoC at four abstraction levels, namely, the system architecture, virtual architecture, transaction-accurate architecture, respectively, the virtual prototype design. Chapter 7 draws conclusions and indicates several future research perspectives for embedded software design.

1.2 From Simple Compiler to Software Design for MPSoC

The software compilation is a common concept of both electronic and informatic domains. Usually the applications are implemented in high-level programming languages, such as C/C++. The software compilation represents the translation of a sequence of instructions written in a higher symbolic language into a machine language before the instructions can be executed. Typical situation is the translation of an application from a high-level language like C to the assembly language accepted by processor which will execute that application.

The compilation contains the following steps [2]:

- Lexical analysis, which divides the source code text into small pieces, called tokens. Each token is a single atomic unit of the language, for instance, a keyword, identifier, or symbolic name. The token syntax is often a regular expression. This phase is also called lexing or scanning, and the software doing the lexical analysis is called lexical analyzer or scanner.
- Syntax analysis, which parses the token sequence and builds an intermediate representation, for instance, in the form of a tree. The tree is built according to the rules of the formal grammar which defines the language syntax. The nodes of the parse tree represent elementary operations and operators, while the arcs symbolize the dependencies between the nodes.
- Semantic analysis, which adds semantic information to the parse tree and builds the symbol table. The symbol table is a data structure, where each identifier in a program's source code is associated with information relating to its declaration and appearance in the source, such as type, scope, and sometimes its location. This phase also performs semantic checks, such as type checking (checking for type errors) or object binding (associating variable and function references with their definition).

– Optimization, which transforms the intermediate parse tree into functionally equivalent, but faster or smaller forms. Examples of optimizations are inline expansions, dead code elimination, constant propagation, register allocation, or automatic parallelization.
– Code generation, which traverses the intermediate tree and generates the code in the targeted language corresponding to each node of the tree. This also involves resource and storage decisions, such as deciding which variables to fit into the registers and memory, and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

Figure 1.4 illustrates these steps in case of a C code compilation to the host processor-specific assembly language. The first phases of the compilation depend only on the input language and they are called front end of the compilation. The optimization and generation of the code depends only on the targeted language and it is also known as back end of the compilation. Usually, the compilation to the assembly language of the host processor includes also a linking phase. The linker associates an address to each object symbol of the assembly code, in order to be loaded in the memory of the processor for execution.

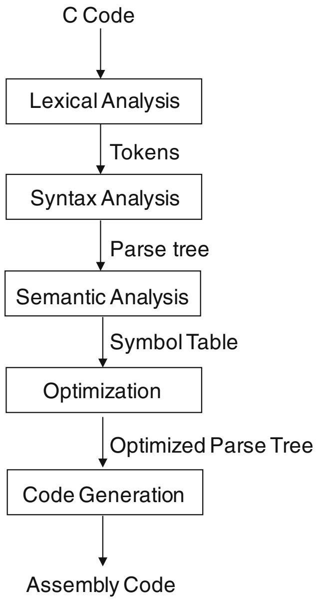


Fig. 1.4 Software compilation steps

The software design for MPSoC is more complex than a simple software compilation. The software design represents the process of producing executable software in the form of a binary code, for a specific architecture, from a high-level application representation (e.g., UML [161], C, or C++). The software design refines the application representation and adapts it to the target architecture in order to produce a compatible and efficient executable code, e.g., parallelization of the application, communication specification. The compilation is the final phase of the software design.

An ideal software design flow allows the software developer to implement the application in a high-level language, without considering the low-level architecture details. In an ideal design flow, the software generation targeting a specific architecture consists of a set of automatic steps, such as application partitioning and mapping on the processing units provided by the targeted architecture, final application software code generation, and hardware-dependent software (HdS) code generation (Fig. 1.5a).

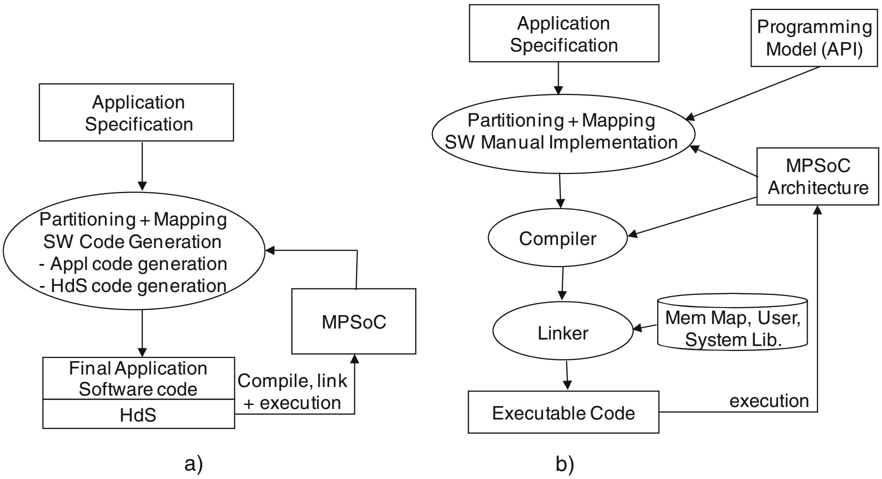


Fig. 1.5 Software design flows: (**a**) ideal software design flow and (**b**) classic software design flow

The HdS is made of lower software layers that may incorporate an operating system (OS), communication management, and a hardware abstraction layer to allow the OS functions to access the hardware resources of the platform. Ideally, the software design should support any type of application description, independently of the programming style, and it should target any type of SoC architecture. Unfortunately, we are still missing such an ideal generic flow, able to map efficiently high-level programs on heterogeneous MPSoC architectures. Additionally, the validation and debugging of HdS remains the main bottleneck in MPSoC design [171] because each processor subsystem requires specific HdS implementation to be efficient.

The classical approaches for the software design use programming models to abstract the hardware architecture (Fig. 1.5b). These generally induce discontinuities in the software design, i.e., the software compiler ignores the processor architecture (e.g., interrupts or specific I/Os). To produce efficient code, the software needs to be adapted to the target architecture by using specific libraries, such as system library for the different hardware components or specific memory mapping for the different CPU and memory architectures.

The software adaptation for a specific MPSoC architecture, in order to obtain an efficient executable code, requires the following information:

– Hardware architecture details: type of processors, type of memories, type of peripherals, etc.
– Memory mapping, more precisely the different memory addresses reserved to various hardware and software components, e.g., memory-mapped address of an I/O device.
– Diverse constraints imposed by the execution environment, such as timing constraints (e.g., execution deadline, data debit), surface constraints (e.g., limited memory resources), power consumption constraints, or other constraints specific to the architecture.

This kind of information can be specified during the software design in several ways: in the form of architecture parameters manually annotated in the application specification, automatically deduced from the specification structure, or they might be given in a natural language.

The software design is not only a very complex process due to the hardware architecture variety and complexity but also the different types of knowledge required by a successful design.

The variety of MPSoC architectures is mainly determined by the heterogeneity of the processors and the combination of the various communication schemes. The semiconductor industry provides many types of processors, which do not share the instruction set architecture (ISA). Employing processor-specific compiler for the assembly code generation does not seem to reduce totally the difficulties induced by the processors diversity in the software design. Examples of processor characteristics which make difficult the software to be adapted by the compiler for the target architecture are as follows:

– *Data type*: each processor usually provides preferable data types that can be efficiently utilized. They depend on the size of its local registers, bit size of the data path, and memory access routes. For performance reasons, it is strongly recommended to use these data types for most of the application variables. Since different kinds of processors do exist, the preferable data type can be integer (int) of 8 bits, 16 bits, or 32 bits, or even more sophisticated data types depending on the internal architecture of the processor. The C language uses a generic integer (int) type, and then the compiler decides the number of bits allocated for the variable, depending on the target processor (8 bits, 16 bits, 32 bits, etc.). If the data need to be exchanged between multiple processors, the data types have to be identical at both producer and consumer sides. This increases the software design complexity, if the producer and consumer processors have different preferable data types. But a robust API can help dealing with data type conversion between heterogeneous processors.
– *Data representation*: the data are stored in the memories in the form of packets of bits. But there are many ways of interpreting these bits (e.g., two's complement,

exponential representation). An important aspect of the processor's architecture is the endianness. The endianness is the way of ordering the bytes in the memory to represent a data. Mainly, the architectures are divided into two categories: *big endian* (most significant byte first, stored at the lowest memory address) and *little endian* (increasing byte numeric significance with increasing memory addresses). Additionally, the same data type, e.g., 32 bits, can be represented in both types of endianness. Byte order is an important consideration in multi-processor architectures, since two processors with different byte orders may be communicating.
– *Instruction set*: each type of processor is characterized by a specific instruction set. The compiler is responsible to translate the high-level application into the instruction set interpretable by the processor. Generally, the high-level description does not take into consideration the hardware characteristics to attain performances. But, sometimes it is desirable to optimize the application algorithm for a particular processor or to use processor-specific instructions in the high-level representation, e.g., instructions to control the power consumption.
– *Interrupts*: most of the processors provide interrupt mechanism to control the events occurred during the computation. Even if the interrupt mechanisms are very specific to each type of processor and they can be very complex, the compilers do not take them into consideration during the assembly code generation.

All these different features among the processors cannot be handled only by the compilers.

Besides the processor characteristics, the architecture heterogeneity is amplified also by the variety of communication schemes between the processors. Current multi-processor systems on chip (MPSoC) architectures integrate a massive number of processors which range from 2 to 20+ and scaling up to 100 processors in a multi-tile-based architecture. The processors can exchange application and synchronization data in different ways [171]. The communication architecture is characterized by a large set of parameters and adopted design choices, such as

– programming model: shared memory (e.g., OpenMP [33]), message passing (e.g., MPI [MPI])
– blocking versus non-blocking semantic
– synchronous versus asynchronous communication
– buffered versus unbuffered data transfer
– synchronization mechanism, such as interrupt or polling
– type of connection: point-to-point dedicated link or global interconnection component, such as system bus or network on chip (NoC)
– communication buffer mapping: stored in the sender subsystem, stored in the receiver subsystem, or using a dedicated storage resource such as global memory or hardware FIFO
– direct memory access (DMA)

All these different characteristics can be combined in multiple ways, thus making the software design more difficult. Initially, the software uses high-level primitives in order to abstract all these architecture details. During the design, several implementations are provided to these primitives in order to map the high-level software onto the hardware architecture. Since the hardware architecture allows numerous configuration schemes which can be explored (e.g., diverse communication schemes), the software design includes several iteration steps until the required performances are achieved. Moreover, the software design requires copious competences in large domains, such as processors knowledge, communication protocol knowledge, application knowledge, architecture knowledge, hardware/software interface knowledge.

The processors knowledge includes the following types of required information:

- Number and size of the local registers, size of the data bus, size of the address bus, etc., in order to better suite the data types
- Data transfer mode of the processor: does it use a common data/program memory (Von Neuman) or distinct (Harvard), what type of protocol is used by the processor to read/write data, or what type of interrupt mechanism is used
- Assembly language of the processor used to implement the algorithm code optimizations, the processor-specific interrupt service routines, and the context switch in the HdS code
- Processor performances: the CPU speed or the number of clock cycles required to load/store data in the memory. This type of information helps to better choose the parallelization way of the application algorithm
- Type of processor architecture (pipeline, RISC, CISC, etc.) to better implement the application algorithm.
- Type of data transfer, with or without initiating a DMA transfer request. With DMA, the CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done. This is especially useful in real-time computing applications where not stalling behind concurrent operations is critical. Another and related application area is various forms of stream processing where it is essential to have data processing and transfer in parallel, in order to achieve sufficient throughput.

The communication protocol knowledge include knowledge about the communication protocol or the communication implementation, e.g., whether it is managed by the operating system or whether it is managed by the hardware, with or without DMA engine.

The knowledge about the application include the description language (e.g., C, C++, UML, Simulink), the application algorithm to know how to optimize its implementation, and the application parameters, such as number of variables to be exchanged, number of possible tasks executed in parallel.

The knowledge about the hardware/software interface is important to better adapt the software to the hardware, more precisely to better select the operating system responsible with the scheduling of the parallel executed tasks, the implementation of the communication in the software for the tasks running on the same CPU, etc.

1.3 MPSoC Programming Steps

Programming an MPSoC means to generate software running on the MPSoC efficiently by using the available resources of the architecture for communication and synchronization. This concerns two aspects: software stack generation and validation for the MPSoC, and communication mapping on the available hardware communication resources and validation for MPSoC.

Efficient programming requires the use of the characteristics of the architecture. For instance, a data exchange between two tasks mapped on different processors may use different schemes through either the shared memory or the local memory of one of these processors. Additionally, different synchronization schemes (polling and interrupts) may be used to coordinate this exchange. Furthermore, the data transfer between the processors can be performed by a DMA engine, thus permitting the CPU to execute other computation, or by the CPU itself. Each of these communication schemes has advantages and disadvantages in terms of performance (latency, throughput), resource sharing (multi-tasking, parallel I/O), and communication overhead (memory size, execution time). The ideal scheme would be able to produce an efficient software code starting from high-level program using generic communication primitives.

As shown in Fig. 1.6, the software design flow starts with an application and an abstract architecture specification. The application is made of a set of functions. The architecture specification represents the global view of the architecture, composed of several hardware and software subsystems.

The main steps in programming the MPSoC architecture are as follows:

– Partitioning and mapping the application onto the target architecture subsystems
– Mapping application communication on the available hardware communication resources of the architecture
– Software adaptation to specific hardware communication protocol implementation
– Software adaptation to detailed architecture implementation (specific processors and memory architecture)

The result of each of these four phases represents a step in the software and communication refinement process. The refinement is an incremental process. At each stage, additional software component and communication architecture details are integrated with the previously generated and verified components. This conducts to a gradual transformation of a high-level representation with abstract components into a concrete low-level executable software code. The transformation has to be validated at each design step. The validation is performed by formal analysis, simulation, or combining simulation with formal analysis [82].

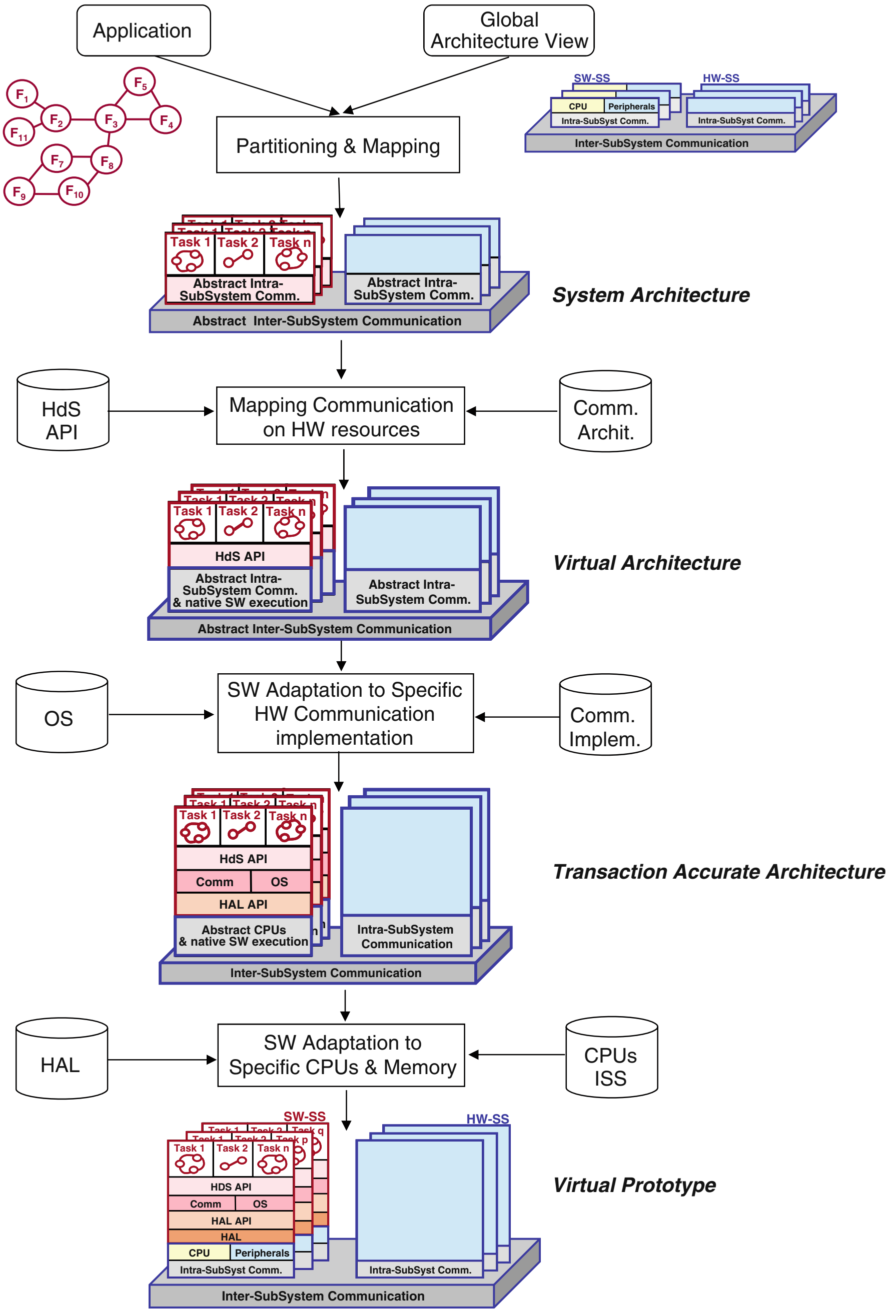


Fig. 1.6 MPSoC programming steps

The formal verification is defined by the Wiktionary as being "the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics." Generally, the formal verification is performed using the model checking technique, which consists of a systematically exhaustive exploration of the mathematical model that corresponds to the hardware and software architecture. The exploration of the mathematical model consists of exploring all states and transitions in the model. Usually, the model checking is applied to verify the structure of a hardware design. In case of the software architecture, the model checking is used to prove or disprove a given property characterizing the software. There are several academic and commercial tools developed for model checking. The Incisive Formal Verifier from Cadence is an example of tool for verification, which provides a formal means of verifying RTL functional correctness with assertions [29]. The formal analysis does not require a set of test vectors for the verification and supports the verification of the RTL description in several design languages, including Verilog, SystemVerilog, or VHDL. Another example of model checking tool is the Kronos from Verimag [165], which supports the verification of complex real-time systems, whose components are modeled as timed automata and the correctness requirements are expressed in real-time temporal logic TCTL. The timed automata is an automata extended with a finite set of real-valued clocks, used to express timing constraints. Clocks can be set to zero and their values increase uniformly with time. In this kind of automata, a transition is enabled only if the timing constraint associated with it is satisfied by the current values of the clocks. Many MPSoC design projects use no formal validation at all, except perhaps equivalence checking from RTL to gates and between physical design steps such as insertion of scan.

In the following, we will consider simulation-based validation to ensure that the system behavior respects the initial specification. The simulation-based validation requires the software execution using an executable model.

During the *partitioning and mapping* of the application on the target architecture, the relationship between application and architecture is defined. This refers to the number of application tasks that can be executed in parallel, the granularity of these tasks (coarse grain or fine grain), and the association between tasks and the processors that will execute them. The result of this step is the decomposition of the application into tasks and the correspondence tasks–processors [154]. This step is also called *system architecture design* and the resulting model is the system architecture model.

The system architecture model represents a functional description of the application specification, combined with the partitioning and mapping information. Aspects related to the architecture model (e.g., processing units available in the target hardware platform) are combined into the application model (i.e., multiple tasks executed on the processing units). Thus, the system architecture model expresses parallelism in the target application through capturing the mapping of the functions into tasks and the tasks into subsystems. It also makes explicit the communication units to abstract the intra-subsystem communication protocols (the communication between the tasks inside a subsystem) and the inter-subsystem communication protocols (the communication between different subsystems).

The second step implements the *mapping of communication* onto the hardware platform resources. At this phase, the different links used for the communication between the different tasks are mapped on the hardware resources available in the architecture to implement the specified protocol. For example, a FIFO communication unit can be mapped to a hardware queue, a shared memory, or some kind of bus-based device. The task code is adapted to the communication mechanism through the use of adequate HdS communication primitives. This step is also entitled *virtual architecture design* and the resulting model is named virtual architecture model.

The next step of the proposed flow consists of *software adaptation to specific communication protocol implementation*. During this stage, aspects related to the communication protocol are detailed, for example, the synchronization mechanism between the different processors running in parallel becomes explicit. The software code has to be adapted to the synchronization method, such as events or semaphores. This can be done by using the services of OS and communication components of the software stack. In general, where an OS is referred in the MPSoC context, it might be assumed to be a heavyweight OS such as Linux. However, often the operating systems, e.g., in portable devices, are much lighter weight aggregates of the communication primitives and perhaps simple scheduling mechanisms, and they are specific for a single application device or for a small family, but not a commercial OS or RTOS. The phase of integrating the OS and communication is also named *transaction-accurate architecture design* and the resulting model is the transaction-accurate architecture model.

The last step corresponds to the *specific adaptation of the software to the target processors and the specific memory mapping*. This includes the integration of the processor-dependent software code into the software stack (HAL) to allow low-level access to the hardware resources and the final memory mapping. This step is also known as *virtual prototype design* and the resulting model is called virtual prototype model.

These different steps of the global flow correspond to different software components generation and validation at different abstraction levels, as it will be described in the following paragraphs.

1.4 Hardware/Software Abstraction Levels

The structured model of the software stack representation allows generation and validation of the different software components separately [87]. The different components and layers of the software stack correspond to different abstraction levels. The debug of this software stack made of several components is one of the MPSoC current design challenges [96].

In order to verify the software, an execution model is required at each abstraction level to allow debugging the specific software component. The *execution model* represents an abstract architecture model [133] which allows simulating and validating the software component at each abstraction level. The *execution model* makes use of a *software development platform*, which is the result of abstracting different components of the target hardware architecture. This abstract architecture model hides details of the underlying implementation of the hardware platform, but ensures a sufficient level of control that the software code can be validated in terms of performance, efficiency, and reliable functionality.

As illustrated in Fig. 1.7, the software development platform is an abstract model of the architecture in form of a run-time library or simulator aimed to execute the software (e.g., instruction set simulator) [43, 95]. The combination of this platform with the software code produces an executable model that emulates the execution of the final system including hardware and software architecture. This executable model allows the simulation of the software with detailed hardware–software interaction, software debug, and eventually performance measurement. Generic software development platforms have been designed to fully abstract the hardware–software interfaces, i.e., MPITCH is a run-time execution environment designed to execute parallel software code written using MPI (message-passing interface) [22]. MPICH provides an implementation of the MPI standard library for message passing that combines portability with high performance. Examples of MPI primitives are *MPI_send(...)*, *MPI_Bsend(...)*, *MPI_Buffer_attach(...)*, *MPI_Recv(...)*, *MPI_Bcast(...)*. In fact, the platform and the software may be combined using different schemes.

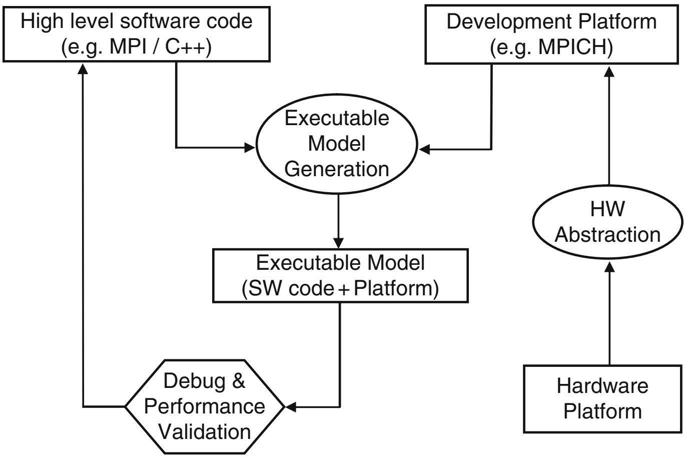


Fig. 1.7 Software development platform

Traditional software development strategies make use of generic software development platforms. But the generic platforms do not allow simulating the software execution with detailed hardware–software interaction and, therefore, they do not allow accurate performance measurement. Additionally, since the hardware–software interfaces are fully abstracted, the generic platforms cannot be used to debug the lower layers of the software stack, e.g., the RTOS (real-time operating system) and the implementation of the high-level communication primitives.

Thus, several architecture and application-specific software development platforms are required for the validation of the various software components at different abstraction levels.

The software validation and debug is performed by execution of the software code on a corresponding execution model. The debug is an iterative process because the different software components need different detail levels in order to be validated. For example, the debug of the application tasks does not need explicit implementation of the synchronization protocol between the processors using mailboxes in the development platform, while the debug of the integration of the tasks code with the OS requires this kind of detail. The detailed hardware–software interaction allows debugging this low-level architecture-specific software code. All these requirements are considered during the abstraction of the architecture at each design step to build the executable model. Thus, depending on the software component to be validated (application tasks code, tasks code execution upon an OS, HAL integration in the software stack) the platform may model only a subset of hardware components, more precisely those components that are required for the software validation. The rest of the hardware components, which are not relevant for the software validation, are abstracted.

The debug of the software is performed by simulation at the different abstraction levels. Thus, the *system architecture model* simulation is used to debug the application algorithm. The *virtual architecture model* simulation serves to debug the final application tasks code. The *transaction-accurate architecture model simulation* is used to debug the glue between the application tasks code and the OS and the communication libraries. The *virtual prototype model* uses instruction set simulators to execute and debug the full software stack, including final binary and memory mapping. In practice, the various levels of debugging are not used by all design teams, although many use one or two, i.e., the system architecture model is useful for algorithm developers, who implement new algorithms or optimize existing ones; but it can also serve as a functional specification of the design requirements. The virtual architecture and the transaction-accurate architecture models are useful for system architects, who mostly determine the hardware–software partitioning but they do not require accurate results for the performance estimation or for embedded software developers who integrate the applications with the OS. The virtual prototype level is useful not only for device drivers development, architecture exploration, but also for hardware designers to verify their VHDL design through stimuli and test vectors by means of co-simulation with the platform.

At all these abstraction levels, the debug process uses standard debugging tools and environments, such as GNU debuggers [61] or trace waveforms during the simulation, such as SystemC waveforms [SystemC].

1.4.1 The Concept of Hardware/Software Interface

The hardware/software interface links the software part with the hardware part of the system. As illustrated in Fig. 1.8, the hardware/software interface needs to

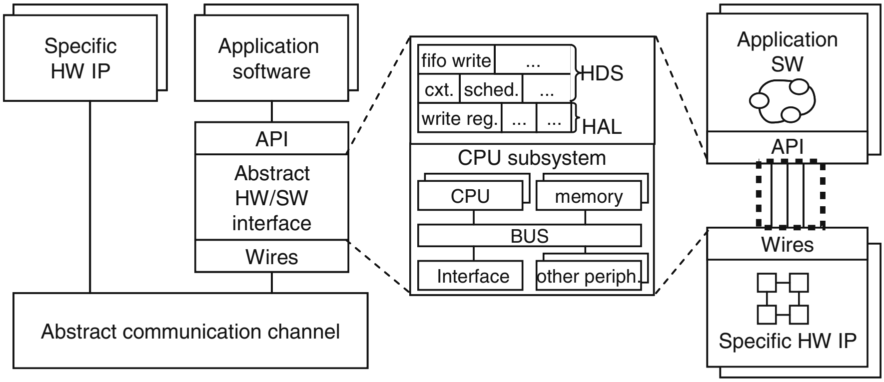


Fig. 1.8 Hardware/software interface

handle two different interfaces: one on the software side using APIs and one on the hardware side using wires [24]. This heterogeneity makes the hardware/software interface design very difficult and time-consuming because the design requires both hardware and software knowledge and their interaction [86].

The hardware/software interface has different views depending on the designer. Thus, for an application software designer, the hardware/software interface represents a set of system calls used to hide the underlying execution platform, also called programming model. For a hardware designer, the hardware/software interface represents a set of registers, control signals, and more sophisticated adaptors to link the processor to the hardware subsystems. This can be in the form of a register description language (RDL), which allows to specify and implement software-accessible hardware registers and memories, or an XML description, like the standard IP-XACT proposed by the Spirit consortium [146] to ensure interoperability between hardware components provided by different vendors, which was already adopted in the Socrates chip integration platform [45]. For a system software designer, the hardware/software interface is defined as the low-level software implementation of the programming model for a given hardware architecture. In this case, the processor is the ultimate hardware–software interface. This scheme is a sequential scheme assuming that the hardware architecture is ready to start the low-level software design. Finally, for a SoC designer the hardware/software interface abstracts both hardware and software in addition to the processor.

The design of the hardware/software interface is a complex and time-consuming task. The authors in [137] propose a unified model to represent the hardware/software interfaces, called service dependency graph, shortly SDG. This model is based on the concept of services. Thus, the interface is specified by a set of requiring and providing services. This kind of interface modeling has the following goals: it allows handling heterogeneous components at several abstraction levels, being independent of specific modeling standards, it hides implementation details and allows delaying the implementation decisions through the use of abstract architecture models, it allows different and sophisticated adaptation schemes, and it makes possible the automation of the design for application-specific interfaces and/or target architectures. Thus, the approach proposed in [137] supports automatic generation of the hardware/software interfaces based on the service and resource requirements described using an SDG.

Whether the hardware/software interface is designed automatically or manually, the designer needs to fix all the implementation parameters, such as address map, interrupt priorities, software stack size. Before obtaining the final code, the designer may need to find and fix several bugs that may occur during the implementation. Generally, the bugs in the hardware/software interface design are due to incorrect configuration and access to the platform resources or misunderstanding by the designer of the hardware architecture. An example of such kind of situation is the wrong configuration of the memory map for the registers of the interrupt controller. During the MPSoC design conducted by the authors in [175] for the MPEG4 video encoder application, 78% of the total bugs were found in the hardware/software interfaces. Examples of such kind of bugs are as follows:

- Processor booting bugs, when the booting is not synchronized among the various processors
- Bugs in the kernel of the real-time operating system, more precisely, bugs due to wrong interrupt priority-level assignments, missed interrupts, or improper functionality of the context switch service to resume the new task
- Bugs found in the high-level programming model, such as incorrect FIFO configuration which produces communication deadlock between the tasks
- Bugs found in the hardware management, such as wrong memory map assignment.

Thus, a gradual validation of the hardware/software interface to guarantee correct functionality becomes trivial. The hardware/software interface requires handling many software and hardware architecture parameters. To allow the gradual validation of the software stack, the hardware–software interface needs to be described at the different abstraction levels.

1.4.2 *Software Execution Models with Abstract Hardware/Software Interfaces*

Figure 1.9 illustrates the software execution models at different abstraction levels for a simplified application made of three tasks (T1, T2, and T3) that need to be mapped on an architecture made of two processing units and several memory hardware subsystems.

For each level, Fig. 1.9 shows the software organization, the hardware–software interface and the execution model that will be used to verify the software component at the corresponding abstraction level. The key differentiation between these levels

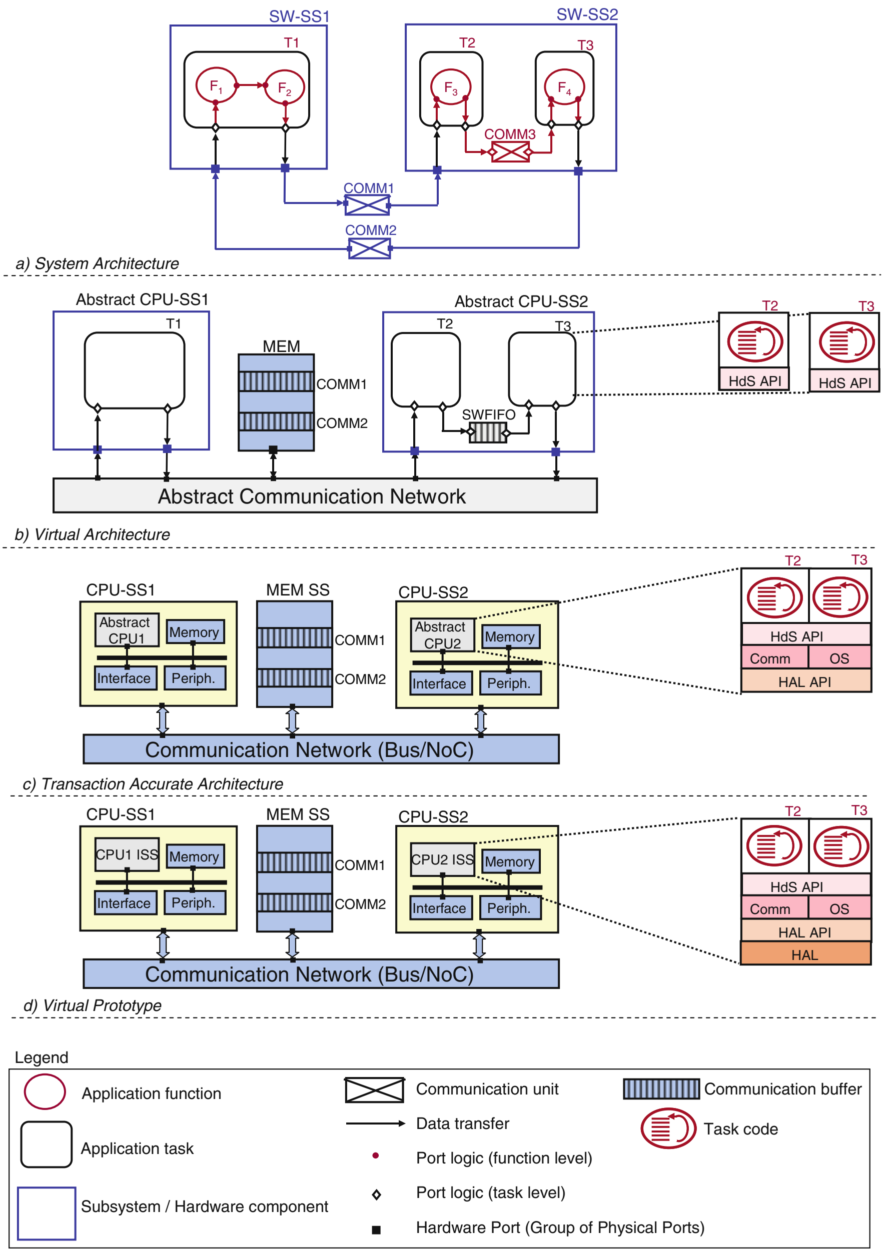


Fig. 1.9 Software execution models at different abstraction levels

is the way of specifying the hardware–software interfaces and the communication mechanism implementation.

The highest level is the *system architecture* level (Fig. 1.9a). In this case, the software is made of a set of functions grouped into tasks. The function is an abstract view of the behavior of an aspect of the application. Several tasks may be mapped on the same software subsystem. The communication between functions, tasks, and subsystems makes use of abstract communication links, e.g., standard Simulink links or explicit communication units that correspond to specific communication paths of the target platform. The links and units are annotated with communication mapping information. The corresponding execution model consists of the set of the abstract subsystems. The simulation at this level allows validation of the application's functionality. This model captures both the application and the architecture in addition to the computation and communication mapping.

Figure 1.9a shows the system architecture model with the following symbols: circles for the functions, rounded rectangular to represent the task, rectangular for the subsystem, crossed rectangular for the communication units between the tasks, filled circles for the ports of the functions, diamonds for the logic ports of the tasks, and filled rectangular for group of hardware ports. The dataflow is illustrated by unidirectional arrows.

In this case, the system is made of two abstract software subsystems (SW-SS1 and SW-SS2) and two inter-subsystem communication units (COMM1 and COMM2). The SW-SS1 software subsystem encapsulates task T1, while the subsystem SW-SS2 groups together tasks T2 and T3. The intra-subsystem communication between the tasks T2 and T3 inside SW-SS1 is performed through the communication unit COMM3.

The next abstract level is called *virtual architecture* level (Fig. 1.9b). The hardware–software interfaces are abstracted using HdS API that hides the OS and the communication layers. The application code is refined into tasks that interact with the environment using explicit primitives of the HdS API. Each task represents a sequential C code using a static scheduling of the initial application functions. This code is the final application code that will constitute the top layer of the software stacks. The communication primitives of the HdS API access explicit communication components. Each data transfer specifies an end-to-end communication path. For example, the functional primitives *send_mem(ch,src,size)/recv_mem(ch,dst,size)* may be used to transfer data between the two processors using a global memory connected to the system bus, where *ch* represents the communication channel used for the data transfer, *src/dst* the source/destination buffer, and *size* the number of words to be exchanged. The communication buffers are mapped on explicit hardware resources.

At the virtual architecture level, the software is executed using an abstract model of the hardware architecture that provides an emulation of the HdS API. The software execution model is comprised of these abstract subsystems, explicit interconnection component, and storage resources. During the simulation at the virtual architecture level, the software tasks are scheduled by the hardware platform since the final OS is not yet defined. The simulation at this level allows validation of the final code of tasks and may give useful statistics about the communication requirements. The virtual architecture is message accurate in terms of data exchange between the different tasks. Thanks to the HdS APIs, the tasks code remains unchanged for the following levels. In this book, the virtual architecture platform is considered as a SystemC model where the software tasks are executed as SystemC threads.

In the example illustrated in Fig. 1.9b, the system is made of two abstract processor subsystems (CPU1-SS and CPU2-SS) and a global memory (MEM) interconnected through an abstract communication network. The communication units *comm1* and *comm2* are mapped on the global memory and the communication unit *comm3* becomes a software fifo (swfifo).

The next level is called the *transaction-accurate architecture* level (Fig. 1.9c). At this level, the hardware–software interfaces are abstracted using a HAL API that hides the processor's architecture. The code of the software task is linked with an explicit OS and specific I/O software implementation to access the communication units. The resulting software makes use of hardware abstraction layer primitives (HAL_API) to access the hardware resources. This will constitute the final code of the two top layers of the resulting software stack. The data transfers use explicit addresses, e.g., *read_mem(addr, dst, size)/ write_mem(addr, src, size),* where *addr* represents the source, respectively, the destination address, *src/dst* represents the local address, and *size* the size of the data.

The software is executed using a more detailed development platform to emulate the network component, the explicit peripherals used by the HAL API, and an abstract computation model of the processor. During the simulation at this level, the software tasks are scheduled by the final OS, while the communication between tasks mapped on the same processor is also implemented by the OS. The simulation at this level allows validating the integration of the application with the OS and the communication layer. It may also provide precise information about the communication performances. The accuracy of the performance estimation is transaction-accurate level. In this book, the transaction-accurate architecture is generated as a SystemC model where the software stacks are executed as external processes communicating with the SystemC simulator through the IPC layer of the Linux OS running on the host machine.

In the example illustrated in Fig. 1.9c, the system is made of the two processor subsystems (CPU1-SS and CPU2-SS) and the global memory subsystem (MEM-SS) interconnected through an explicit communication network (bus or NoC). Each processor subsystem includes an abstract execution model of the processor core (CPU1, respectively, CPU2), local memory, interface, and other peripherals. Each processor subsystem executes a software stack made of the application tasks code, communication, and OS layers.

Finally, the HAL API and processor are implemented through the use of a HAL software layer and the corresponding processor part for each software subsystem. This represents the *virtual prototype* level (Fig. 1.9d). At the virtual prototype level the communication consists of physical I/Os, e.g., *load/store*. The platform includes all the hardware components such as cache memories or scratch pads. The scheduling of the communication and computation activities for the processors becomes explicit. The simulation at this level allows cycle-accurate performance validation and it corresponds to classical hardware/software co-simulation models with instruction set simulators [110, 134, 140] for the processors and RTL components or cycle-accurate TLM components for the hardware resources.

In the example illustrated in Fig. 1.9d, the two processor subsystems (CPU1-SS and CPU2-SS) include ISS for the execution of the software stack corresponding to CPU1, respectively, CPU2. Each processor subsystem executes a software stack made of the application tasks code, communication, OS, and HAL layers.

In order to verify the software during the different design steps, different execution models are used adapted to each software abstraction level. In the rest of the book, we use Simulink for the initial simulation at system architecture level, while for all others we use SystemC design language.

1.5 The Concept of Mixed Architecture/Application Model

The following paragraphs give the definition of the mixed architecture/application model and describe the execution scheme that allows simulating this model in Simulink, respectively, in SystemC.

1.5.1 Definition of the Mixed Architecture/Application Model

The architecture and application specifications can be combined in a mixed hardware/software model where the software tasks are mapped on the processor subsystems. This mixed hardware/software representation can be modeled by abstracting the processor subsystems and communication topology. The processor subsystems are substituted by abstract subsystem models, while the communication is described using an abstract communication platform. The result is a mixed architecture/application model, named also mixed hardware/software model. The mixed architecture/application concept allows modeling heterogeneous MPSoC at different abstraction levels, independent from the description language used by the designer. The mixed hardware/software model is also called combined algorithm/architecture model [21].

The combined algorithm/architecture model comprises a set of significant advantages:

- It captures the behavior of the architecture and the algorithm and the interconnection between them. This allows to build a correct system, which ensures the good functionality of the application and the architecture running together.
- It avoids inconsistency and errors and it helps to ensure completeness of the specifications. The execution of the combined architecture/application model consists of the realization of a model that behaves in the same way as the global system.

– It allows validating, simulating, and debugging the system functionality, even before the implementation of the final hardware architecture.
– It helps to build early performance models of the global hardware/software architecture and to validate the overall performances.
– It allows the use of several test bench data at input for the global system in order to verify the functionality. Thus, different functionality scenarios can be tested, even before the actual implementation of the MPSoC.
– It avoids inconsistency and errors and helps to ensure completeness of the specifications. The execution of the combined architecture/application model represents the realization of the model that behaves in the same way as the system. The following section will give details on the execution model for the mixed architecture/application description in Simulink and SystemC.

1.5.2 Execution Model for Mixed Architecture/Application Model

The execution of the mixed hardware/software model is performed through a simulation which allows validation and debug of the system functionality at different stages of the design process. The execution model allows capturing the behavior of the application together with the architecture and a detailed hardware–software interaction. The execution model helps to create early performance models of the MPSoC and validate the system performances. By using different test benches, the execution model allows to test different functionality scenarios, even before the final implementation.

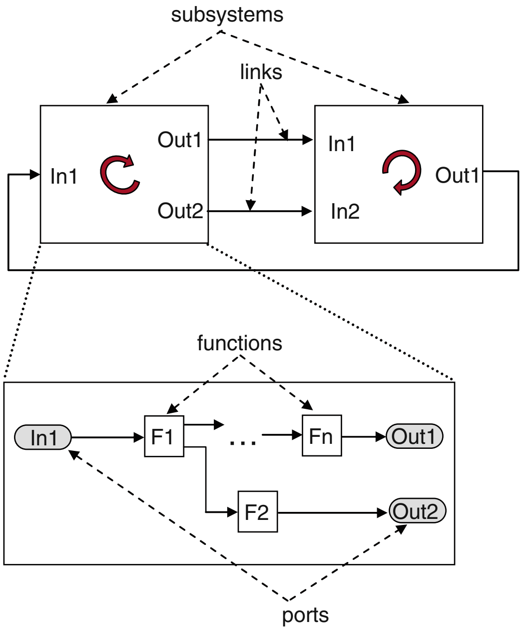
The execution model can be described using different simulation environments, such as Simulink or SystemC. In the following sections, the execution models described in Simulink, respectively, SystemC will be presented.

1.5.2.1 Execution Model Described in Simulink

Simulink is an environment for multi-domain simulation and model-based design for dynamic and embedded systems [94]. It provides an interactive graphical environment and a customizable set of block libraries that allows designing, simulating, implementing, and testing a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing, from a large set of application domains, e.g., automotive, aerospace and defense, communications and electronics. The supported models of computation include but it is not limited to discrete events, continuous time domains, discrete time domains, or finite state machines [89].

Additionally, Simulink provides the capability to model and simulate the mixed architecture/application representation like a synchronous dataflow model. The hardware is described in Simulink using the concept of subsystems, ports, and signals provided by the standard Simulink library (Fig. 1.10).

Fig. 1.10 Simulink concepts



The software is described as a set of functions using the standard Simulink blocks or user-defined functions (Fig. 1.10). The functions are encapsulated into the subsystems and may access external signals through the ports of the subsystems. In this model, the hardware–software interaction is modeled using the concepts of signals connecting different ports.

The execution model in Simulink supports various simulation options, such as the simulation's start and stop time and the type of solver used to solve the model at each simulation time step. Specifying simulation options is called configuring the model. Simulink enables to create multiple model configurations, called configuration sets, modify existing configuration sets, or switch configuration sets.

Once the configuration model that meets the application requirements is defined or selected, the mixed architecture/application model can be executed. Simulink runs the simulation from the specified start time to the specified stop time. While the simulation is running, the system designer can interact with the simulation in various ways, stop or pause the simulation, and launch simulations of other models. If an error occurs during the simulation, Simulink halts the simulation and pops up a diagnostic viewer that helps the user to determine the cause of error.

Figure 1.11 shows the main steps of the simulation engine in Simulink. The first step is the initialization. This includes the compilation and link phases. First, the Simulink engine invokes the model compiler. The model compiler converts the model to an executable form. In particular, the compiler evaluates the values of

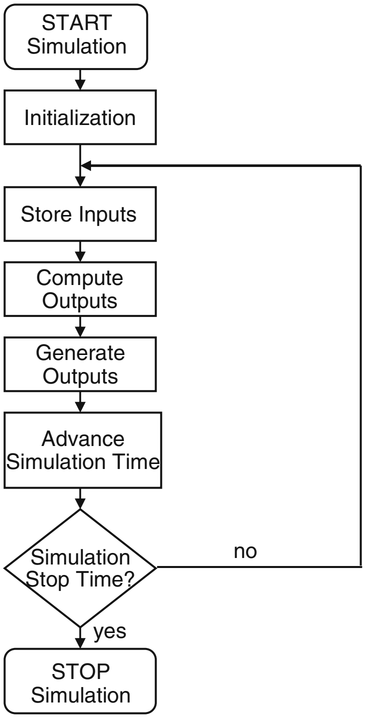


Fig. 1.11 Simulink simulation steps

the all block parameters. Then, it determines the signal attributes for the links not explicitly specified, e.g., name, data type, numeric type, and dimensionality of the signal.

The model compiler checks that each block can accept the signals connected to its inputs. Simulink uses a process called attribute propagation to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives. Then, the model compiler performs block reduction optimizations and flattens the model hierarchy by replacing the hierarchical subsystems with the blocks that they contain. It also determines the sorted order of the blocks, which represent the invocation order of the blocks during the simulation. Finally, the model compiler determines by propagation the sample times of all blocks in the model whose sample times are not explicitly specified by the designer. After the compilation, the Simulink engine allocates memory needed for signals, states, and run-time parameters. It also allocates and initializes memory for data structures that store the run-time information for each block. This corresponds to the link phase. After the memory space allocation, initial values are assigned to the states and outputs of the model to be simulated. The initialization phase occurs once at the start of the simulation loop.

After the initialization, during the simulation loop, the Simulink engine successively stores the inputs, computes, and generates the outputs and states of the system at intervals from the simulation start time to the finish time. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called step size. The step size depends on the type of the solver. The next simulation step is the sum of the current simulation time and the step size. When the simulation stop time is reached, the simulation stops.

A solver is a Simulink software component that determines the next time step that a simulation needs to take to meet target accuracy requirements specified by the user. Simulink provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. There are two types of solvers: fixed step and variable step. With a fixed-step solver, the step size remains constant throughout the simulation. By contrast, with a variable-step solver, the step size can vary from step to step, depending on the model's dynamics. In particular, a variable-step solver reduces the step size when a model's states are changing rapidly to maintain accuracy and increases the step size when the system's states are changing slowly in order to avoid taking unnecessary steps.

In Simulink, the simulation of the application model starts by default at 0.0 s and ends at 10.0 s. The solver configuration panel allows specifying other start and stop times for the currently selected simulation configuration. The simulation time and the actual clock time are not the same. For example, running a simulation for duration defined in the solver configuration pane of 10 s usually does not take 10 s. The amount of time it takes to run a simulation depends on many factors, including the model's complexity, the solver's step sizes, and the host computer's speed.

1.5.2.2 Execution Model Described in SystemC

SystemC is a standard system-level design language based on a C++ class library used mostly for embedded and SoC applications [118]. Any discrete time system can be modeled in SystemC. This includes some well-known models of computation like multi-rate dataflow (static and dynamic), Kahn process networks [72], discrete events used for hardware RTL (register transfer level) modeling, network modeling and transaction-based SoC modeling, or communicating sequential processes [89]. Furthermore, the AMS extension of SystemC permits modeling of analog devices.

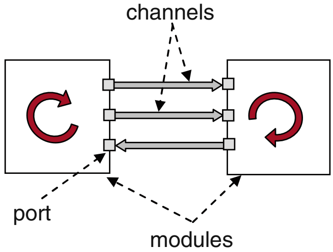
SystemC is convenient for mixed hardware/software modeling. It provides the abstraction and constructs needed for high-level hardware modeling and verification. Such abstraction, primarily at the transaction level, allows much faster simulations and analysis and enables design issues to be detected early in the process. At the same time, the software can be described as C or C++ modules.

The hardware is described in SystemC using the concept of modules, ports, and channels or signals provided by a C++ extension library, as depicted in Fig. 1.12.

The software is described using the concept of concurrent threads. The threads are encapsulated into the modules and may access external channels through the ports of the modules. In this model, the hardware–software interaction is modeled using the classical concepts of channels or signals.

The execution model in SystemC allows the execution of the threads independently using their own execution stacks [118]. These threads or processes can be

Fig. 1.12 SystemC concepts



sensitive to events on input or output ports. The sensitivity list of a process may be defined statically or can change dynamically during the simulation. There are three types of SystemC processes: SC_THREAD, SC_CTHREAD, and SC_METHOD. The threads (SC_THREAD) can suspend and resume execution only when they call the *wait()* function. The clocked threads (SC_CTHREAD) are special threads which are sensitive only to clock signals. The methods (SC_METHOD) behave like a non-preemptable standard procedure written in a high-level programming language. An SC_METHOD may suspend and resume execution when it gives the control to the SystemC simulation kernel.

The simulation in SystemC involves execution of the SystemC scheduler which may execute processes of the application. The SystemC simulation kernel does not preempt the execution of a thread. The SystemC processes are executed until completion or until they yield control to the simulation engine. Hence, the SystemC scheduler is cooperative multi-tasking, as the processes run without interruption up to the point where it either returns or calls the function *wait()*. The thread code between two *wait()* calls is executed in one simulation clock cycle. Simulation time can advance only when a *wait()* statement has been called.

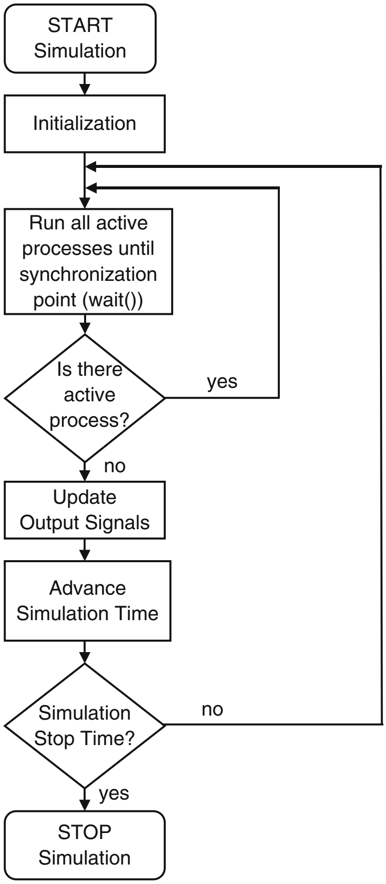
The SystemC processes scheduler is event-driven, meaning that the processes are executed to the occurrence of events. Events occur or are notified at precise points in the simulation time. The scheduler can execute a process (a SystemC method or a SystemC thread) in the following cases:

– In response to the process instance having been made runnable during the initialization phase of the simulation
– In response to a call function *sc_spawn* to create processes dynamically during the simulation
– In response to the occurrence of an event to which the process instance is sensitive. Each process can have static or dynamic sensitivity list. The static sensitivity list represents the list of events that may cause the process to be resumed or triggered that are fixed before the simulation. The dynamic sensitivity list may change during the simulation
– In response to a time-out having occurred. A time-out occurs when a given time interval has elapsed

The simulation can start only after instantiation and proper connection of all modules and signals. The simulation starts with calling *sc_start()* from the top level that contains *sc_main()*. The *sc_start()* accepts as argument the total number of default time units of the simulation period. If the argument is a negative number, the system is simulated infinitely.

Similar to VHDL or Verilog, SystemC threads scheduler supports delta cycles. A delta cycle is comprised of evaluate and update phases, and multiple delta cycles may occur at a particular simulated time. As illustrated in Fig. 1.13, the SystemC simulation has the following steps: initialization, evaluation, and update.

Fig. 1.13 SystemC simulation steps



In the initialization step, the SystemC scheduler establishes the initial value for all signals and makes all the processes active. During the evaluation step, the scheduler executes all the processes ready to run in an unspecified order. The order of the thread execution is non-deterministic within a certain simulation phase. This may cause events notification to occur which make other processes ready. The active processes are executed until a synchronization point, i.e., *wait()* statement. If new processes become active, the evaluation will continue until the list of processes ready for execution becomes empty. The last phase is to update the values of the output signals and to advance the simulation time to the earliest pending time notification. If the simulation time attains the end of simulation time, the simulation will stop, otherwise it will continue with the evaluation phase.

1.6 Examples of Heterogeneous MPSoC Architectures

In the following paragraphs, examples of heterogeneous MPSoC architectures are given. These examples will be used as case studies in the next chapters.

The target hardware architecture considered is represented by a heterogeneous MPSoC architecture. The heterogeneous architecture contains different processor subsystems and memory or hardware subsystems. The different subsystems are interconnected via a global communication network such as bus or network on chip (NoC). A processor subsystem, which executes several tasks, includes one or more processors, local memories, peripherals, and local buses connecting them. The hardware subsystem has the similar structure with the processor subsystem. It includes one or more hardware IPs, local memories, local buses, and communication I/Os such as bus bridge or network interface. A memory subsystem includes a set of memories such as embedded SDRAMs, flash memories, and external memories and it is connected to a communication network via communication I/Os.

In this book, the programming environment and the different software generation and validation steps are illustrated for three examples of heterogeneous MPSoC architectures, namely, the 1AX, the Diopsis RDT architecture with AMBA bus, and the Diopsis R2DT architecture with NoC. The main differentiation between these architectures, as it will be described in the following paragraphs, consists of the type and number of processors incorporated in the architecture, the type of the adopted network component (bus or NoC in different topologies), and the different communication and synchronization schemes provided by the architecture.

1.6.1 1AX with AMBA Bus

The first example of heterogeneous MPSoC represents the 1AX architecture. This architecture is illustrated in Fig. 1.14.

The 1AX architecture is comprised of two processor subsystems (ARM-SS and XTENSA-SS) and one global memory subsystem (MEM-SS). The ARM-SS includes an ARM7 processor [6] used to execute the control functions of the application, while the XTENSA-SS contains a configurable Xtensa processor [152] for processing data-intensive algorithms. The Xtensa processor can be customized to the target application functions with an automatic instruction set generator

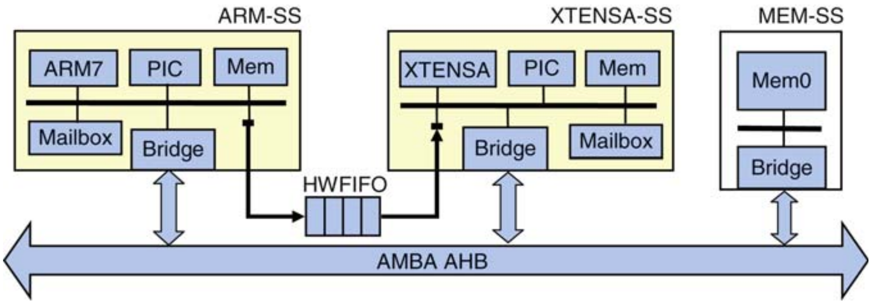


Fig. 1.14 1AX MPSoC architecture

called XPRES (Xtensa processor extension synthesis). The different subsystems are interconnected using an AMBA bus [6].

Each processor subsystem integrates the processor core (ARM7, respectively, XTENSA), the local memory (Mem), the programmable interrupt controller (PIC), mailbox for the processors synchronization, local bus, and the bridge to interface with the AMBA bus.

The interrupt controller handles external interrupts according to priority to cope with external events (from mailbox) or data arriving from the other components (hardware FIFO).

The local memories store program code and data. They also serve to store the buffer used for the communication between the tasks running on the same processor. The MEM-SS includes a global memory accessible by both processing units and the bridge for the connection with the AMBA bus. The 1AX architecture contains also a hardware FIFO (HWFIFO) directly connected to the local buses of the two processor subsystems. The HWFIFO contains synchronization.

Each processor and hardware subsystem have a memory address space of 4 MB (megabytes), while the memory subsystem has a memory address space equal to its memory size. The 1AX architecture contains a global memory of 256 MB. The processor subsystems have the first 4 MB address space reserved for the local bus transactions (0×00000000–0×003FFFFF). The memory address space of a processor subsystem is divided into two parts: 3 MB for local memory and 1 MB for peripheral memories. Bus transactions with addresses lower than 4 MB (0×00400000) are treated as accesses to local components, while those with addresses higher than 4 MB are forwarded to the global AMBA bus via the bridge component of the processor subsystem. The bus bridge receives the forwarded transactions within the address space assigned to its subsystem. The memory address space for the 1AX architecture is illustrated in Fig. 1.15.

This architecture allows two types of communication schemes between the processors: using the global memory and using the hardware FIFO.

In the first communication scheme, one processor can deliver data to other processor through global shared memory and send a synchronization event via a mailbox between different processors. For example, a data transfer from the ARM

Fig. 1.15 Memory address space for the 1AX MPSoC architecture

Reserved:	0 MB ~ 4 MB
ARM7-SS:	4 MB ~ 8 MB
XTENSA-SS:	8 MB ~ 12 MB
HWFIFO:	12 MB ~ 13 MB
MEM-SS:	2 GB ~ 2 GB +256 MB

processor to the XTENSA processor using the global memory has the following steps: first, the ARM processor checks a bit in its mailbox. If the bit is set to 1, which means that a space is available in the global memory, the ARM processor clears the bit to 0, writes data to the global memory, and sets a bit of the mailbox in the Xtensa processor subsystem to 1, which means that data are available in the global memory. After checking the bit in the mailbox, the Xtensa processor clears the bit of its mailbox to 0, reads the data from the global memory, and sets the bit of the mailbox in the ARM processor subsystem to notify the completion of the read operation. For this type of communication, the bandwidth of the global interconnect could become bottleneck of the inter-processor communication. It may also cause long latency to access the data because of the limitation of the shared bus.

The second possible communication scheme between the two processors is based on the hardware FIFO. The HWFIFO is a point-to-point communication between two processor subsystems. Besides the data transfer, the HWFIFO also implements the synchronization mechanism of the processors. For instance, a data transfer initiated by the ARM processor using the HWFIFO has the following steps: the ARM processor copies data from its local memory to the hardware FIFO directly. When the data number in the FIFO reaches a certain threshold, the Xtensa processor checks it through interrupt or polling methods. Then, the Xtensa processor copies the data from the hardware FIFO to its local memory. When the hardware FIFO reaches empty, the ARM processor checks it through interrupt or polling methods and copies the data again. The HWFIFO provides a new path to transfer data instead of using the shared memory and global network. Thus, it can decrease the required bandwidth of the global memory and network and speed up the communication. But compared to the global memory, the HWFIFO increases hardware area because it needs extra shared memory. It also relies on the processor to transfer data.

1.6.2 Diopsis RDT with AMBA Bus

The second target architecture example is the Shapes MPSoC architecture [121, 141], which is a multi-tile architecture based on a Diopsis tile. Figure 1.16 illustrates an elementary tile, namely, the RDT (RISC + DSP Tile).

The Diopsis tile is a triple core system integrating an ATMEL mAgicV VLIW DSP [9], an ARM 9 RISC microcontroller [6], and a distributed network processor

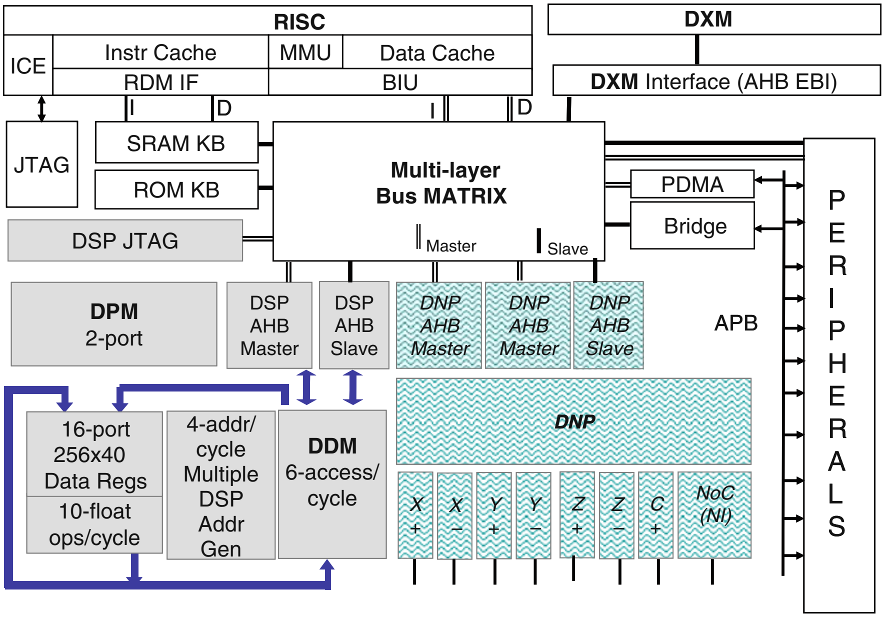


Fig. 1.16 Diopsis RDT heterogeneous architecture

(DNP), all interconnected through an AMBA bus. The DNP uses 3D next-neighbor connections for off-chip interconnects and interfaces a NoC to integrate multiple tiles on a single chip. An appropriate balancing of instruction level parallelism (ILP), memory size, and frequency provides a sound control of potential wire delay problems within the individual tiles. The elementary Diopsis tile is also called D940. The system combines the flexibility of the ARM9 controller with the high performance of the DSP and the on-chip and off-chip networking capability of the DNP.

The ARM9 processor represents the ARM926EJ-S type of the ARM9 family of 32 bits general-purpose processors, which offers very low power consumption. The CPU runs at 200 MHz frequency. This processor is equipped with 16 kB data cache and 16 kB instruction cache memories. It provides DSP instruction extensions and it operates 264 mega instructions per second (MIPS) [6]. The processor is outfitted with a memory management unit.

The DSP processor runs at 100 MHz and delivers 1.2 giga floating-point operations per second (GFLOPS) and 1.8 giga operations per second (GOPS). It is equipped with 256 data registers, 64 address registers, 10 independent arithmetic operating units, 2 independent address generation units, and a DMA engine. To sustain the internal parallelism, the data bandwidth among the register file, the operators, and the data memory system amounts to 80 bytes/cycle. The data memory system is designer to transfer 28 bytes/cycle. For instance, activating all the computing units, the DSP is able to compute one FFT butterfly per cycle [9].

The architecture is based on the AMBA bus, more precisely the multi-layer AHB matrix and the APB [6]. The AHB matrix consists of seven masters and five slaves.

The local memories of the DSP and RISC can be accessed by both processing units. Additionally, a distributed external memory (DXM) can be used to share data between all the processors. The data transfer between these processors can follow different schemes based on an AMBA bus, e.g., the DSP can read/write data to the local memory of the ARM by using or not a DMA transfer.

This book considers as example of MPSoC architecture a simplified version of the initial Diopsis tile. The selection of the components from the original architecture still captures all the possible communication schemes and specific I/O components. The subset is shown in Fig. 1.17.

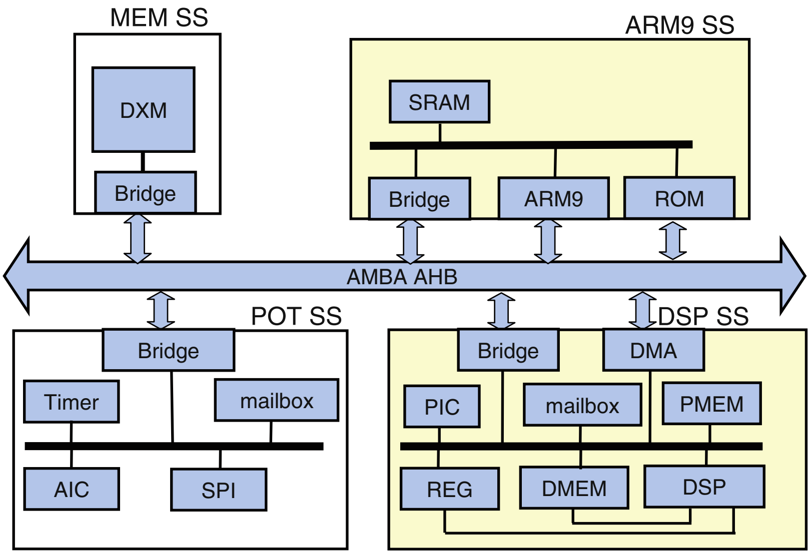


Fig. 1.17 Target Diopsis-based architecture

The reduced Diopsis tile contains two software subsystems: the ARM and the DSP software subsystems. The ARM subsystem includes the processor core and local memories: SRAM for data and ROM for program code. The DSP subsystem includes the DSP core, data memory (DMEM), program memory (PMEM), control and data registers (REG), direct memory access engine (DMA), programmable interrupt controller (PIC), and the mailbox as synchronization component for the communication between the two processors. The interrupt controller handles the external interrupts according to their priorities. The timer has the highest priority of all the interrupt sources for both processors. All the communication devices are assumed to generate an interrupt when new data become available.

Apart from the software subsystems, the architecture contains two hardware subsystems as well. The hardware nodes consist of distributed external memory subsystem (DXM) and peripherals on tile (POT) subsystem. The distributed external memory subsystem includes a global memory shared by the processors. The POT includes not only the system peripherals of the RISC processor, e.g., timer, advanced interrupt controller (AIC), but also the I/O components of the tile such as the serial peripheral interface (SPI).

The interconnection between these software and hardware subsystems is made via the AMBA bus. Hence, all the subsystems contain a bridge component to interface with the AMBA bus and a local bus for the local components interconnection. The AMBA bus supports burst mode transmissions in order to allow continuously data transfer through the bus after its initialization.

For performance reasons, the ARM processor can access directly the data memory and control/status registers of the DSP processor via the AMBA slave interface of the DSP subsystem. In the same way, the DSP core can read/write directly on the local memory of the RISC processor by initiating a DMA transfer. Moreover, the processors can store and load data to/from DXM connected to the AMBA bus. Therefore, this architecture allows different kinds of communication mapping schemes between the processors characterized by different performances.

1.6.3 Diopsis R2DT with NoC

The third target architecture considered in this book represents the Diopsis R2DT (RISC + 2 DSP) tile. This heterogeneous architecture is an extension of the previously presented RDT tile. Figure 1.18 shows the Diopsis R2DT tile.

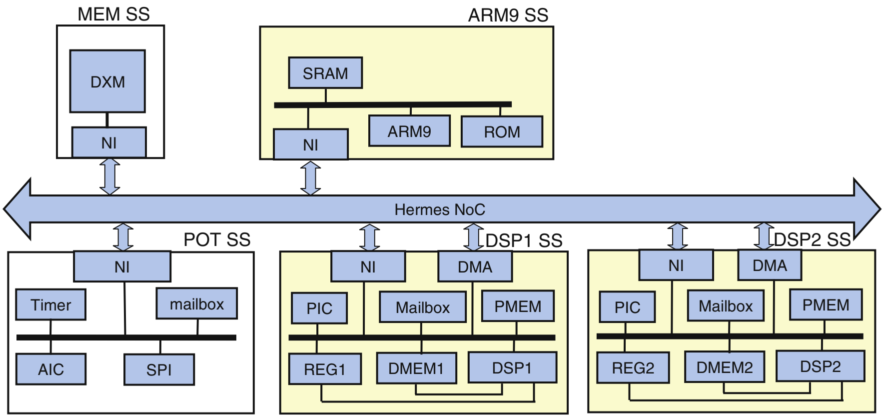


Fig. 1.18 Diopsis R2DT with Hermes NoC

It contains three software subsystems: one ARM9 RISC processor subsystem and two ATMEL magicV VLIW DSP processing subsystems. Similarly with the RDT tile, the hardware nodes represent the global external memory (DXM) and POT (peripherals on tile) subsystem. The POT subsystem contains the peripherals of the ARM9 processor and the I/O peripherals of the tile. All the three processors may access the local memories of the other processors and the distributed external memory (DXM).

In this architecture, the different subsystems are interconnected using the Hermes network on chip (NoC) [107]. The bridges required for the data transfer through the AMBA bus of the RDT architecture are replaced with network interface (NI) components. In the same manner, the DMA engines of the DSP subsystems provide interfaces to the NoC instead of the AMBA AHB interface.

The NoC represents an on-chip packet-switched micro-network of interconnects. It is constructed from multiple point-to-point data links interconnected by switches (routers), such that data messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches. As the NoC can operate simultaneously on different data packets, it allows several data transfers in parallel through the network. Therefore, it overcomes the limitations offered by a bus in terms of bandwidth and scalability [16].

The basic components of a NoC are the switches, network interfaces, and the links between them [17]. The data delivered through the NoC from the source module to the destination module are divided into packets. A packet represents the basic unit for routing and sequencing. Packets are often composed by a header, a payload, and a tailor.

To ensure correct functionality during the message transfers, a NoC must avoid deadlock, livelock, and starvation. The deadlock means the situation when a packet does not reach its destination because it is blocked at some intermediate router. Usually, deadlock happens in case of a cyclic dependency among the routers requiring access to a set of resources so that no forward progress can be made, no matter what sequence of events happens. Livelock means the situation when a packet does not reach the destination because it enters in a cyclic graph. This situation can be avoided with adaptive routing strategies. Starvation means when a packet does not reach its destination because some resource does not grant access, while it grants access to other resources. An example of this kind of situation is when a packet in a buffer requests the output channel, but it remains blocked because the output channel is always allocated to another packet.

Depending on the switching strategy, the packets may be divided into flits. A flit (flow control digit) is the basic unit of bandwidth and storage allocation. Flits do not have any routing or sequence information and have to follow the router for the whole packet.

There are several factors that may influence the performances of a NoC [128], such as the following:

– *Topology*. The topology represents the static arrangement of the routers and the channels between them. A good topology allows fulfilling the requirements of the traffic at reasonable cost. According to the topology, NoCs can be classified into static and dynamic networks. In static networks, each router has fixed point-to-point connections to other routers. Examples of static topologies are the ring, butterfly, tree, torus, and mesh topologies. Dynamic networks employ communication channels that can be configured at application runtime.

- *Routing techniques*. The routing algorithm performs the selection of a path through the network. For instance, the XY routing algorithm supposes to route the flit first on the horizontal direction (X) and then, when it reaches the column where the destination module is located, it is routed in a vertical direction (Y). The XY routing algorithm is minimal path routing algorithm and is free of deadlock [8]. The YX routing algorithm is similar to the XY, but reverses the order of vertical and horizontal routing. Another type of routing technique is the west-first algorithm.
- *Communication mechanism.* The communication mechanism specifies how messages pass through the network. Two methods of transferring messages are circuit switching and packet switching. In circuit switching, a path named connection is established before the packets can be sent by the allocation of sequence of channels between the source and its destination. In packet switching, the packets are transmitted without any need for connection establishment procedures.
- *Switching strategy*. The switching strategy characterizes the packet switching communication mechanism [132]. It specifies how the packets are forwarded by the routers during the transmission. The most well-known switching strategies are store and forward, virtual cut-through, and wormhole [17]. In the store-and-forward switching strategy, a switch cannot forward a packet until it has been completely received. In virtual cut-through mode, a switch can forward the packet as soon as the next switch gives the guarantee that a packet will be accepted completely [131]. The wormhole switching mode is a variant of the virtual cut-through mode. A packet is transmitted between the switches in units called flits. In the wormhole transmission scheme, the router can start forwarding the first flit of a packet without waiting for the tail [106]. Another type of switching strategy represents the small frame switching.
- *Flow control*. The flow control means how are the network resources allocated, if packets traverse the network.
- *Router architecture*. This defines the properties of the switches and the buffers of the switches, such as buffer size, buffer dimension, number of buffers.
- *Traffic pattern*. The traffic pattern defines the dataflows between every pair of modules connected to NoC.

The Hermes NoC supports two types of topologies: mesh and torus. In the mesh topology, the NoC employs a 2D arrangement with nine routers (3×3). The routers may have from three to five ports, depending on the router position relative to the limits of the mesh. The mesh NoC uses a pure XY routing algorithm shared by all the ports, a round-robin scheduler to arbitrate the simultaneous packet transmission requests, and wormhole packet switching strategy.

In the torus NoC model, every router has five bidirectional ports to implement a 3×3 2D torus topology with wraparound links at the edges of the network. The routing algorithm is a deadlock free version of the well-known non-minimal west-first algorithm proposed in [60]. The arbitration and switching strategies are those that characterize the mesh topology as well.

Figure 1.19 illustrates the Hermes NoC in both topologies, torus and mesh.

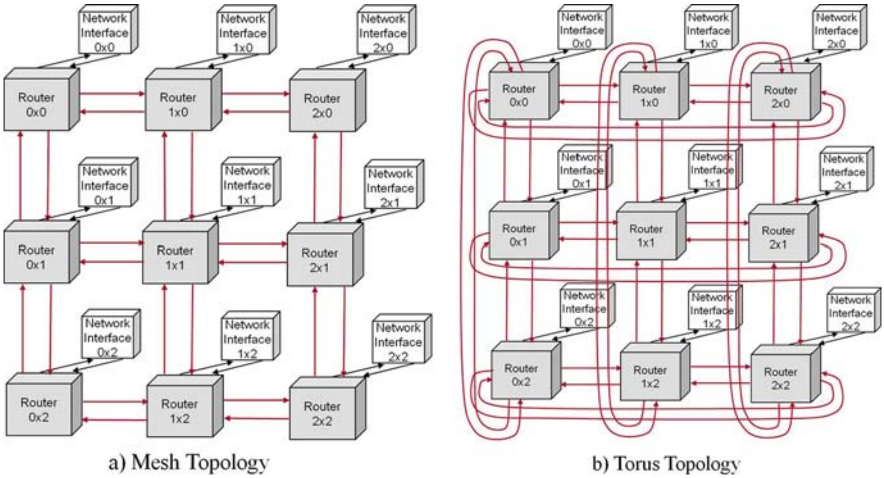


Fig. 1.19 Hermes NoC

1.7 Examples of Multimedia Applications

In the following paragraphs, three examples of applications are given. These examples represent the target applications that will run on the architecture examples previously described, considered as case studies in the remaining part of this book.

The target application domain represents the multimedia applications domain. This kind of application can be found in many areas, such as entertainment, engineering, advertisements, medicine, scientific research, spatial–temporal applications (visual thinking, visual/spatial learning). Multimedia applications are based on information processing and compression, e.g., text, audio, graphics, animation, video, interactivity. Multimedia compression methods are lossy, meaning that the decompressed signal is different from the original signal. Compression algorithms make use of perceptual coding techniques that try to throw away data that are less perceptible to the human eye and ear. These algorithms also combine lossless compression with perceptual coding to efficiently code the signal.

Examples of multimedia applications are video compression standards. Mainly, there are two series of standards: the MPEG (e.g., MPEG2 and MPEG4) and H.26× (e.g., H.263) series. The MPEG was developed primarily for broadcast applications, while the H.26× for symmetric applications, where both sides must encode and decode, such as videoconferencing. The two groups were recently completed a standard which covers both types of applications, namely, the H.264 or advanced video codec, shortly AVC. The H.264 will be presented in detail later in this book. Other examples of multimedia applications are the JPEG 2000 and motion JPEG for image compression and decompression, or the MPEG 1 audio (layer 3) for audio encoding and decoding, also known as MP3.

As they will be described in the next sections, this book considers the following target applications:

– The token ring application, a simpler example used to illustrate the concepts and methodology and targeted to be executed on the 1AX MPSoC architecture
– The motion JPEG decoder for image processing that will be mapped and executed on the Diopsis RDT architecture
– The H.264 encoder application for video processing, which will be running on the Diopsis R2DT architecture with different NoC topologies

The following paragraphs describe these three application examples (token ring, motion JPEG, and H.264).

1.7.1 Token Ring Functional Specification

The first target application is the token ring application. The application is comprised of three nodes that exchange a token. The nodes are connected in the form of a ring. When a node receives the token, it checks if the node is the destination of the token by comparing the node's identifier with the token's value. In this case, the node performs some computations. Otherwise, it forward the token to the next node. The functional specification of the token ring application is illustrated in Fig. 1.20.

If the token is designed to the first node, the node increments the token value with two units. The second node increments the token value with one unit. Finally, the third token multiplexes the value of the token and computes a DFT (discrete Fourier

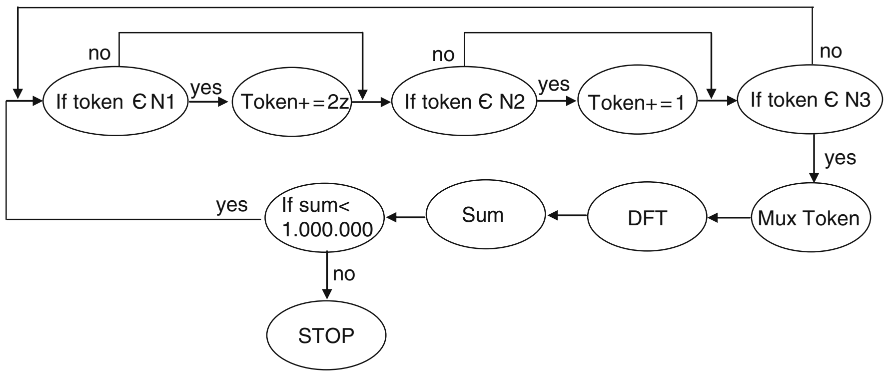


Fig. 1.20 Token ring functional specification

transform) function. The multiplexed value of the token represents the input for the DFT computation.

The DFT, occasionally called the finite Fourier transform, is a transform for Fourier analysis of finite-domain discrete time signals. It expresses an input function in terms of a sum of sinusoidal components by determining the amplitude and phase of each component.

However, the DFT is distinguished by the fact that its input function is *discrete* and *finite*: the input to the DFT is a finite sequence of real numbers, which makes the DFT ideal for processing information stored in computers. In particular, the DFT is widely employed in signal processing and related fields to analyze the frequencies contained in a sampled signal, to solve partial differential equations, and to perform other operations such as convolutions. The DFT can be computed efficiently in practice using a fast Fourier transform (FFT) algorithm [39]. The formula of the DFT computation is given below:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-((2\pi i)/N)kn}, \quad k = 0, \ldots, N-1$$

where $x_0, \ldots, x_{N-1}$ is transformed into the sequence of N complex numbers $X_0, \ldots, X_{N-1}$ during the DFT computation; e is the base of natural logarithm; i is the imaginary unit ($i^2 = -1$); and π is pi.

After the DFT computation, the generated coefficients are summed and assigned as new value to the token. The iteration process will stop when the resulted sum is bigger than 1,000,000. Otherwise, the new value of the token is transmitted to the first node forming a loop.

1.7.2 Motion JPEG Decoder Functional Specification

The motion JPEG decoder application represents an image processing multimedia application. In this book, the baseline motion JPEG decoder is used as target application example, which represents the basic JPEG decoding process supported by all the JPEG decoders [169]. JPEG is named from its designer, the Joint Photographic Expert Group.

The JPEG compression algorithm splits the input image on blocks of 8×8 pixels, as shown in Fig. 1.21.

The two major techniques used by JPEG encoder are *the discrete cosine transform (DCT)* plus quantization, which performs perceptual coding, plus *Huffman coding*, also called *variable length coding* as a form of entropy coding for lossless encoding.

The DCT is a frequency transform, whose coefficients describe the spatial frequency content of an image. The DCT operates on 2D set of pixels, in contrast with the Fourier transform which operates on a 1D signal. The JPEG algorithm performs the DCT on 8×8 blocks of pixels. The DCT coefficients can be arranged in an 8×8

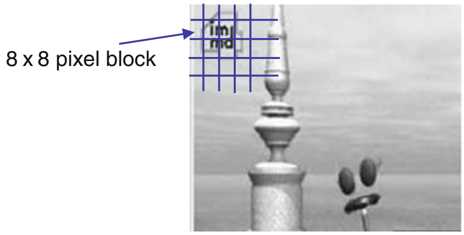


Fig. 1.21 Splitting images in 8×8 pixel blocks

matrix. The top left is known as DC coefficient, since it describes the lowest resolution component of the image. The other elements of the matrix are named as AC coefficients. The DCT coefficients are quantized to change the high-order AC coefficients to zero, thus allowing efficient compression. Then, the quantified DCT matrix is traversed in a zigzag pattern, as shown in Fig. 1.22, to create long strings of zero values that can be efficiently Huffman encoded.

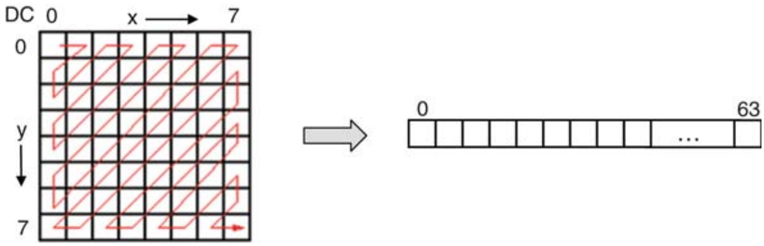


Fig. 1.22 Zigzag scan

The JPEG decoder performs the exact opposite process of the encoder. A simplified view of the JPEG decoder is illustrated in Fig. 1.23. The JPEG decoder performs the decompression of an encoded JPEG bitstream (01011...) and renders the decoded images on a screen.

The main functions of the motion JPEG decoder algorithm are described as follows:

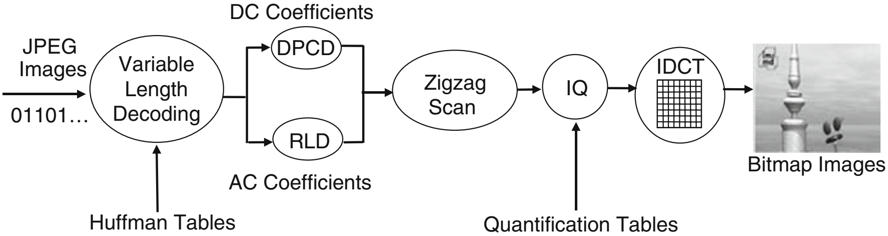


Fig. 1.23 Motion JPEG decoder

- *VLD (variable length decoder).* The VLD represents the Huffman entropy decoder. The input binary sequence of the compressed image is converted to a symbol sequence using Huffman tables. This represents the opposite step of the VLC (variable length coder) step of the JPEG encoder, when variable length codes are assigned to the symbols created from the DCT coefficients. The decoder can store only two sets of Huffman tables: one AC table and one DC table per set.
- *DPCD (differential pulse code demodulation).* The DPCD represents the opposite process of the DPCM (differential pulse code modulation) part of the JPEG encoder, which is applied on the DC coefficient. The DC coefficient represents the coefficient with zero frequencies in both dimensions of the DCT coefficients matrix, located at the left-top corner [0, 0]. The DPCD is in charge with the reconstruction of the DC coefficient.
- *RLD (run length decoding).* The RLD is the opposite step of the RLC (run length coding) of the compression algorithm, which is applied on the AC coefficients (the 63 DCT coefficients which are different from the DC coefficient). The AC coefficients are treated separately from the DC coefficient. The RLD supposes to reconstruct the sequence of the AC coefficients from the sequence of symbols by inserting the zero-valued AC coefficients before the non-zero-valued AC coefficients in the coefficients sequence.
- *Zigzag Scan.* This step puts back the 64 DCT coefficients in the form of a matrix with 8×8 dimensions. The input of this step is an array of 64 elements in zigzag order and its output is an 8×8 matrix in original order.
- *IQ (inverse quantization).* The IQ is applied upon the 64 DCT coefficients using quantification tables. This step consists of the multiplication of each of the 64 coefficients by its corresponding quanta step size. The quanta steps are stored in the quantification tables.
- *IDCT (inverse discrete cosine transform).* This step transforms the 64 DCT coefficients (the 8×8 block) from frequency domain to spatial domain and reconstructs the 64-point output image signal by summing the basis signals.

1.7.3 H.264 Encoder Functional Specification

The H.264 encoder application represents the third application example used as case study. This application is a video-processing multimedia application. It represents a standard for video compression also known as MPEG-4 part 10 or AVC (advanced video coding). The H.264 supports coding and decoding of 4:2:0 *YUV* video formats.

The input image frame (F_n) of a video sequence is processed in units of a macroblock, each consisting of 16 pixels. A pixel consists of three color components: *R* (red), *G* (green), and *B* (blue). Usually, pixel data are converted from *RGB* to *YUV* color space, where *Y* represents the luma, and *U* and *V* the chroma samples. The widely used *RGB–YUV* conversion equations are the following [130]:

$$\begin{cases} Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \\ U = 0.564 \times (B - Y) \\ V = 0.713 \times (R - Y) \end{cases}$$

$$\begin{cases} R = Y + 1.402 \times V \\ G = Y - 0.344 \times U - 0.714 \times V \\ B = Y + 1.772 \times U \end{cases}$$

A macroblock contains 16×16=256 Y luma samples and 8×8=64 U and 8×8=64 V chroma components (Fig. 1.24). Each of these components is processed separately. There are three types of macroblocks: I, P, and B. The macroblocks are numbered in raster scan order within a frame. A set of macroblocks is called slice. A video picture is coded as one or more slices. The I slice contains only I macroblocks. The P slice contains P macroblocks and/or I macroblocks. The B slice contains B macroblocks and/or I macroblocks [130].

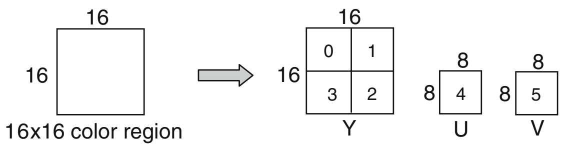


Fig. 1.24 Macroblock (4:2:0)

To encode a macroblock, there are three main steps: prediction, transformation with quantization, and entropy encoding. These main functions of the standard H.264/AVC (advanced video coding) are illustrated in Fig. 1.25 [130].

The H.264 encoder and decoder maintain two lists of reference pictures (list 0 and list 1), which represent sets of numbers corresponding to the reference pictures and containing the frames that are previously encoded and decoded.

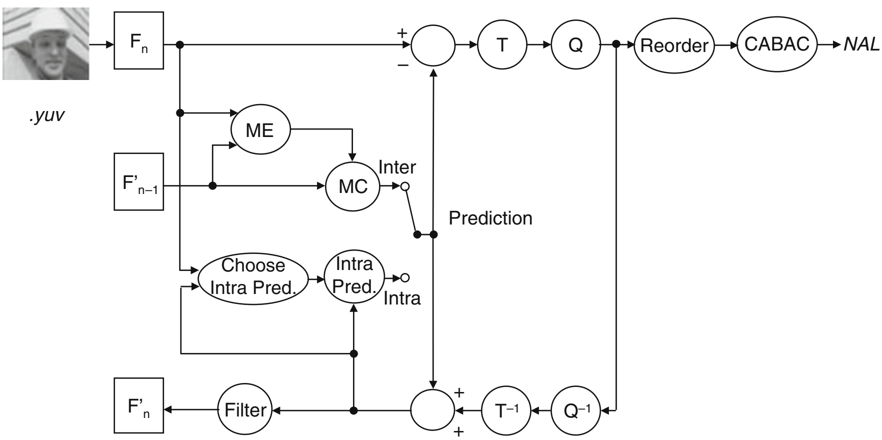


Fig. 1.25 H.264 encoder algorithm main profile

The prediction step tries to find a reference macroblock which is similar to the current macroblock to be encoded. Depending on where the reference macroblock comes from, there are two types of prediction: intra- and inter-mode. The I macroblocks are predicted using intra-prediction from the decoded samples in the current slice. A prediction is formed either for the complete macroblock or for each 4×4 block luma and associated chroma samples. The P macroblocks are predicted using inter-prediction from the reference frames. Each P macroblock can be predicted from one frame in list 0. The B macroblocks are predicted using inter-prediction from one or two reference frames, one from list 0 and/or one frame from list 1.

The first frame from the movie is always encoded using intra-mode (because there are no previous frames to be considered as reference frames) [22]. Then, during the movie, several other slices/frames can be encoded in intra-mode. The interval between two consecutive frames encoded in intra-mode is called key frame. In intra-mode or I-mode, the macroblocks are predicted using the previously encoded, reconstructed, and unfiltered macroblocks. In this case, the reference macroblock is usually calculated with mathematical functions of neighboring pixels of the current macroblock.

In the inter (P or B)-mode the macroblocks are predicted from a reference picture (F'_{n-1}) by motion estimation (ME) and motion compensation (MC). This involves finding a 16×16 sample region in the reference frame that closely matches the current macroblock. The reference picture maybe chosen from a selection of past or future (display order) pictures that have been already encoded, reconstructed, and filtered. The ME involves finding a 16×16 sample region in the reference frame that closely matches the current macroblock. A popular matching criterion is to measure the *s*um of *a*bsolute *d*ifference (SAD) between the current block and the candidate block and to find its minimal value [37]. The formula for the SAD computation is shown below, where C is the current macroblock that is searched in the reference frame and P is the macroblock in the reference frame that is compared with the current one:

$$\mathrm{SAD} = \sum_{x=0,y=0}^{x<16,y<16} |C[i,j] - P[i,j]|$$

During the motion estimation, the SAD values are computed for multiple coordinate positions (Fig. 1.26). The bigger is the SAD value, the bigger is the difference between the current macroblock and the compared macroblock from the previous frame. The macroblock from the reference picture which implies the smallest SAD value is considered as the best matching macroblock. The motion estimation computes the motion vectors for the current macroblock compared with the best matching region from the reference picture. Thus, during the motion compensation, the predicted macroblock is constructed.

After the prediction, the resulted macroblock is subtracted from the initial block to produce a residual (difference) block. Then, the residual block is transformed

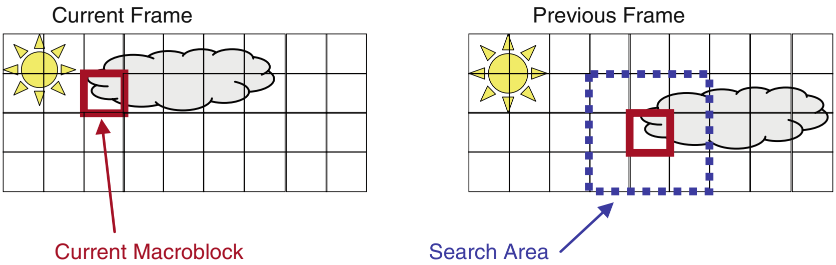


Fig. 1.26 Motion estimation

(T) and quantized (Q) to give a set of quantized transform coefficients, which are reordered (reorder) and finally entropy encoded (CABAC in Fig. 1.25).

The H.264 uses two types of transforms (T): Hadamard and DCT transforms, depending on the residual data that are to be encoded. The DCT operates on X, a block of $N \times N$ samples, typically the residual values after prediction. The equation for the DCT computation is the following:

$$Y = AXA^T$$

where X is the matrix of samples, Y is the matrix of resulting coefficients, and A is an $N \times N$ transform matrix. The elements of the matrix A are

$$A_{ij} = C_i \cos \frac{(2j+1)i\pi}{2N},$$

where

$$C_i = \sqrt{\frac{1}{N}}(i = 0),\ C_i = \sqrt{\frac{2}{N}}(i > 0)$$

The H.264 supports DCT on 4×4 block thus the value of N is 4. After, the transformation, a quantizer step is used for the division of the DCT coefficients. The quantizer step size can be different for the luma and chroma components and it is indexed by a quantizer parameter with values in range 0–51. After the quantization, the next step is the reordering, when each 4×4 block of quantized transform coefficients is mapped to a 16-element array in a zigzag order. Afterward, this element array is entropy encoded.

There are two types of entropy encoder: CAVLC (context adaptive variable length coder) and CABAC (context adaptive binary arithmetic coding). CAVLC is the method used to encode residual, zigzag-ordered 4×4 and 2×2 blocks of the transform coefficients. It takes advantage of the sparse nature of the blocks after the prediction, transformation, and quantization, containing mostly zero values. CAVLC uses run-level coding to compactly represent strings of zeros. CABAC is the second method for entropy encoding, which achieves good performance through (i) selecting probability models for each syntax element according to the element's context, (ii) adapting probability estimates based on local statistics, and (iii) using arithmetic coding.

The entropy-encoded coefficients together with the information required to decode each macroblock (prediction modes, quantizer parameters, motion vector information, etc.) form the compressed bitstream. This compressed bitstream is passed to a network abstraction layer (NAL) for transmission or storage of the encoded image.

During the encoding process, the H.264 algorithm decodes (reconstructs) the macroblock to provide a reference for further predictions. The quantized transform coefficients are scaled (Q^{-1}) and inverse transformed (T^{-1}) to produce a difference block. The equation of the IDCT (inverse discrete cosine transform) is the following:

$$X = A^T YA$$

where A is the same transform matrix used in DCT computation and Y is the matrix of the DCT coefficients.

After this step, the obtained difference block is added to the predicted macroblock to create the reconstructed block. Then, a filter is applied to reduce the effects of blocking distortion. The filter smoothes block edges, improving the appearance of the decoded frame. The filtered macroblocks are used for motion-compensated prediction of further frames in the encoder, resulting in a smaller residual after prediction. After that, the reconstructed reference picture is created from a series of blocks.

The H.264 standard supports seven sets of capabilities, which are referred as profiles, targeting specific class of applications. The most used profiles for MPSoC implementation are the baseline, main profile, and extended profile. Potential applications of the baseline profile include video-telephony, videoconferencing, and wireless communications. Potential applications of the main profile include television broadcasting and video storage. The extended profile may be particularly useful for streaming media applications.

In this book, the main profile will be used as application case study. The main profile includes support for interlaced video, which means that not the entire image is compressed, but only every second line, i.e., the odd lines of the first image and the even lines of the second image. The main profile supports both methods of entropy coding: CAVLC and CABAC. It also permits inter-prediction with the use of two reference picture lists, characteristic to the B type of slices.

1.8 Conclusions

Recently, the SoC architectures have been moved from single processing core based architectures to multiple-processor embedded systems. Moreover, processors' heterogeneity became adopted in the multimedia processing domain, where different application functions have different execution requirements.

This chapter defined the main steps and concepts used in programming this kind of application-specific heterogeneous MPSoC architectures. Some of the fundamental concepts defined were MPSoC, hardware–software interface representation at different abstraction levels, the concept of combined application/architecture model, the execution model of the mixed hardware/software representation using different design languages, etc.

Programming MPSoC represents a gradual software design process performed in several steps corresponding to different abstraction levels (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype). An efficient software code requires taking into consideration the characteristics of the architecture to achieve performance. These steps are application partitioning and mapping on the target architecture, corresponding to the system architecture design, communication mapping on the hardware resources, equivalent to the virtual architecture design, software code adaptation to specific implementation of the communication protocol, also called transaction-accurate architecture design, and final binary generation and memory map decision, corresponding to the virtual prototype design. The software validation is performed by simulation using an abstract architecture model. The execution model is expressed in Simulink and SystemC design languages.

The chapter also gave the specification of three applications from the multimedia domain, namely, the token ring, the motion JPEG decoder, and the H.264 encoder that will be used as case studies in the remaining part of the book. The applications are aimed to run on three different heterogeneous and complex MPSoC architectures. These architectures are the following: the 1AX architecture, made of two processors (ARM7 and Xtensa), the Diopsis RDT tile made of one DSP and one ARM9 microcontroller, and the Diopsis R2DT SoC, made of two DSPs and one ARM9 interconnect by a NoC.

The following chapters will define the basic components of the MPSoC hardware and software architecture. Then, each chapter of the book will detail the software design and validation at each of the adopted MPSoC abstraction levels (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype).

Chapter 2
Basics

Abstract This chapter presents the basic components of the MPSoC hardware and software architecture. The MPSoC hardware architecture is made of several interconnected hardware and software subsystems. Each software subsystem executes a specific software stack. The software stack has a layered organization composed of application tasks, operating system, communication, and hardware abstraction layer. This chapter gives the definition of these different hardware and software components of MPSoC.

2.1 The MPSoC Architecture

System–on–Chip (SoC) represents the integration of different computing elements and/or other electronic subsystems into a single integrated circuit (chip). It may contain digital, analog, mixed-signal, and often radio-frequency functions – all on one chip.

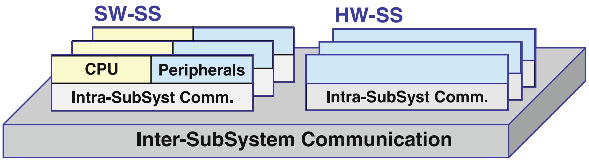
Multi-processor System–on–Chip (MPSoC) are SoC that may contain one or more types of computing subsystems, memories, input/output devices (I/O), and other peripherals. These systems range from portable devices such as MP3 players, videogame consoles, digital cameras, or mobile phones to large stationary installations like traffic lights, factory controllers, engine controllers for automobiles, or digital set-top boxes.

The MPSoC architecture is made of three types of components: software subsystems, hardware subsystems, and inter-subsystem communication, as illustrated in Fig. 2.1.

The *hardware subsystems* (HW-SS) represent custom hardware subsystems that implement specific functionality of an application or global memory subsystems. The HW-SS contain two types of components: intra-subsystem communication and specific hardware components. The hardware components implement specific functions of the target application or represent global memories accessible by the computing subsystems. The intra-subsystem communication represents the communication inside the HW-SS between the different hardware components. This can

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_2,

Fig. 2.1 MPSoC architecture



be in form of a small bus (collection of parallel wires for transmitting address, data, and control signals) or point-to-point communication links.

The *software subsystems* (SW-SS) represent programmable subsystems, also called processor nodes of the architecture. The SW-SS include computing resources, intra-subsystem communication, and other hardware components, such as local memories, I/O components, or hardware accelerators. The computing resources represent the processing units or CPUs. The *CPU* (central processing unit) also known as processor core, processing element, or shortly processor executes programs stored in the memory by fetching their instructions, examining them, and then executing them one after another [150]. There are two types of SW-SS: single core and multi-core. The single-core SW-SS includes a single processor, while the multi-core SW-SS can integrate several processor cores in the same subsystem, usually of same type. The intra-subsystem communication represents the communication inside the SW-SS, e.g., local bus, hardware FIFO, point-to-point communication links, or other local interconnection network used to interconnect the different hardware components inside the SW-SS.

The *inter-subsystem communication* represents the communication architecture between the different software and hardware subsystems. This can be a hardware FIFO connecting multiple subsystems or a scalable global interconnection network, such as bus or network on chip (NoC). Despite most of the buses, the NoC allows simultaneous data transfers, being composed of several links and switches that provide means to route the information from the source node to the destination node [42].

Homogeneous MPSoC architectures are made of identical software subsystems incorporating the same type of processors. In the *heterogeneous MPSoC* architectures, different types of processors are integrated on the same chip, resulting in different types of software subsystems. These can be GPP (general-purpose processor) subsystems for control operations of the application; DSP (digital signal processor) subsystems specially tailored for data-intensive applications such as signal processing applications; or ASIP (application-specific instruction set processor) subsystems with a configurable instruction set to fit specific functions of the application.

The different subsystems working in parallel on different parts of the same application must communicate each other to exchange information. There are two distinct MPSoC designs that have been proposed and implemented for the communication models between the subsystems: shared memory and message passing [42].

The *shared memory* communication model characterizes the homogeneous MPSoC architecture. The key property of this class is that communication occurs implicitly. The communication between the different CPUs is made through a global shared memory. Any CPU can read or write a word of memory by just executing LOAD and STORE instructions. Besides the common memory, each processor code may have some local memory which can be used for program code and those items that need not be shared. In this case, the MPSoC architecture executes a multithreaded application organized as a single software stack.

The *message-passing* organization assumes multiple software stacks running on identical or non-identical software subsystems. The communication between different subsystems is generally made through message passing. The key property of this class is that the communication between the different processors is explicit through I/O operations. The CPUs communicate by sending each other message by using primitives such as *send* and *receive*. There are three types of message passing: synchronous (if the sender executes a *send* operation and the receiver has not yet executed a *receive*, the sender is blocked until the receiver executes the *receive*), buffered or asynchronous blocking (when a message is sent before the receiver is ready, the message is buffered somewhere, for example, in a mailbox, until the receiver takes it out; thus the sender can continue after a *send* operation, if the receiver is busy with something else), and asynchronous non-blocking (the sender may continue immediately after making the communication call) [150].

Heterogeneous MPSoC generally combines both models to integrate a massive number of processors on a single chip [122]. Future heterogeneous MPSoC will be made of few heterogeneous subsystems where each may include a massive number of the same processor to run a specific software stack [87].

This book considers heterogeneous MPSoC architectures organized as it was illustrated previously in Fig. 2.1 with the support of message-passing communication model.

Besides the hardware architecture previously presented, the MPSoC means also software running on hardware. The major challenge for technical success of MPSoC is to make sure that the software executes efficiently on the hardware [18].

2.2 Programming Models for MPSoC

Several tools exist for automatic mapping of sequential programs on homogeneous multiprocessor architectures. Unfortunately, these are not efficient for heterogeneous MPSoC architectures. In order to allow the design of distributed applications, programming models have been introduced and extensively studied by the software communities to allow high-level programming of heterogeneous multiprocessor architectures.

The programming model specifies how parts of the application running in parallel communicate information to one another and what synchronization operations are available to coordinate their activities. Applications are written in a programming model. The programming model specifies what data can be named by the different parallel processes, what type of operations can be executed on the named data, and what ordering exists between the different operations [42].

Examples of parallel programming models are as follows:

– Shared address space, when the communication is performed by posting data into shared memory locations, accessible by all the communicating processing elements. This programming model also involves special atomic operations for the synchronization and data protection.
– Data-parallel programming, when several processing units perform the same operations simultaneously, but on separate parts of the same data set. The data set has a regular structure, i.e., array or matrix. At the end of the operations, the processes exchange synchronization information globally, before continuing the operations with a new data set.
– Message passing, when the communication is performed between a specific sender and a specific receiver. This involves a well-defined event when the data is sent or received, and these events are the basis for orchestrating the individual activities. Anyhow, there are no shared locations accessible to all processing elements. The most common communication primitives used in message-passing programming model are variants of *send* and *receive*. In its simplest form, *send* specifies a local data buffer that is to be transmitted and a receiving process (typically a remote processor). The *receive* operation specifies a sending process and a local data buffer into which the transmitted data will be placed. The message passing can be further divided into two communication-centric programming models: client–server and streaming.

In the basic client–server model, the communicating processes are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client–server interaction, also known as request–reply behavior is shown in Fig. 2.2. When a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The

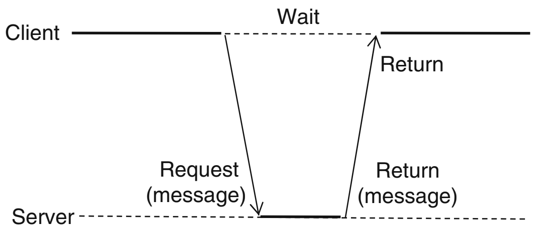


Fig. 2.2 Client–server communication model

message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.

Common object request broker architecture (CORBA) is a well-known specification for distributed systems which adopts an object-based approach for the communication, based on client–server model [116]. All communication takes place by invoking an object. An object provides services, and it combines functional interface as well as data. An object request broker (ORB) connects a client to an object that provides a service. Each object instance has a unique object reference. The client and the object do not need to reside on the same processor; a request to a remote processor can invoke multiple ORBs. The object logically appears to be a single entity, but the server may keep a thread pool running to implement object calls for a variety of clients. Because the client and the object use the same protocol, the object can provide a consistent service independent of the processing element on which it is implemented.

Streaming is a form of communication in which timing plays a crucial role [10]. The support for the exchange of time-dependent information is often formulated as a support for continuous media or stream, e.g., support for reproducing a sound wave by playing out an audio stream at a well specified rate, or displaying a certain number of images per second for a movie. Streams can be simple or complex. A simple stream consists of only a single sequence of data, whereas a complex stream consists of several related simple streams, called substreams. For example, stereo audio can be transmitted by means of a complex stream consisting of two substreams, each used for a single audio channel. It is important, however, that those two substreams are continuously synchronized. Another example of a complex stream is one for transmitting a movie. Such a stream could consist of a single video stream, along with two streams for transmitting the sound of the movie in stereo. The transmission of these data streams can be effectuated in the following:

– Asynchronous mode, when the data items in a stream are transmitted one after the other, but there are no further timing constraints when transmission of these items should take place.
– Synchronous mode, when there is a maximum end-to-end delay defined for each data unit of the stream. In this case, the time required for the data transmission has to be guaranteed to be lower than the maximum permitted delay.
– Isochronous mode, when there is a maximum and minimum end-to-end delay for each data unit of the stream. The end-to-end delays are usually expressed as quality of service (QoS) requirements.

The QoS ensures that the temporal relationship between the streams is preserved. There are several ways to enforce QoS for streaming applications, e.g., by data dropping if the communication network gets congested, or by applying error correction techniques, e.g., encoding the outgoing data units in such a way that any k out of n received data units is enough to reconstruct k correct data units.

StreamIt is an example of programming model for streaming systems [153]. The StreamIt language has mainly two goals: to provide high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain and, second, to serve as a common machine language for grid-based processors. At the same time, the StreamIt compiler aims to perform stream-specific optimizations to achieve high performance.

2.2.1 Programming Models Used in Software

The programming model is usually embodied in a parallel language or a programming environment [42].

As long as only the software is concerned, Skillicorn [145] identifies five key concepts that may be hidden by the programming model, namely, concurrency or parallelism of the software, decomposition of the software into parallel threads, mapping of threads to processors, communication among threads, and synchronization among threads. These concepts define six different abstraction levels for the programming models.

Table 2.1 summarizes the different levels with typical corresponding programming languages for each of them. All these programming models take into account only the software side. They assume the existence of lower levels of software and a hardware platform able to execute the corresponding model.

The programming models are presented in decreasing order of abstraction in the following six categories:

– Programming models that abstract the parallelism completely. Such programming models describe only the purpose of an application and not how it is to achieve this purpose. Software designers do not need to know even if the application will execute in parallel. Such programming levels are abstract and relatively simple, since the applications need to be no more complex than sequential ones.

Table 2.1 The six programming levels defined by Skillicorn

Abstraction level	Typical languages	Explicit concepts
Implicit concurrency	PPP, crystal	None
Parallel level	Concurrent prolog	Concurrency
Thread level	SDL	Concurrency, decomposition
Agent models	Emerald, CORBA	Concurrency, decomposition, mapping
Process network	Kahn process networks	Concurrency, decomposition, mapping, communication
Message passing	MPI, OCCAM	Concurrency, decomposition, mapping, communication, synchronization

– Programming models in which parallelism is made explicit. But the decomposition of the application into threads is still implicit, hence so is the mapping, communication, and synchronization concepts. In such programming models, the software designers are aware that parallelism will be used and must have expressed the potential for it in the application. But they do not know how much parallelism will actually be applied at runtime. Such programming models often require the applications to express the maximal parallelism provided by the algorithm, and then reduce that degree of parallelism to fit the target architecture, while at the same time working out the implications for mapping, communication, and synchronization.
– Programming models in which parallelism and decomposition must both be made explicit, but mapping, communication, and synchronization are implicit. Such programming models require decisions about the breaking up of the application into parallel executed threads, but they relieve the software designer of the implications of such decisions.
– Programming models in which parallelism, decomposition, and mapping are explicit, but communication and synchronization are implicit. In this case, the software developer must not only decompose the application into parallel threads but also consider how best to map the parallel threads on the target processor. Since mapping will often have a marked effect on the communication performance, this almost inevitably requires an awareness of the target processor's interconnection network. It becomes very hard to make such software portable across different architectures.
– Programming models in which parallelism, decomposition, mapping, and communication are explicit, but synchronization is implicit. In this case, the software designer is making almost all of the implementation decisions, except that fine-scale timing decisions are avoided by having the system deal with synchronization.
– Programming models in which all the five concepts are explicit. In this case, the software designers must specify the whole implementation. Thereby, it is extremely difficult to build software using such programming models, because both correctness and performance can only be achieved by attention to vast numbers of details.

2.2.2 Programming Models for SoC Design

In order to allow concurrent hardware/software design, we need to abstract the hardware/software interfaces, including both software and hardware components. Similar to the programming models for software, the hardware/software interfaces may be described at different abstraction levels. The four key concepts that we consider are explicit hardware resources, management and control strategies for the hardware resources, the CPU architecture, and the CPU implementation. These concepts define four abstraction levels described in the previous chapter, namely system architecture level, virtual architecture level, transaction-accurate architecture level,

Table 2.2 Additional models for SoC design

Abstraction level	Typical programming languages	Explicit concepts
System architecture	MPI, Simulink [15]	All functional
Virtual architecture	Untimed SystemC [16]	+Abstract communication resources
Transaction-accurate architecture	TLM SystemC [16]	+Resources sharing and control strategies
Virtual prototype	Co-simulation with ISS	+ ISA and detailed I/O interrupts

and virtual prototype level, as summarized in Table 2.2. The different abstraction levels may be expressed by a single and unique programming model that uses the same or different primitives for each level.

At the system architecture level, all the hardware is implicit similar to the message-passing model used for software. The hardware/software partitioning and the resources allocation are made explicit. This level fixes also the allocation of the tasks to the various subsystems. Thus, the model combines both the specification of the application and the architecture and it is also called combined architecture algorithm model (CAAM). At the virtual architecture level, the communication resources, such as global interconnection components and buffer storage components, become explicit. The transaction-accurate architecture level implements the resource management and control strategies. This level fixes the RTOS on the software side. On the hardware side, a functional model of the bus is defined. The software interface is specified to the HAL level while the hardware communication is defined at the bus transaction level. Finally, the virtual prototype level corresponds to the classical co-simulation with instruction set simulators (ISS). At this level the architecture of the CPU is fixed, but not yet its implementation that remains hidden by an ISS.

Several languages can cover multiple abstraction levels for SoC design, such as C, C++. In fact, most real embedded software at both higher abstraction levels (system architecture) and lower levels uses C/C++ as a stretch. While SystemC, a C++ class, is useful to model behavior and architecture blocks, the behavior is likely to be written in C especially at low level.

2.2.3 Defining a Programming Model for SoC

The use of programming models for the software design of heterogeneous MPSoC requires the definition of new design automation methods to enable concurrent design of hardware and software. This also requires new models to deal with non-standard application-specific hardware/software interfaces at several abstraction levels. The software design makes use of a programming model.

The programming model abstracts the hardware for the software design. It is made of a set of functions (implicit and/or explicit primitives) that can be used by the software to interact with the hardware. Additionally, the programming model needs to cover the four abstraction levels required for the software refinement previously presented (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype). In order to cover different abstraction levels of both software and hardware, the programming model needs to include three kinds of primitives:

– Communication primitives: these are aimed to exchange data between the hardware and the software.
– Task and resource control primitives: these are aimed to handle task creation, management, and sequencing. At the system architecture level, these primitives are generally implicit and built in the constructions of the language. The typical scheme is the module hierarchy in block structure languages, where each module declares implicit execution threads.
– Hardware access primitives: these are required when the architecture includes specific hardware. The primitives include specific primitives to implement specific protocol or I/O schemes, for example, a specific memory controller allowing multiple accesses. These will always be considered at lower abstraction layers and cannot be abstracted using the standard communication primitives.

The programming models at the different abstraction levels previously described are summarized in Table 2.3. The different abstraction levels may be expressed by a single and unique programming model that uses the same primitives applicable at different abstraction levels or it uses different primitives for each level.

Table 2.3 Programming model API at different abstraction levels

Abstraction level	Communication primitives	Task and resource control	HW access primitives
System architecture	Implicit, e.g., Simulink links	Implicit, e.g., Simulink blocks	Implicit, e.g., Simulink links
Virtual architecture	Data exchange, e.g., send/receive (data)	Implicit tasks control, e.g., threads in SystemC	Specific I/O protocols related to architecture
Transaction-accurate architecture	Data access with specific addresses, e.g., read/write (data, adr)	Explicit tasks control, e.g., create/resume task HW management of resources, e.g., test/set	Physical access to HW resources
Virtual prototype	Load/store	HW arbitration and address translation, e.g., memory map	Physical I/Os

2.2.4 Existing Programming Models

A number of MPSoC-specific programming models, based on shared memory or message passing, have been defined recently. Examples of programming models can be considered: OpenMP [33] for shared memory architectures and MPI [108], TTL [160], or YAPI [78] for message passing architectures. This section will detail some of them.

2.2.4.1 Message-Passing Interface (MPI)

The message-passing interface (MPI) is a message-passing library interface specification. The last version 2.2 was recently adopted as standard [108]. It includes the specification of a set of primitives for point-to-point communication with message passing, collective communications, process creation and management, one-sided communications, and external interfaces.

The following APIs represent examples of blocking communication primitives within MPI:

- `MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `MPI_Recv (void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)`

where

- `buf` is the source/destination buffer to be sent/received
- `count` is the number of elements to be sent/received
- `datatype` is the data type of the data to be sent/received (e.g., MPI_CHAR, MPI_INT, MPI_DOUBLE)
- `dest`/`src` represents the rank or identifier of the destination, respectively, source
- `tag` represents the message tag used to distinguish among the messages in case that two communicating partners exchange more than one message
- `comm` identifies the group of the communicator
- `status` indicates the result of the receive operation, whether or not an error occurred during the data transmission

The send call previously presented blocks until the message is copied to the destination buffer. But message buffering can decouple the send and receive operations. This means that the send operation can complete as soon as the message was buffered, even if no matching receive operation was executed by the receiver. Essentially, there are three types of communication modes:

- Buffered mode, when the send operation can start even if no matching receive operation was initiated and it may complete before the corresponding receive

starts. If there is no space in the buffer for the outgoing message, then an error will occur.
– Synchronous mode, when the send operation can start even if no matching receive operation was initiated, but the send will complete successfully only if the matching receive started to receive the message sent by the synchronous send.
– Ready mode, when the send operation can start only if the corresponding receive is already posted.

The primitives associated with these three modes are `MPI_BSend`, `MPI_SSend`, and `MPI_RSend`/`MPI_RRecv`.

MPI defines also non-blocking communication primitives: `MPI_ISend` and `MPI_IRecv` (immediate send and immediate receive). The non-blocking APIs support also the three communication modes: buffered, synchronous, and ready mode and they use the same naming conventions as the blocking type: B for buffered mode, S for synchronous, and R for ready, i.e., `MPI_IBSend`, `MPI_ISSend`, `MPI_IRSend`/`MPI_IRRecv`.

Besides communication, MPI defines also standard APIs for the process management. These are `MPI_Comm_Spawn` and `MPI_Comm_Multiple`, used to start new processes by a running MPI application and establish communication with them.

All these standard primitives are implemented in various libraries. An example of implementation is the MPICH library [109]. This supports three types of devices for the communication: sockets for communication between processing units; mixed socket for communication between processors and shared memory within a multicore processor; and shared memory within an SMP architecture.

2.2.4.2 Multi-core Communications API (MCAPI)

Other research works focus on the standardization of the communication APIs, such as the Multi-core Association working group, which developed the MCAPI (multi-core communications APIs) [98]. The MCAPI defines a set of communication APIs for multi-core communications, to support lightweight, high-performance implementations typically required in embedded applications. MCAPI captures the basic elements of inter-core communications that are required for embedded “closely distributed” systems and scales to support hundreds of processor cores. The potential applications for such an API are extremely varied, but its principal use is the embedded multi-core systems with tight memory constraints and task execution times, requiring reliable on-chip interconnect and high system throughput. Besides the details of the API, the MCAPI specification includes example usage models for multimedia, networking, and automotive applications. MCAPI provides three communication modes: connectionless messages, connected channels for packets, and connected channels for scalars. It also provides functions for endpoint and non-blocking operations management.

2.2.4.3 Y-Chart Application Programmer's Interface (YAPI)

The Y-chart application programmer's interface (YAPI) is an application programmer's interface to write signal and stream processing applications as process networks, developed by Philips Research [78]. The communication between processes is based on Kahn process networks with blocking reads on theoretically unbounded FIFOs.

The Kahn process network is a computational model which consists of a set of concurrent processes [72]. Each of the processes performs sequential computation on its private state space. The processes communicate with each other via uni-directional FIFO channels. A FIFO channel has one input end and one output end, i.e., there is exactly one process that writes to the channel and there is also exactly one process that reads values from the FIFO. The process has input ports which transfer data from the FIFO to the process by *reading values* and output ports which copy data from the process to the FIFO by *writing* values.

YAPI is a C++ library with a set of rules which can be used to model and execute an application as a Kahn process network. The syntax for reading values is `read (p, x)`. This statement reads a value from input port `p` and stores this value in variable `x`. The syntax for writing values is `write (q, y)`. This statement writes the value of variable `y` to output port `q`. YAPI supports also to read and write vectors, not only scalars. The corresponding APIs are `read (p, x, m)`, which reads `m` values from port `p` into array `x`, respectively, `write (q, y, n),` which writes `n` values of array `y` to the port `q`.

2.2.4.4 Task Transaction Level (TTL)

The task transaction level interface (TTL) proposed in [160] is derived from YAPI and focuses on stream processing applications in which concurrency and communication are explicit. The interaction between tasks is performed through communication primitives with different semantics, allowing blocking or non-blocking calls, in order or out of order data access, and direct access to channel data. The TTL APIs define three abstraction levels. The `vector_read` and `vector_write` functions are typical system level functions, which combines synchronization with data transfers. The `reAcquireRoom` and `releaseData` functions (`re` stands for relative) grant/release atomic accesses to vectors of data that can be loaded or stored out of order, but relative to the last access, i.e., with no explicit address. This corresponds to virtual architecture level APIs. Finally, the `AcquireRoom` and `releaseData` lock and unlock access to scalars, which requires the definition of explicit addressing schemes. This corresponds to the transaction-accurate architecture level APIs.

2.2.4.5 Distributed System Object Component (DSOC)

The Multiflex approach proposed in [122] targets multimedia and networking applications, with the objective of having good performance even for small granularity tasks. Multiflex supports both symmetric multi-processing (SMP) approach used on shared memory multiprocessors and remote procedure call based programming approach, called DSOC (distributed system object component). The SMP functionality is close to the one provided by POSIX, i.e., thread creation, mutexes, condition variables, etc. [28]. The DSOC uses a broker to spawn the remote methods. These abstractions make no separation between virtual architecture and transaction-accurate architecture levels, since they rely on fixed synchronization mechanisms. The hardware support to locks and the run queues management is provided by a concurrency engine. The processors have several hardware contexts to allow context switches in one cycle. DSOC uses a CORBA-like approach but implements hardware accelerators to optimize the performances.

2.2.4.6 Compute Unified Device Architecture (CUDA)

Another example of programming model used in industry is the CUDA architecture provided by Nvidia [115]. CUDA (compute unified device architecture) is a software platform for massively parallel high-performance computing on powerful Nvidia GPUs (graphics processing units). CUDA requires programmers to write special code for parallel processing, but it does not require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU. The architecture of GPUs is hidden beneath APIs. This hardware abstraction has two benefits: first, it simplifies the high-level programming model, insulating programmers from the complex details of the GPU hardware. Second, the hardware abstraction allows flexibility in the GPU architecture. Currently, CUDA aims at data-intensive applications that need single-precision floating-point math, but future perspective envisions a new double precision floating-point GPU.

CUDA's programming model differs significantly from single-threaded CPU code and even the parallel code that some programmers began writing for GPUs before CUDA. In a single-threaded model, the CPU fetches a single instruction stream that operates serially on the data. A superscalar CPU may route the instruction stream through multiple pipelines, but there is still only one instruction stream, and the degree of instruction parallelism is severely limited by data and resource dependencies. Even the best four-way, five-way, or six-way superscalar CPUs struggle to average 1.5 instructions per cycle, which is why superscalar designs rarely venture beyond four-way pipelining. Single-instruction multiple-data (SIMD) extensions permit many CPUs to extract some data parallelism from the code, but the practical limit is usually three or four operations per cycle [115].

Another programming model is general-purpose GPU (GPGPU) processing. This model is relatively new and has gained much attention in recent years. Essentially, developers hungry for high performance began using GPUs as general-purpose processors, although "general purpose" in this context usually means data-intensive applications in scientific and engineering fields. Programmers use the GPU's pixel shaders as general-purpose single-precision FPUs. GPGPU processing is highly parallel, but it relies heavily on off-chip "video" memory to operate on large data sets. (Video memory, normally used for texture maps and so forth in graphics applications, may store any kind of data in GPGPU applications.) Different threads must interact with each other through off-chip memory. These frequent memory accesses tend to limit performance [115].

CUDA takes a third approach. Like the GPGPU model, it is highly parallel. But it divides the data set into smaller chunks stored in on-chip memory, and then allows multiple thread processors to share each chunk. Storing the data locally reduces the need to access off-chip memory, thereby improving performance. Occasionally, of course, a thread does need to access off-chip memory, such as when loading the off-chip data it needs into local memory. In the CUDA model, off-chip memory accesses usually do not stall a thread processor. Instead, the stalled thread enters an inactive queue and is replaced by another thread that is ready to execute. When the stalled thread's data becomes available, the thread enters another queue that signals it is ready to go. Groups of threads take turn executing in round-robin fashion, ensuring that each thread gets execution time without delaying other threads [115].

2.2.4.7 Open Computing Language (OpenCL)

Open computing language (OpenCL) is an open standard for writing applications that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors, introduced by Khronos working group [81]. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL includes a C-based language for writing *kernels* (functions that execute on OpenCL devices), called also run-time APIs, plus APIs that are used to define and then control the platforms, also known as platform layer APIs. The run-time APIs serve to execute computational or compute kernels and manage scheduling of computational and memory resources. The platform layer APIs represent a hardware abstraction layer over diverse computational resources and are used to query, select, and initialize compute devices in the systems and to create compute contexts and work queues. A compute device is a collection of one or more computational units or cores, which can be a CPU or GPU.

The execution model in OpenCL resides on two concepts: compute kernel and compute program. The compute kernel is the basic unit of executable code, similar to a C function, and it can be data parallel or task parallel. The compute program is a collection of compute kernels and internal functions, similar to a dynamic library. The applications queue the compute kernel execution instances in order. Then, the compute kernel execution instances can be executed in order or out of order. The data-parallel execution is achieved by defining work items that execute in parallel. The work items can be grouped together to form a work group. The work items within a group can communicate with each other and can synchronize their execution to coordinate the memory access. Also, multiple work groups can be executed in parallel. Some compute devices such as CPUs can also execute task-parallel compute kernels as a single work item. The following examples illustrate the usage of OpenCL APIs to create compute programs for the FFT application:

```
//create compute context with GPU device
  context = clCreateContextFromType (CL_DEVICE_TYPE_GPU);
//create a work-queue
  queue = clCreateWorkQueue (context, NULL, NULL, 0);
//allocate memory buffer objects
  memobjs[0]      =      clCreateBuffer      (context,
CL_MEM_READ_WRITE,
  sizeof (float)*2, NULL);
//create the compute program for FFT application
  program = clCreateProgramFromSource (context, 1,
  &fft1D_kernel_src, NULL);
//build the compute program executable
  clBuildProgramExecutable      (program, false, NULL,
NULL);
//create the compute kernel
  kernel = clCreateKernel (program, "fft1D");
```

The memory model used in OpenCL is shared memory model but with multiple distinct address spaces that can be collapsed.

2.2.4.8 Open Multi-processing (OpenMP)

The open multi-processing (OpenMP) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures [117].

The OpenMP assumes a shared memory model, with all the threads having access to the same, globally shared memory. The data needs to be labeled as shared or private. The shared data is accessible by all threads and there is a single instance of the data. Private data can be accessed only by the thread which owns it. The data transfer is transparent to the programmer and the synchronization is mostly implicit.

OpenMP consists of a set of compiler directives to define a parallel region, work sharing, data sharing attributes like shared or private, tasking, etc.; library routines and environment variables (e.g., number of threads, stack size, scheduling type, dynamic thread adjustment, nested parallelism, active levels, thread limit) that influence the run-time behavior. An example of parallel region which declares two parallel loops is illustrated below:

```
#pragma omp parallel if (n>limit) default (none)
    shared (n,a,b,c,x,y,z) private (f,i,scale)
{
  f = 1.0;
//parallel loop, work is distributed
#pragma omp for nowait
  for (i=0; i<n; i++)
    z[i]=x[i]+y[i];
//parallel loop, work is distributed
```

```
#pragma omp for nowait
  for(i=0; i<n; i++)
    a[i]=b[i]+c[i];
...
// sychronization
#pragma omp barrier
    scale = sum(a,0,n) + sum(z,0,n) + f;
    ...
} // end of parallel region
```

2.2.4.9 Transaction-Level Modeling (TLM)

Another standardization work is related to SystemC transaction-level modeling (TLM) [118]. The open SystemC initiative (OSCI) proposes TLM-2 draft 1 and TLM-2 draft 2, which offers a set of standard APIs and a library that implements a foundation layer upon which interoperable transaction-level models can be built. The standard proposal is designed to facilitate intellectual property (IP) sharing and re-use, faster EDA tool development, and make it a lot easier for electronic OEMs to use TLM. The TLM-2 draft 1 version defines two transaction modeling styles – the untimed programmer's view (PV) model and the PV+T (programmer's view with timing information) model. The new draft adopted recently, TLM-2 draft 2, retires one previously proposed modeling style, the programmer's view plus annotated timing (PV+T), and introduces two new transaction-level modeling styles – loosely timed (LT) and approximately timed (AT). The newly introduced loosely timed (LT) modeling style is suitable for software application development, software performance analysis, and hardware architectural analysis. It employs the flexible non-blocking transport in a lightweight manner and close to untimed performance. The newly proposed approximately timed (AT) modeling style is closer to timed behavior, i.e., in terms of modeling contention and arbitration. Thus, it is suitable for hardware architectural analysis and performance verification. The AT shares the same non-blocking transport used by LT but defines finer-grained timing control. Using a generic payload, it monitors four or more communication events – depending on the protocol – for example, beginning and end of request and beginning and end of response.

2.2.4.10 Other Examples of Programming Models

The authors in [24] introduce the concept of service dependency graph to represent HW/SW interface at different abstraction levels and to handle application-specific API. This model represents the hardware/software interface as a set of interdependent components providing and requiring services.

Cheong et al. [35] propose a programming model called TinyGALS, which combines the locally synchronous with the globally asynchronous approach for programming event-driven embedded systems.

In [176] the authors describe PTIDES (programming temporally integrated distributed embedded systems) programming model. This defines an execution strategy based on discrete-event semantics and then specializes this strategy to give distributed policies for execution scheduling and control.

LLVM stands for low-level virtual machine and it represents a compilation strategy designed to enable software code optimization at compile time, link time, and runtime [92]. It has a virtual instruction set to represent the low-level object code.

In the previous section (Table 2.3), we showed that a suitable programming model for MPSoC needs to be defined at several abstraction levels corresponding to different design steps. This hierarchical view of the programming model ensures a seamless implementation of high-level APIs onto the low-level ones. In order to ensure a better match between the programming model and the underlying hardware architecture, the APIs also have to be extensible at each abstraction level, to cope with the broad range of possible hardware components. The existing MPSoC programming models seem to focus either on one aspect or on the other. We think that it is important to consider both aspects, i.e., hierarchy and extensibility when designing an MPSoC-oriented programming model.

2.3 Software Stack for MPSoC

The software running on the MPSoC architecture is called *embedded software*. The software costs are often a large part of the total cost of an embedded system and are characterized by different performance requirements [69].

Often, the performance requirement in an embedded application is a *real-time* requirement. A real-time performance requirement is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more sophisticated requirement exists: the average time for a particular task is constrained as well as the numbers of instances when some maximum time is exceeded. Such approaches (sometimes called *soft real time*) arise when it is possible to occasionally miss the time constraints on an event, as long as not too many are missed. Real-time performances tend to be highly application dependent.

Two other key characteristics exist in many embedded applications: the need to *minimize the memory* and the need to *minimize the power*. Sometimes the application is expected to fit completely in the memory of the processor on chip; other times the application needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application's algorithm. Large memories also mean more power [69].

2.3.1 Definition of the Software Stack

In this book, the software running on the software subsystems is called *software stack*. In heterogeneous MPSoC architectures, each software subsystem executes a software stack. The software stack is made of two components: the application tasks code and the hardware-dependent software (HdS). The HdS layer is made of three components: the operating system (OS), specific I/O communication software, and the hardware abstraction layer (HAL). The HdS is responsible to provide application- and architecture-specific services, i.e., scheduling the application tasks, communication between the different tasks, external communication with other subsystems, hardware resource management and control. The following paragraphs detail the software stack organization, including all these different components.

2.3.2 Software Stack Organization

The software stack is structured in different software layers that provide specific services. Figure 2.3 illustrates the software stack organization in two layers: application layer and HdS (hardware-dependent software) layer. In the first section, the application layer will be presented and then the HdS will be defined.

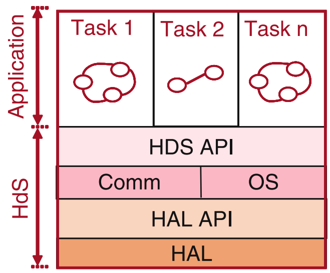


Fig. 2.3 Software stack organization

2.3.2.1 Application Layer

The application layer may be a multi-tasking description or a single task function of the application targeted to be executed on the software (processor) subsystem. A *task or thread* is a lightweight process that runs sequentially and has its own program counter, register set, and stack to keep track of where it is. In this book, the terms task and thread are used as interchangeable terms. Multiple tasks can be executed in parallel by a single CPU (single-core) or by multiple CPUs of the same type grouped in the software subsystem (multi-core). The tasks may share the same resources of the architecture, such as processors, I/O components, and memories. On a single processor core node, the multithreading generally occurs by time slicing, wherein a single processor switches between different threads. In this case, the processing is not literally simultaneous, as the single processor is doing only one thing at a time. On a multi-core processor subsystem, threading can be achieved via multiprocessing, wherein different threads can run literally simultaneously on different processors inside the software node [148].

The application layer consists of a set of tasks that makes use of programming model or application programming interface (API) to abstract the underlying HdS software layer. These APIs corresponds to the HdS APIs.

2.3.2.2 HdS Layer

The HdS layer represents the software layer which is directly in contact with, or significantly affected by, the hardware that it executes on, or can directly influence the behavior of that hardware [127]. The HdS integrates all the software that is directly depending on the underlying hardware, such as hardware drivers or boot strategy. It also provides services for resource management and sharing, such as scheduling the application tasks on top of the available processing elements, inter-task communication, external communication, and all other kinds of resource management and control. The federative HdS term underlines the fact that, in an embedded context, we are concerned with application-specific implementations of these functionalities that strongly depend on the target hardware architecture [87].

Current research studies proved that the HdS debug represents 78% of the global system total debugging time of an MPSoC software design cycle [175]. This may due to incorrect configuration or access to the hardware architecture, e.g., a wrong configuration of the memory mapping for the interrupt control registers. In order to reduce its complexity, the HdS is structured into three software components: operating system (OS), communication management (Comm), and hardware abstraction layer (HAL).

Operating System

The operating system (OS) is the software component that manages the sharing of the resources of the architecture. It is responsible for the initialization and management of the application tasks and communication between them. It provides services such as tasks scheduling, context switch, synchronization, and interrupt management. In the following, more details about these OS services will be given.

The tasks scheduling service of the OS usually follows a specific scheduling algorithm. Finding the optimal algorithm for the tasks scheduling represents an NP-complete problem [162]. There are different categories of scheduling algorithms. The classic criteria are hard real time versus soft real time or non-real time; preemptive versus cooperative; dynamic versus static; and centralized versus distributed [148].

Contrary to non-real time, the real-time scheduler must guarantee the execution of a task in a certain period of time. Hard real time must guarantee that all deadlines are met.

Preemptive scheduling allows a task to be suspended temporally by the OS, for example, when a higher priority task arrives, resuming later when no higher priority tasks are available to run. This is associated with time sharing between the tasks. Examples of preemptive scheduling algorithms are round-robin, shortest remaining time, or rate monotonic schedulers. The cooperative or non-preemptive scheduling algorithm runs each task to its completion. In this case, the OS waits for a task to surrender control. This is usually associated with event-driven operating systems. Examples of non-preemptive algorithm are the shortest job next or highest response ratio next.

With static algorithms, the scheduling decisions (preemptive or non-preemptive) are made before run-time. Contrary to static algorithms, the dynamic schedulers make their scheduling decisions during the execution.

The implementation of the scheduler may be centralized or distributed. In case of a centralized scheduler implementation, the scheduler controls all the task execution ordering and communication transactions. In case of a distributed scheduler implementation, the scheduler distributes the control decision to the local task schedulers corresponding to each processor [38].

When a task is ready for execution and it is selected by the scheduler of OS according to the scheduler algorithm, the OS is also responsible to perform the context switch between the currently running task and the new task. The context switch represents the process of storing and loading the state of the CPU in order to share the available hardware resources between different tasks. The state of the current task, including registers, is saved, so that in case the scheduler gets back for execution of the first task, it can restore its state and continue normally.

In order to ensure a correct runtime and communication order between the different tasks running on parallel, synchronization is required. The tasks can synchronize by using semaphores to control access to shared resource or by sending/receiving synchronization signals (events) to each other. The mutex is a binary semaphore which ensures mutual exclusion on a shared resource, such as a buffer shared by two threads, by locking and unlocking it whenever the resource is accessed by a task [149, 150].

The interrupt handler is another OS service used for interrupts management. There are two types of processor interrupts: hardware and software. A hardware interrupt causes the processor to save its state of execution via a context switch and begins the execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set of the processor, which cause a context switch to an interrupt handler similar to a hardware interrupt. The interrupts represent a way to avoid wasting the processor's execution time in polling loops waiting for external events. Polling means when the processor waits and monitors a device until the device is ready for an I/O operation.

Examples of commercial OS are the eCos [46], FreeRTOS [51], LynxOS [93], VxWorks [170], WindowsCE [104], or μITRON [158].

Communication Software Component

The second software component of the HdS layer constitutes the communication component, which is responsible to manage the I/O operations and more generally the interaction with the hardware components and the other subsystems. The communication component implements the different communication primitives used inside a task to exchange data between the tasks running on the same processor or between the tasks running on different processors. It may include different communication protocols, such as FIFO (first in–first out) implemented in software, or communication using dedicated hardware components. If the communication requires access to the hardware resources, the communication component invokes primitives that implement this kind of low-level access. These function calls are done in form of HAL APIs.

The HAL APIs allow for the OS and communication components to access the third component of the software stack, that is, the HAL layer.

Hardware Abstraction Layer

Low-level details about how to access the resources are specified in the hardware abstraction layer (HAL) [174]. The HAL is a thin software layer which not only totally depends on the type of processor that will execute the software stack but also depends on the hardware resources interacting with the processor. The HAL includes the device drivers to implement the interface for the communication with the device. This includes the implementation of drivers for the I/O operations or for other peripherals. The HAL is responsible also for processor-specific implementations, such as loading the main function executed by an OS, more precisely the boot code, or the implementation of the *load* and *restore* of CPU registers during a context switch between two tasks, but also code for the configuration and access to the hardware resources, e.g., MMU (memory management unit), timer, interrupt enabling/disabling, etc.

The structured representation of the software stack in several layers (application tasks, OS, communication, and HAL), as previously described, has two main advantages: flexibility in terms of software components re-use by changing the OS or the communication software components, and portability to other processor subsystems by changing the HAL software layer.

2.4 Hardware Components

2.4.1 Computing Unit

The microprocessor, also known as a *CPU*, central processing unit, computing unit, or just processor, is a complete computation engine that is fabricated on a single chip. A chip is also called an integrated circuit. Generally it is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few thousand transistors etched onto a chip just a few millimeters square.

Generally, the microprocessors provide the following characteristics:

- The *date* is the year that the processor was first introduced. Many processors are re-introduced at higher clock speeds for many years after the original release date.
- *Transistors* is the number of transistors on the chip. The number of transistors on a single chip has risen steadily over the years.

– *Microns* is the width, in microns, of the smallest wire on the chip. As the feature size on the chip goes down, the number of transistors rises.
– *Clock speed* is the maximum rate that the chip can be clocked at.
– *Data width* is the width of the ALU (arithmetic and logic unit), which is the main component of the processor. An 8-bit ALU can add/subtract/multiply/etc., two 8-bit numbers, while a 32-bit ALU can manipulate 32-bit numbers. An 8-bit ALU would have to execute four instructions to add two 32-bit numbers, while a 32-bit ALU can do it in one instruction. In many cases, the external data bus is the same width as the ALU, but not always. For instance, the 8088 Intel processor had a 16-bit ALU and an 8-bit bus, while the modern Pentium processors fetch data 64 bits at a time for their 32-bit ALUs [85].
– *MIPS* stands for "millions of instructions per second" and is a rough measure of the performance of a CPU.

The microarchitecture of the CPU is comprised of five basic components: memory, registers, buses, the ALU, and the control unit. Each of these components is pictured in Fig. 2.4.

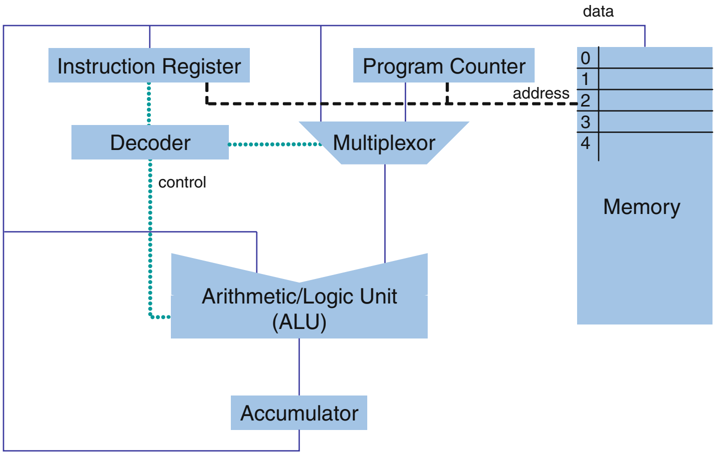


Fig. 2.4 CPU microarchitecture

– *Memory:* this component is created from combining latches with a decoder. The latches create circuitry that can store information, while the decoder creates a way for individual memory locations to be selected.
– *Registers*: these components are special memory locations that can be accessed very fast. Three registers are shown in the figure: the instruction register (IR), the program counter (PC), and the accumulator.
– *Buses*: these components are the information highway for the CPU. Buses are bundles of tiny wires that carry data between components. The three most important buses are the address, the data, and the control buses.

– *ALU*: this component is the number cruncher of the CPU. The *a*rithmetic/*l*ogic *u*nit performs all the mathematical calculations of the CPU, including add, subtract, multiply, divide, and other operations on binary numbers.
– *Control Unit*: this component is responsible for directing the flow of instructions and data within the CPU. The control unit is actually built of many other selection circuits such as decoders and multiplexors. In the diagram, the decoder and the multiplexor compose the control unit.

A microprocessor executes a collection of machine instructions that tell the processor what to do. Based on the instructions, a microprocessor does three basic activities:

– Using its ALU (arithmetic/logic unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication, and division. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large floating-point numbers.
– A microprocessor can move data from one memory location to another.
– A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

To support these basic activities, the processor architecture includes the following:

– An address bus (that may be 8, 16, or 32 bits wide) that sends an address to memory
– A data bus (that may be 8, 16, or 32 bits wide) that can send data to memory or receive data from memory
– An RD (read) and WR (write) line to tell the memory whether it wants to set or get the addressed location
– A clock line that lets a clock pulse sequence the processor
– A reset line that resets the program counter to zero (or whatever) and restarts execution

The processor can perform a large set of instructions. The collection of instructions is implemented as bit patterns, each one of which has a different meaning when loaded into the instruction register. A set of short words are defined to represent the different bit patterns. This collection of words is called the *assembly language* of the processor. An *assembler* can translate the words into their bit patterns very easily, and then the output of the assembler is placed in memory for the microprocessor to execute.

Examples of assembly language instructions for the Intel x86 processors are as follows [85]: ADC (add operation with carry), ADD (add operation), AND (logical AND operation), CLI (clear interrupt flag), CMP (compare operands), DEC (decrement by 1), DIV (unsigned divide operation), IN (input from data port), INC (increment by 1), INT (call interrupt), JMP (jump), LEA (load effective address operation), MOV (move), MUL (unsigned multiplication operation), NOT (logical NOT operation), OR (logical OR operation), PUSH (push data into stack), RET (return from procedure), SHL (shift left operation), SUB (subtraction operation), or XOR (exclusive OR logical operation). The list of instructions that can be executed by a processor is called instruction set architecture, shortly ISA.

Based on the instruction set style, the processors can be classified in CISC (complex instruction set computer) and RISC (reduced instruction set computer). The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very small memory capacity is required to store instructions. The complex instructions are built directly into the hardware.

RISC processors only use simple instructions that can be executed within one clock cycle. Because there are more lines of code, more memory is needed to store the assembly-level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general-purpose registers. Because all of the instructions execute in a uniform amount of time (i.e., one clock), pipelining is also possible.

Another type of classification of the processors takes into account the amount of data being processed and the number of instructions being executed. The Flynn's taxonomy divides the processors into four categories [50]:

- Single instruction, single data (SISD), also known nowadays as a RISC processor. In this case, a single stream of instructions operates on a single set of data. Instructions are executed sequentially, but may be overlapped by pipelining. Most of the SISD systems are now pipelined.
- Single instruction, multiple data (SIMD). These machines include several interconnected processing elements, each with its own data, but under the supervision of a single control unit. All the processing elements perform the same operations on their data in lockstep. Thus, the execution of the instructions is synchronous. A single program counter can be used to describe the execution of all the processing elements.
- Multiple instruction, multiple data (MIMD). In this case, several processing elements have their own data and their own program counters. The tasks executed by different processors can start or finish at different times. The programs do not have to run in lockstep.
- Multiple instruction, single data (MISD). These machines have many processing elements, all of which execute independent stream of instructions, but on the same data stream.

2.4.1.1 General-Purpose Processor

The general-purpose processors can be found in laptop, desktop, or server computers. These processors have very high performance and designed to work well in a variety of contexts. They support most of the popular Windows, Linux, and real-time operating systems. They run a wide range of application systems and are relatively inexpensive for high-end applications [59].

An example of well-known general-purpose processor with both desktop and embedded system markets is x86 from Intel [85].

2.4.1.2 Application-Specific Instruction Set Processor

An application-specific instruction set processor (ASIP) is a stored memory CPU whose architecture is tailored for a particular set of applications. This specialization of the processor core provides a trade-off between the flexibility of a general-purpose CPU and the performance of a DSP. The ASIP exploits special characteristics of the application to meet the desired performance, cost, and power requirements. The programmability of the ASIP allows changes to the implementation, use in several different chips, and high data path utilization. The application-specific architecture provides smaller silicon area and higher computation speed [171].

Compared to general-purpose processors, usually the ASIPs are enhanced with the following features:

- Special-purpose registers and buses to provide the required computations without unnecessary generality.
- Special-purpose function units to perform long operations in fewer clock cycles.
- Special-purpose control for instructions to execute common combinations in fewer clock cycles.

Some ASIPs have a configurable instruction set. Usually, these cores are divided into two parts: *static* logic, which defines a minimum ISA, and *configurable* logic which can be used to design new instructions. The configurable logic can be programmed either in the field in a similar fashion to an FPGA, dynamically reconfigured during execution, or during the chip synthesis.

Generally, the ASIP design relies on automatic tools. Usually these tools start from a set of characteristics of the target application domain and the required execution profiling (e.g., number of execution clock cycles for a specific application function). Then, the automatic tools generate both micro-architecture for the ASIP core and an optimized compiler targeted to the synthesized ASIP. Finally, the application is implemented using the generated ASIP core and ASIP compiler. Thus, the ASIP design consists of two main steps: processor synthesis and compiler design. The processor synthesis consists of choosing an instruction set, optimizing the data path, and extracting the instruction set from the register transfer design. The compiler design consists of driving the compilation from a parametric description of the data path, binding values to registers, selecting instructions for the code matched to the parameterized architecture, and scheduling the processor instructions.

Example of ASIP processors is the Xtensa processor from Tensilica [152]. The Xtensa processors are synthesizable processors that are configurable and extensible. The processor can be configured to fit the application by selecting and configuring predefined elements of the architecture, such as the following:

– *Instruction set*: ALU extensions, co-processors, wide instructions, DSP style, function unit implementation
– *Memory*: instruction cache configuration, data cache configuration, memory protection/translation, address space size, mapping of special purpose memories, DMA access
– *Interface*: bus width, bus access protocol, system registers access, JTAG, queue interfaces to other processors
– *Peripherals*: timers, interrupts, exceptions, and remote debug procedures

Additionally, the designers can optimize the processor by inventing completely new instructions and hardware execution units that can deliver high performance. The new instruction sets can be defined using the TIE language, which offers support to new state declarations, new instruction encodings and formats, and new operation descriptions. The TIE instructions can be manually written or automatically generated by using the XPRESS compiler, a tool which identifies which functions of a C/C++ application need to be accelerated in hardware.

Once the designer determines the optimal configuration and extensions, the Xtensa processor generator automatically generates a synthesizable hardware description as well as a complete optimized software development environment [152].

An example of commercial ASIP design tool is the Coware Processor Designer, which represents an integrated design environment for unified application-specific processor, programmable accelerator design, and software development tool generation [41].

The key to processor design's automation is its Language for Instruction Set Architectures, shortly LISA. In contrast to SystemC, which has been developed for efficient specification of systems, LISA is a processor description language that incorporates processor-specific components, such as register files, pipelines, pins, memory and caches, and instructions. The LISA language enables the efficient creation of a single "golden" processor specification as the source for the automatic generation of the instruction set simulator (ISS) and the complete suite of software development tools, like assembler, linker, and C-compiler, and synthesizable RTL code. An example of LISA code is illustrated in Fig. 2.5.

The development tools, together with the extensive profiling capabilities of the debugger, enable rapid analysis and exploration of the application-specific processor's instruction set architecture to determine the optimal instruction set for the

```
OPERATION add
{
  DECLARE  { GROUP dest, src1, src2 = { register }; }
  CODING   { dest  src2  src1  0b01000010000 }
  SYNTAX   { "ADD"  ".D"  src1  ","  src2  ","  dest }
  BEHAVIOR { dest = src1 + src2; }
}

OPERATION register
{
  DECLARE    { LABEL index; }
  CODING     { index:0bx[4] }
  SYNTAX     { "A"  index:#U }
  EXPRESSION { A[index] }
}
```

Fig. 2.5 Sample LISA modeling code

target application domain. Processor designer enables also the designer to optimize instruction set design, processor micro-architecture and memory subsystems, including caches [41].

Another example of commercial ASIP design tool is provided by Target Compiler Technologies IP designer tool suite [151]. The design starts from descriptions of the processor architecture and the instruction set, using a high-level definition language, called nML. Then, based on the ASIP description and the targeted application, the Chess tool automatically maps the C application into optimized machine code of the target ASIP. The Checkers tool generates automatically the instruction set simulator and the graphical software debugger. The Go tool of the tool suite produces the synthesizable RTL architecture model of the ASIP from the nML processor description. Darts is used as assembler and disassembler of the ASIP that translates the machine code into binary format and vice versa.

2.4.1.3 Digital Signal Processor

The digital signal processor (DSP) is a specialized microprocessor designed specifically for digital signal processing [5].

Digital signal processing algorithms typically require a large number of mathematical operations to be performed quickly on a set of data. Signals are converted from analog to digital, manipulated digitally, and then converted again to analog form. Many DSP applications have constraints on latency; that is, for the system to work, the DSP operation must be completed within some time constraint. Most general-purpose microprocessors and operating systems can execute DSP algorithms successfully. But these microprocessors are not suitable for application of mobile telephone and pocket PDA systems, etc., because of power supply and space limit. A specialized digital signal processor, however, will tend to provide a lower cost solution, with better performance and lower latency.

The architecture of a digital signal processor is optimized specifically for digital signal processing applications. They usually provide parallel multiply and add operations, multiple memory accesses to fetch two operands and store the result, lots of registers to hold data temporarily, efficient address generation for array handling, and special features such as delays or circular addressing.

Examples of commercial DSPs are provided by Freescale [52], Texas Instruments [156], or Atmel [9].

A key player at the high-end signal processor manufacturer is Freescale. The company provides the fully programmable MSC8144 multi-core DSP architecture, which is based on next-generation StarCore™ technology (Fig. 2.6) [52]. Freescale's MSC8144 is helping to make IP-based connections faster, easier, and more reliable by taking advantage of one programmable DSP platform that supports VoIP/data, video, and wireless standards using multiple software implementations. The MSC8144 DSP combines four programmable SC3400 StarCore™ DSP cores. Each SC3400 StarCore™ DSP core runs at 1 GHz. The quad-core DSP delivers the equivalent performance of a 4 GHz single-core DSP [52].

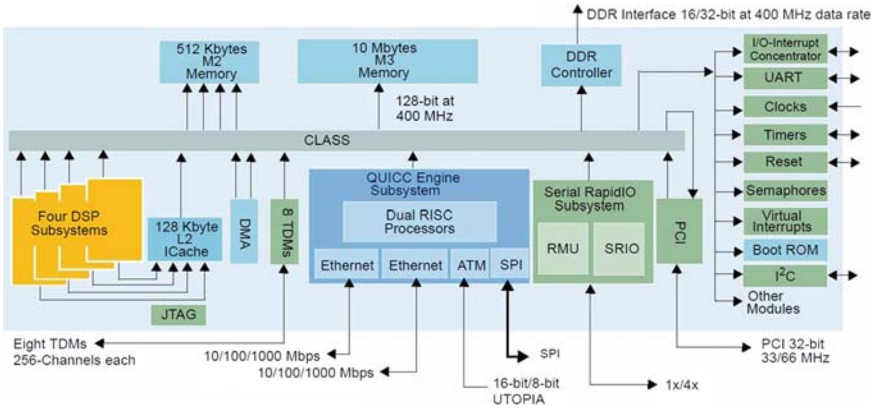


Fig. 2.6 The Freescale MSC8144 SoC architecture with quad-core DSP

Other examples of DSP are the C6000 DSP series from Texas Instruments. These series are fix point DSPs, which may operate at the clock frequency up to 1.2 GHz. They implement separate instruction and data caches with 2 MB second-level cache (L2). The access to the I/O is fast thanks to its 64 DMA (direct access memory) channels. The top models are capable of even 8,000 MIPS (million instructions per second), use VLIW (very long instruction word) encoding, perform eight operations per clock cycle, and are compatible with a broad range of external peripherals and various buses (PCI/serial/etc.) [156].

2.4.1.4 Microcontroller

The microcontroller (MCU) is a special processor used for control functions. Microcontrollers are embedded inside consumer products, so that they can control the features or actions of the product. Another name for a microcontroller, therefore, is *embedded controller*. The microcontrollers are dedicated for the execution of one application task. Usually they are small and low power devices.

A microcontroller may have many low power modes, depending on the application. There are numerous methods that the microcontroller can use to lower the static power consumed by the devices in standby or low power mode (also called standby or leakage power). These include low-leakage transistors and turning off the power to various parts of the MCU. Usually, the deeper asleep the device is, the longer it takes to wake up. Wakeup time becomes an important consideration when determining how low power modes are implemented.

Some microcontrollers provide digital signal processing functionality. Processor cores for control are different from those that perform very complex mathematical functions. Cores that perform both functions are blurring that line.

In some cases, DSP-like mathematical functions are being added to a regular core's instruction set, with the hardware to support it. And the opposite is occurring too, as DSP cores add control-like instructions.

An alternative is to embed both a controller and a DSP core in the same device, creating a hybrid. Whether these devices are considered microcontrollers with DSP functionality or DSPs with microcontroller functionality is up to the vendor to decide.

Examples of microcontrollers are the 16-bit MSP430 from Texas Instruments [156], AVR from Atmel [9], the PIC microcontroller families from Microchip [103], or the 32-bit ARM Cortex-M3 or the 8-bit 8051 [6] or those provided by Freescale for automotive applications [52].

2.4.2 Memory

The memory is a hardware component used to store data. The basic unit of storage is called memory cell [171]. Cells are arranged in a 2D array to form the memory circuit. Within the memory core, the cells are connected to row and bit (column) lines that provide a 2D addressing structure. The row line selects a one dimensional row of cells, which then can be accessed (written or read), via their bit lines. The memory may have multiple ports to accept multiple addresses and data for simultaneous read and write operations.

The memories can be classified into two main types: RAM (random access memory) and ROM (read-only memory). Traditional RAM memories store data that can be read and written in a random order. They are usually volatile memories, meaning that their content is lost after turning off the power. The ROM memories store data that cannot be modified (at least not quickly or easily). They are mainly used to store firmware, software very closely tied to the hardware.

The RAM family includes two important memory devices: static RAM (SRAM) and dynamic RAM (DRAM). The primary difference between them is the lifetime of the data they store. SRAM retains its contents as long as electrical power is applied to the chip. If the power is turned off or lost temporarily, its contents will be lost forever. DRAM, on the other hand, has an extremely short data lifetime – typically about few milliseconds, even when power is applied constantly [14]. The DRAM can behave like SRAM if a piece of hardware called a DRAM controller is used. The job of the DRAM controller is to periodically refresh the data stored in the DRAM. By refreshing the data before it expires, the contents of memory can be kept alive for as long as they are needed.

SRAM devices offer extremely fast access times (approximately four times faster than DRAM) but are much more expensive to produce. Generally, SRAM is used only where access speed is extremely important. A lower cost per byte makes DRAM attractive whenever large amounts of RAM are required. Many embedded systems include both types: a small block of SRAM (a few kilobytes) along a critical data path and a much larger block of DRAM (perhaps even megabytes) for other types of data.

The NVRAM (non-volatile RAM) is an SRAM memory with a battery backup. When the power is turned on, the NVRAM operates like an SRAM. When the power is turned off, the NVRAM uses the battery to retain its data. NVRAM is common in embedded systems. However, it is more expensive than SRAM, because of the battery. So the applications are typically limited to the storage of a few hundred bytes of system-critical information that cannot be stored in any other type of memory.

Memories in the ROM family are distinguished by the methods used to write new data to them (usually called programming) and the number of times they can be rewritten. This classification reflects the evolution of ROM devices from hardwired to programmable to erasable and programmable. A common feature of all these devices is their ability to retain data and programs forever, even during a power failure.

The very first ROMs were hardwired devices that contained a preprogrammed set of data or instructions. The contents of the ROM had to be specified before chip production, so the actual data could be used to arrange the transistors inside the chip. Hardwired memories are still used, though they are now called "masked ROMs" to distinguish them from other types of ROM. The primary advantage of a masked ROM is its low production cost. Unfortunately, the cost is low only when large quantities of the same ROM are required.

Another type of ROM is the PROM (programmable ROM), which is purchased in an unprogrammed state (the data are made up entirely of bits with value equal to 1). The process of writing your data to the PROM involves a special piece of equipment, called device programmer. The device programmer writes data to the device one word at a time by applying an electrical charge to the input pins of the chip. Once a PROM has been programmed in this way, its contents can never be changed. If the code or data stored in the PROM must be changed, the current device must be discarded. As a result, PROMs are also known as one-time programmable (OTP) devices.

An EPROM (erasable-and-programmable ROM) is programmed in exactly the same manner as a PROM. However, EPROMs can be erased and reprogrammed repeatedly. To erase an EPROM, the device is exposed to a strong source of ultraviolet light. Thus, the entire chip is reset to its initial and unprogrammed state. The EPROMs are more expensive than PROMs, but they are essential for the software development and testing process.

EEPROM (electrically erasable-and-programmable ROM) is similar to EPROM, but the erase operation is accomplished electrically, rather than by exposure to ultraviolet light. Any byte within an EEPROM may be erased and rewritten. Once written, the new data will remain in the device forever or until it is electrically erased. The primary trade-off for this improved functionality is higher cost, though write cycles are also significantly longer than writes to a RAM.

Flash memory combines the best features of the memory devices described thus far. Flash memory devices are high density, low cost, non-volatile, fast (to read, but not to write), and electrically reprogrammable. The use of flash memory has increased dramatically in embedded systems. From a software point of view, flash and EEPROM technologies are very similar. The major difference is that flash devices can only be erased one sector at a time, not byte by byte. Typical sector sizes are in the range of 256 bytes to 16 kB. Despite this disadvantage, flash is much more popular than EEPROM.

Another frequently used type of memory is the cache. The cache plays a key role in reducing the average memory access time of a processor. It also decreases the bandwidth requirement each processor places on the shared interconnect and memory hierarchy. The cache is located near the processor, thus it offers a very fast access time (Fig. 2.7). It replicates parts of the data stored in the main memory, i.e., the most often used or the latest used memory blocks, depending on the cache policy. In cache-based SoC, every time a processor requires data from the main memory, first it checks whether the data are already in the cache. In this case, called *cache hit*, the data are directly retrieved from the cache, thus avoiding the transfer through the global interconnect component. If the data are not stored in the cache, situation called *cache miss*, the data are retrieved from the main memory and eventually stored in the local cache for a latter access.

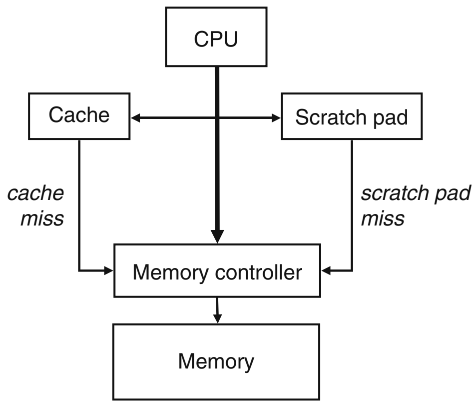


Fig. 2.7 Cache and scratch pad memory

But the use of cache memories raises another important issue: the cache coherence. The cache coherence problem appears in architectures made of multiple processors with their own cache memories. The problem arises when a memory block is present in the caches of one processor or more processors, and another processor modifies that memory block. Unless special action is taken, the other processors continue to access the old copy of the block that it is in their caches [42]. The algorithms used to maintain the cache coherency are implemented in hardware. The hardware decides when values are added or removed from the cache [171].

As an alternative to the hardwired algorithms to manage close-in memory, software-oriented approaches have been proposed. The scratch pad memories represent another type of memory, similar to cache memories, but which require software algorithms for their management (Fig. 2.7). The scratch pad memory is a high-speed memory, located near the processor. It does not include hardware to manage its contents. The CPU can access the scratch pad to read and write directly, because the scratch pad is part of the address space of the main memory. Therefore, its access time is predictable, unlike cache accesses. The software is responsible to manage which data are in the scratch pad or which data need to be removed. Usually the software manages the scratch pad by combining compile-time information with run-time decision making.

Memory is a key bottleneck in embedded systems. Many embedded computing applications spend a lot of their time accessing memory. The memory hierarchy is a prime determinant of not only performance but also energy consumption. So optimizing the memory system becomes crucial. There are several techniques of memory optimization which target either data or instructions, e.g., loop transformations to express data parallelism, dataflow transformations to improve memory utilization, minimal buffer size usage, or optimal scratch pad memory allocation algorithms. The access time of the main memory can be reduced by using burst modes to access a sequence of memory locations, or by integrating paged memories to take advantage of the properties of memory components to reduce the access times, or banked memories, which are systems of memory components that allow parallel accesses.

2.4.3 Interconnect

The interconnect component is a shared resource between various hardware components. Its role is to transmit data from a source hardware resource to a destination hardware resource, thus implementing the communication network.

The network component can be profiled using two types of measurements: *latency* and *bandwidth*. The latency represents the total time required to transfer *n*-bytes of information from the source to the destination component. The bandwidth represents the amount of data (number of bytes) that can be delivered by the communication network per second. It is desirable for the interconnection network to provide high bandwidth, because higher bandwidth decreases the occupancy of the shared interconnect component. Thus, it can reduce the likelihood of network contention. High bandwidth also allows for the software to exchange large volume of data without waiting for individual data units to be transmitted along the way.

The performance constraints for MPSoC architectures place a requirement on the data bandwidth the network must deliver, if the processors are to sustain a given computational rate. However, this load varies considerably between the applications. The flow of information may be physically localized or dispersed. The data may be transmitted in burst mode or fairly uniformly in time. In addition, the waiting time of the processor is strongly affected by the latency of the network. The time spent waiting affects the bandwidth requirement.

The design of the interconnect component is a complex process for system-on-chip. Any communication failure, whether due to noise or an error in timing or protocol, is likely to require a design iteration that will be expensive in both mask charges and time to market [53].

Early SoCs used an interconnect paradigm inspired by the rack-based microprocessor systems of earlier days. In those rack systems, a backplane of parallel connections formed a "bus" into which all manner of cards could be plugged. A system designer could select cards from a catalogue and simply plug them into the rack to yield a customized system with the processor, memory, and interfaces.

In a similar way, a designer of an early SoC could select hardware IP blocks, place them onto the silicon, and connect them together with a standard on-chip bus (OCB) (Fig. 2.8). The backplane might not be apparent as a set of parallel wires on the chip, but logically the solution is the same. The on-chip bus connects a central processor and standard components like memory, peripherals, interrupt units plus some application-specific components. Among the advantages of this approach we have power savings, higher integration density, lower systems costs, easier procurement, etc.

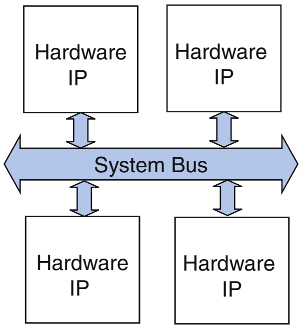


Fig. 2.8 SoC architecture based on system bus

Among the different approaches to the interconnect concept, Virtual Socket Initiative Alliance (VSIA) [168] merits special remark for the effect in standardizing criteria for concepts, methods, and allowed interoperability.

The most popular SoC approach has been the ARM processor strategy [6]. ARM disposes of a complete family of RICS processors, with the AMBA OCB. AMBA (Advanced Microcontroller Bus Architecture) is currently one of the most widely used systems bus architectures for SoC applications (even for processors other than ARM).

However, buses do not scale well. With the rapid increase in the number of hardware components to be connected and the increase in performance demands, today's SoCs cannot be built around a single bus. Instead, complex hierarchies of buses are used (as illustrated in Fig. 2.9), with sophisticated protocols and multiple bridges between them. The communication between remote hardware IPs can go via several buses. Thus, a SoC can implement more than one bus.

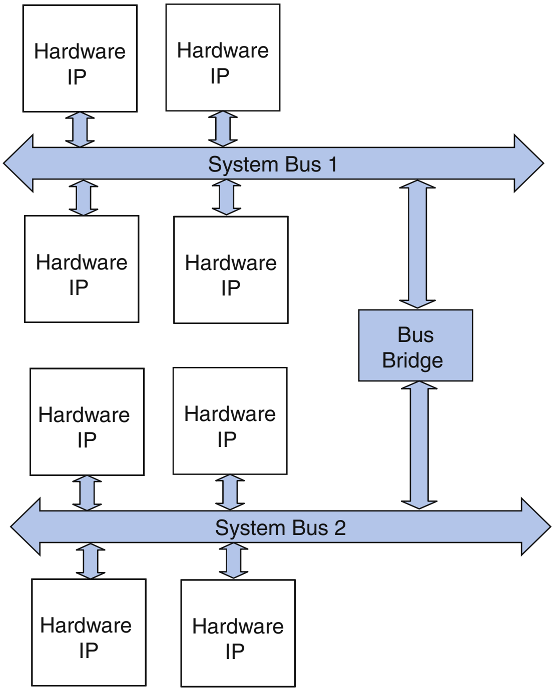


Fig. 2.9 SoC architecture based on hierarchical bus

A typical SoC architecture is comprised of one high-performance bus for memory access and high-speed peripherals and one low-speed bus for low-speed peripherals like UARTs. Examples of buses that can be linked through bridges are those defined within the AMBA specification: advanced high-performance bus (AHB), advanced system bus (ASB), and advanced peripheral bus (APB) [6].

AHB bus is devoted to high-performance communication, for system modules requiring high clock frequencies. This bus acts as the backbone bus. It is intended for connection of processors and coprocessors, DSP units, on-chip memories, and off-chip external memory interfaces. The ASB is mainly deprecated and it has been substituted by AHB. The APB bus is devoted to low-speed peripherals and it is optimized to minimize power consumption and to reduce interface complexity. APB is designed to be used in conjunction with a system bus (AHB/ASB).

The hierarchical bus-based systems have no limitations about the number of buses and its hierarchy. These systems are more flexible and powerful than single bus systems, as they allow any number of CPUs.

Where bus-based solutions reach their limit, packet-switched networks are poised to take over (Fig. 2.10) [53]. A packet-switched network offers flexibility in topology and trade-offs in the allocation of resources to clients. A network on chip (NoC) is constructed from multiple point-to-point data links interconnected by switches (also called routers). The data messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches [17]. A NoC is similar to a modern telecommunications network, using digital bit-packet switching over multiplexed links. Although packet-switching is sometimes claimed as necessity for a NoC, there are several NoC proposals utilizing circuit-switching techniques.

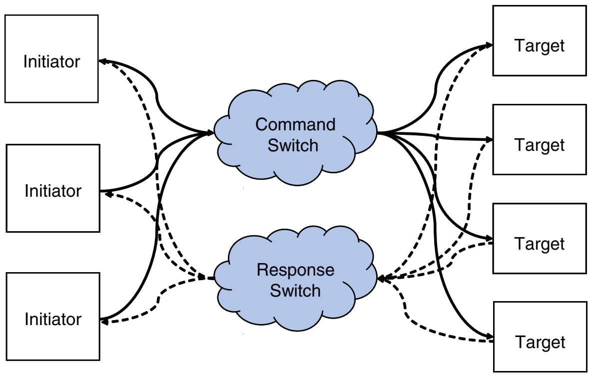


Fig. 2.10 SoC architecture based on packet-switched network on chip

In NoC-based architectures, the IP blocks communicate over the NoC using a five-layered communication scheme, similar to the OSI transmission protocol, as illustrated in Fig. 2.11: application, transport, network, data link, and physical layers.

The application layer corresponds to the communication primitives used by the application for the data exchange. The units of communication consist of data messages.

The transport layer establishes and maintains end-to-end connections. It performs packet segmentation and reassembling and ensures message ordering. The units of communication are packets. The transport layer defines rules that apply as packets are routed through the switch fabric. The packets can include byte enables, parity information, or user information depending on the actual application requirements.

The network layer defines how data are transmitted over the network from a sender to a receiver, such as routing algorithm. The units of communication are flits.

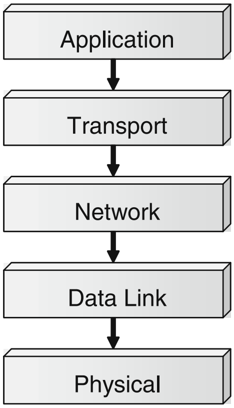


Fig. 2.11 Network-on-chip communication layers

The data link layer defines a protocol to transmit the information between the entities. It may include flow control and error correction. The units of communication in this layer are expressed in bits or words.

The physical layer defines how packets are physically transmitted over an interface. It also determines the number and length of wires connecting the IP blocks and switches. The units of communication at this level are electronic signals.

Examples of NoC interconnect components are Spidergon [40] or the Hermes NoC [107].

2.5 Software Layers

Programmable hardware components are important in a re-usable architectural platform, since it is very cost-effective to tailor a platform to different applications by simply adapting the low-level software and maybe only configuring certain hardware parameters, such as memory sizes and peripherals.

As illustrated in Fig. 2.3, the software view of an embedded system shows three different layers:

– The bottom layer, named hardware abstraction layer or shortly HAL, is comprised of services directly provided by hardware components (processor and peripherals) such as instruction sets, memory and peripheral accesses, and timers. It also includes instance of device drivers, boot code, parts of a real-time operating system (RTOS), such as context-switching code and configuration code to access the MMU (memory management unit), and even some domain-oriented algorithms that directly interact with the hardware.
– The top layer is the multitasking application software, which should remain completely independent from the underlying hardware platform.
– The middle layer is comprised of three different components:

(a) Hardware-independent software, typically high-level RTOS services, such as task scheduling or interrupt service routines;
(b) Communication layer, which implements the high-level communication primitives and offers support for specific I/Os;
(c) The API (application programming interface), which defines a system platform that either isolates the application software from the hardware-dependent software (HdS) (HdS API) or separate the middle layer from all basic low software layer (HAL APIs), enabling their concurrent design.

The standardization of these APIs, which can be seen as a collection of services usually offered by an operating system, is essential for software re-use above and below it. At the application software level, libraries of re-usable software IP components can implement a large number of functions that are necessary for developing systems for given application domains. If, however, one tries to develop a system by integrating application software components that do not directly match a given API, software retargeting to the new platform will be necessary [166]. This can be a very tedious and error-prone manual process, which is a candidate for an automatic software synthesis technique.

Nevertheless, re-use can also be obtained below the API. Software components implementing the hardware-independent parts of the RTOS can be more easily re-used, especially if the interface between this layer and the HAL layer is standardized. Although the development of re-usable HAL may be harder to accomplish, because of the diversity of hardware platforms, it can be at least obtained for platforms aimed at specific application domains.

There are many academic and industrial alternatives providing RTOS services. The problem with most approaches, however, is that they do not consider specific requirements for SoC, such as minimizing memory usage and power consumption. Recent research efforts propose the development of application-specific RTOS containing only the minimal set of functions needed for a given application [55] or including dynamic power management techniques. Embedded software design methodologies should thus consider the generation of application-specific RTOS that are compliant to a standard API and optimized for given system requirements.

The hardware-dependent software part of the software stack is usually comprised of three components called (from lower layer to upper layer) hardware abstraction layer, middleware (or communication layer), and operating system (Fig. 2.3). These three components manage the execution of tasks running on processors and the use of shared resources, including hardware resources. The lowest layer, the hardware abstraction component, is separated from the operating system to facilitate the porting of the OS from one processor to another, which facilitates the porting of an application from one processor to another. Usually, few parts of the HAL are written in assembly, as they are closely linked with the underlying processor. The middleware component includes communication primitives used by the application tasks through the OS, and based on device drivers provided by the HAL component.

This structure by layers (or components) maintains the separation of skills for the development of the software stack. The HAL is usually provided with the hardware or developed by someone who knows deeply the hardware. The OS is developed depending on the main characteristics awaited (scheduling algorithm, real time, symmetric multi-processor, etc.). The communication component makes the link between OS and HAL. The application is developed by the system engineer.

2.5.1 Hardware Abstraction Layer

In the following section, several examples of existing commercial HAL are given. These examples of HAL are used in both academic and semiconductor industry areas.

Even if the HAL represents an abstraction of the hardware architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of the existing HAL is OS dependent. In case of an OS-dependent HAL, the HAL is often called board support package (BSP). In fact, the BSP implements specific support code for a given hardware platform or board, corresponding to a given OS. The BSP includes also a boot loader, which contains a minimal device support to load the OS and device drivers for all the devices on the hardware board.

The embedded version of the Windows OS, namely, Windows CE, provides BSP for many standard development platforms that support several microprocessors family (ARM, x86, MIPS) [104]. The BSP contains an OEM (original equipment manufacturer) adaptation layer (OAL), which includes a boot loader for initializing and customizing the hardware platform, device drivers, and a corresponding set of configuration files.

The VxWorks OS offers BSP for a wide range of MPSoC architectures, which may incorporate ARM, DSP, MIPS, PowerPC, SPARC, Xscale, and other processors family [170].

In eCos, a set of well-defined HAL APIs are presented [46]. However, there is no clear difference between HAL and device driver. Examples of HAL APIs used by eCos are as follows:

- Thread context initialization: HAL_THREAD_INIT_CONTEXT()
- Thread context switching: HAL_THREAD_SWITCH_CONTEXT()
- Breakpoint support: HAL_BREAKPOINT()
- GDB support: HAL_SET_GDB_REGISTERS(), HAL_GET_GDB_REGISTERS()
- Interrupt state control: HAL_RESTORE_INTERRUPTS(), HAL_ENABLE_INTERRUPTS(), HAL_DISABLE_INTERRUPTS()
- Interrupt controller management: HAL_INTERRUPT_MASK()
- Clock control: HAL_CLOCK_INITIALIZE(), HAL_CLOCK_RESET(), HAL_CLOCK_READ()

– Register read/write: HAL_READ_XXX(), HAL_READ_VECTOR_XXX(), HAL_WRITE_XXX(), and HAL_WRITE_ VECTOR_XXX()
– Control the dimensions of the instruction and data caches: HAL_ XCACHE_SIZE(), HAL_XCACHE_LINE_SIZE()

In the software development environment for the Nios II processor provided by Altera [3], the HAL serves as a device driver package, providing a consistent interface to the system peripherals, such as timers, Ethernet MAC, and I/O peripherals.

In Real-Time Linux a HAL, called real-time HAL (RTHAL), is defined to give an abstraction of the interrupt mechanism to the Linux kernel [135]. It consists of three APIs for disabling and enabling interrupts and return from the interrupt.

An example of HAL that does not depend on the targeted OS is the a386 library [1]. The a386 represents a C library which offers an abstraction of the Intel 386 processor architecture. The functions of the library correspond to privileged processor instructions and access to the hardware. The library serves as a minimal hardware abstraction layer for the OS. Later, the library is ported on ARM and SPARC processors.

2.5.2 Operating System

The complexity of the application and the increasing capabilities of hardware architectures have moved embedded software from simple sequential program to concurrent complex system with specific software architecture. This is why embedded systems require some kind of OS to manage the processor (or several processors) and hardware resources efficiently. The OS can be seen as an abstraction of the hardware (processor and resources) used by the software tasks. This abstraction consists in sharing these hardware resources available between tasks and goes to complete virtualization of the hardware architecture for which the number of resources is infinite.

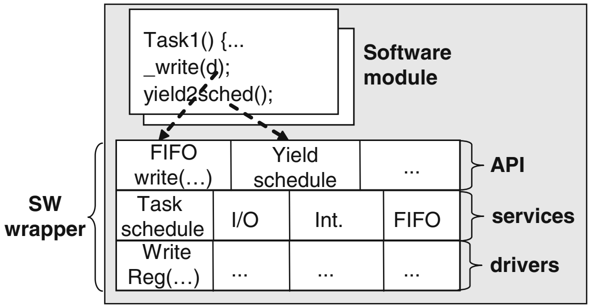
A multiprocessor architecture can run one OS (usually called centralized OS) in only one processor (the others are seen as co-processor) in heterogeneous or asymmetric architecture or in any of the free processors in a symmetric architecture. If one OS is running on each processor (distributed OS), it could be the same copy (homogeneous architecture) or a different copy (heterogeneous architecture). This leads to different performances and different capabilities the software designer has to deal with.

A lot of commercial OSs (see section 2.3.2.2) exist, but an OS can be developed for a particular class of applications or with specific features. In both cases, the software designer has to configure the OS in case of embedded systems to reduce the memory footprint including communication and synchronization services (such as messages, semaphores, clocks) and tasks scheduling policy.

The ASOG (application-specific operating system generator) tool of the ROSES flow [32] allows generation of an application-specific OS, also known as software interfaces or software wrappers, for each processor. The following paragraphs give more details about this OS generator tool.

As shown in Fig. 2.12, the software wrappers provide the implementation of the high-level communication primitives (available through the APIs) used in the system specification and the drivers to control the hardware. If required, the wrapper will also provide sophisticated OS services, such as tasks scheduling and interrupt management, minimally tailored for a particular application.

Fig. 2.12 Software wrapper



The synthesis of wrappers is based on libraries of basic modules from which dedicated OSs are assembled. These libraries may be easily extended with modules that are needed to build software wrappers for any type of processors, memories, and other components that follow various bus and core standards.

The software wrapper generator [55] produces a custom OS for each processor on the target platform. The software wrapper generator produces operating systems streamlined and pre-configured for the software module(s) that run(s) on each target processor. It uses a library organized in three parts: APIs, communication/system services, and device drivers. Each part contains elements that will be used in a given software layer in the generated OS. The generated OS provides services: communication services (e.g., FIFO communication), I/O services (e.g., AMBA bus drivers), memory services (e.g., cache or virtual memory usage), etc. Services have dependency between them, for instance, communication services are dependent on I/O services. Elements of the OS library also have dependency information. This mechanism is used to keep the size of the generated OS at a minimum; the elements that provide unnecessary services are not included.

There are two types of service codes: re-usable (or existing) code and expandable code. As an example of existing code, AMBA bus-master service code can exist in the OS library in the form of C language. As an example of expandable code, OS kernel functions can exist in the OS library in form of macrocode (m4 like). There are several preemptive schedulers available in the OS library such as round-robin scheduler, or priority-based scheduler. In the case of round-robin scheduler, time slicing (i.e., assigning different CPU load to tasks) is supported. To make the OS kernel very small and flexible (1) the task scheduler can be selected from the requirements of the application code and (2) a minimal amount (less than 10% of kernel code size) of processor-specific assembly code is used (for context switching and interrupt service routines).

The software component interfaces must be composed using basic elements of the software wrapper generators library. Table 2.4 lists some API functions available for different kinds of software task interfaces. The application tasks must communicate through API functions provided by the software wrapper generator library. For instance, the shared memory API (SHM) provides read/write functions for inter-task communication. The guarded shared memory API (GSHM) adds semaphore services to the SHM API by providing lock/unlock functions.

Table 2.4 Software communication APIs

Basic component interfaces	API functions
Register	Put/get
Signal	Sleep/wakeup
FIFO	Put/get
SHM	Read/write
GSHM	Lock/unlock/read/wrrite

A recurrent problem in library-based approaches is library size explosion. In the ROSES flow, this problem is minimized by the use of layered library structures where a service is factorized so that its implementation uses elements of different layers. This scheme increases re-use of library elements since the elements of the upper layers must use the services provided by the elements in the immediate lower layer [32]. The designers are able to extend the ROSES libraries since they are implemented in an open format. This is an important feature since it enables the support of different standards while re-using most of the basic elements in the libraries.

The OS generation consists of assembling the required OS services. The services reside in a library made of a set of macrocodes files corresponding to each OS service. The macrocode files are written using a macrolanguage and serve to generate the customized files for the application-specific OS.

The services of the OS are assembled based on a service dependency graph, used to determine, select, and configure the services for the OS generation. The service dependency graph is described using a structural description language, called LiDeL (*l*ibrary *d*escription *l*anguage) [55]. LiDeL is composed of a set of data structures manipulated by several APIs. The structural description language contains three types of items:

– *Elements*, which represent an OS part. The elements are basic components of an OS. They represent a non-specialized component, which is not yet dedicated for a particular architecture case.

– *Services*, which represent system functionality. It is an abstract term, which allows dividing and structuring the behavior of an OS. The *services* are provided by *elements*, but an *element* may also require a *service* from another *element.*
– *Implementations,* which represent a particular behavior description. An *element* can have multiple *implementations*. Each *implementation* corresponds to a part of the generic code of an OS.

The ASOG tool uses as input the system description, more precisely the partitioning and mapping information, the tasks code, the LiDeL library, and the services library written as macrocode. The Colif description contains the services needed by the application, along with the parameters needed for these services (Fig. 2.13) [32].

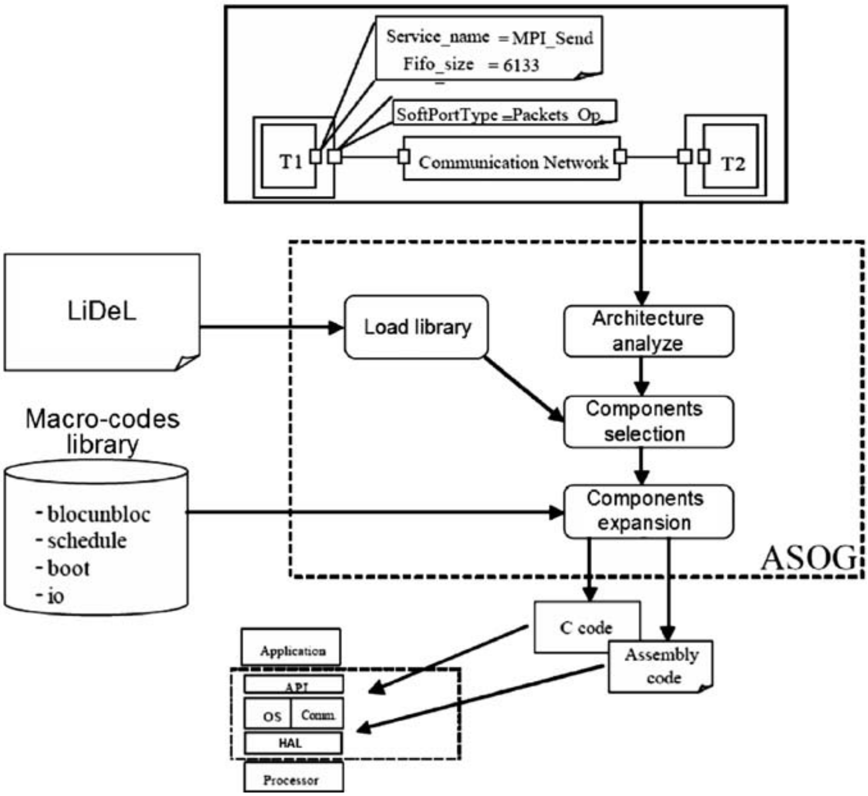


Fig. 2.13 Representation of the flow used by ASOG tool for OS generation

When application tasks require a *service* (i.e., services for the data exchange in the form of MPI communication), the ASOG tool starts crossing the service dependency graph from the required *service* down to the low-level *services*. Based on the crossed *services*, the ASOG will macrogenerate the files for the *implementations* of the *elements* associated with these *services*. The generated files are C or assembly code files. The ASOG also generates the required compilation Makefile scripts, along with some log files useful for debugging the OS generation process.

Table 2.5 shows some of the existing software components in the current ROSES IP library and gives the type of the communication they use in their interfaces.

Table 2.5 Sample software IP library

IP		Description	Interfaces
SW	host-if	Host PC interface	Register/signal
	Rand	Random number generator	Signal/FIFO
	mult-tx	Multipoint FIFO data transmission	FIFO
	reg-config	Register configuration	Register/FIFO/SHM
	shm-sync	Shared memory synchronization	SHM/signal
Stream	FIFO data streaming	GSHM/FIFO/signal	

Figure 2.14 shows the "stream" software IP and part of its code to demonstrate the utilization of the communication APIs. Its interface is comprised of four ports: two for the FIFO API (P3 and P4), one for the signal API (P2), and one for the GSHM API (P1). In line 7 of Fig. 2.14, the stream IP uses P1 to lock the access to the shared memory that contains the data that will be streamed. P2 is used to suspend

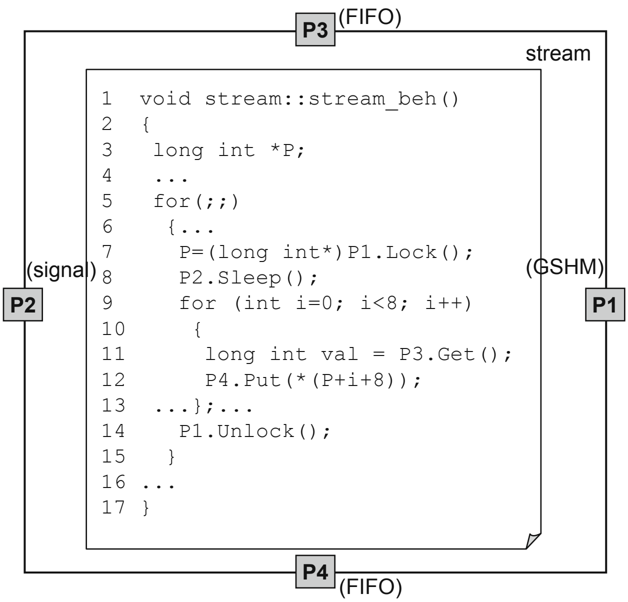


Fig. 2.14 The stream software IP

the task that fills up the shared memory (line 8). Then, some header information is got from the input FIFO using P3 (line 11) and streamed to the output FIFO using P4 (line 12). When streaming is finished, P1 is used to unlock the access to the shared memory (line 14).

2.5.3 Communication and Middleware

The middleware points to the software component that links the application code and the data transfer done by the network. The use of this component or layer helps software designer in porting an application from one architecture to another. In embedded systems, middleware should provide API primitives for several communication schemes (send and receive, blocking or non-blocking, etc.). This API isolates the communication services requested by the application from the network.

The middleware as well as the OS should be configured and tuned specifically for the application/architecture.

2.5.4 Legacy Software and Programming Models

The legacy software and the legacy programming models at both high and low levels can be integrated in the software stack easily. The clear separation between the different software components through well-defined APIs allows using legacy codes. The integration consists of adding new interfaces between the supported components and relying the legacy software on the virtualization technology.

2.6 Conclusions

This chapter detailed the MPSoC hardware and software organization. The definitions of the main hardware components (CPU, memory, interconnect) and software components (applications tasks code, OS, HAL, communication) were given.

The layered organization of the software stack allows a gradual design performed in several steps which correspond to different abstraction levels (system architecture, virtual architecture, transaction accurate architecture, and virtual prototype). The software validation is performed by simulation using an abstract architecture model.

The next chapters will detail the software design and validation of each of these different abstraction levels.

Chapter 3
System Architecture Design

Abstract This chapter presents the system architecture design. The system architecture design consists of partitioning and mapping the application onto the target architecture and mapping the communication onto the available hardware resources. The key contribution in this chapter represents the definition, organization, and design of the system architecture using Simulink, for the Token Ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture, and the H.264 encoder running on the Diopsis R2DT architecture. The functional simulation of the system architecture models allows validation of the applications' algorithm.

3.1 Introduction

As shown in Fig. 3.1, the system architecture design consists of partitioning the application into several parallel tasks and mapping the application tasks onto the target architecture. The result of the system architecture design represents the system architecture model. In this chapter, the mapping process will be defined and then the system architecture model will be presented.

The objectives of the system architecture design are as follows:

– Functional validation of the target application algorithm
– Specification of the application partitioning and mapping onto the hardware architecture.

3.1.1 Mapping Application on Architecture

3.1.1.1 The Mapping

MPSoC design flow starts with two separate models: architecture and application [90] similar to the Y-chart [75], as illustrated in Fig. 3.2. Usually, the description of the application functionality and hardware topology is independent of one other. The architecture is specified as a set of processor and hardware subsystems that interact via communication network.

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_3,

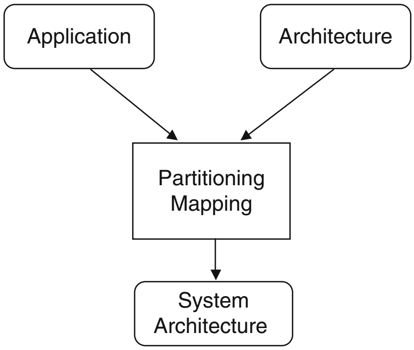


Fig. 3.1 System architecture design

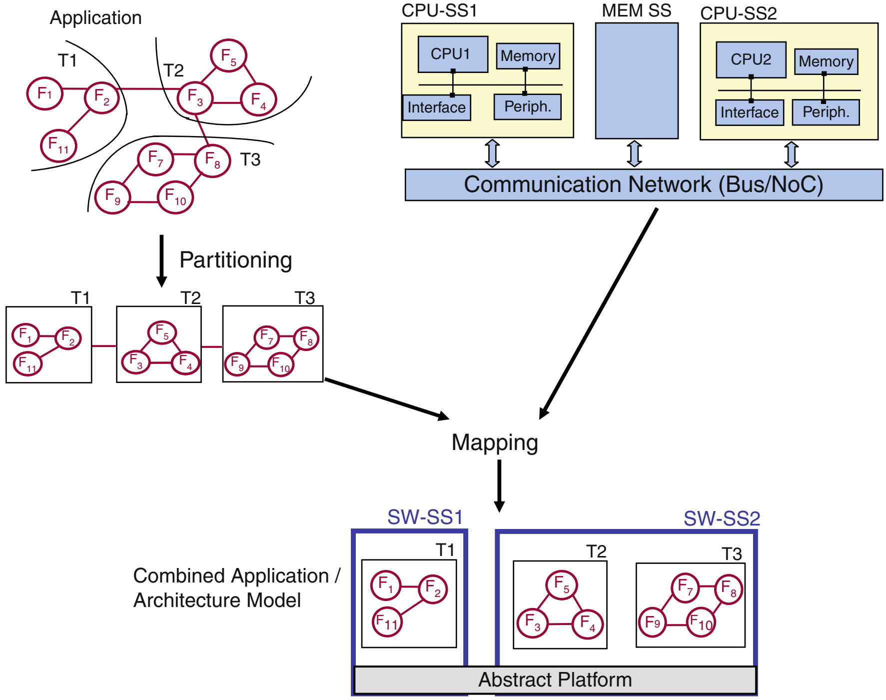


Fig. 3.2 Mapping process

The application is generally specified as a functional model made of a set of multiple functions. Then, the functions are grouped into tasks in order to identify the parts of the application which can be done in parallel. This corresponds to the partitioning step. A parallel software is comprised of multiple cooperating tasks, each of which performs a subset of functions of the application. In the case that the initial model of the application is sequential, e.g., sequential C code, a parallelization step is required.

The parallelization process determines how the computation, data access or input/output operations, and data can be distributed among different processing elements [42]. It also determines which parts of the application will be implemented in software and which parts in hardware. The parallelization of the application consists in dividing the computation in several pieces that can be executed in parallel. These different pieces group several functions of the application and are named tasks or processes. The parallelization mechanism is called partitioning. A parallel application decreases the total execution time of the application compared to its sequential execution [122]. The partitioning step is quite difficult to be optimized in a general case [163] and it will not be considered in this book.

The partitioned model of the application will be mapped on the target architecture. The different tasks running in parallel may be executed by different processors. The number of tasks does not have to be the same with the number of processors available in the architecture. If the number of processors available on the target architecture is less than the number of tasks, more than one task may be executed on the same processor. The assignment of tasks to a target processor that will execute them is called mapping.

The mapping represents the association between the tasks and the processing elements on which they are executing, and the association between the buffers used for the communication between the tasks and the hardware communication resources of the architecture [49]. The mapping should ensure a balanced distribution of the computation over the processors in order to meet the design constraints, e.g., the permitted overhead of the communication, synchronization, and parallelism management [42].

The output of the mapping represents the assignment of the application functions to the architectural units [102]. This corresponds to the system architecture model.

Example 1. Mapping token ring application on 1AX This paragraph illustrates the partitioning, mapping, and binding processes for the token ring application onto the 1AX architecture. Figure 3.3 shows the correlation between the application functions and the architecture elements.

In this example, the functions of the application are grouped into three tasks. Each task corresponds to a token node. The first two tasks (T1 and T2) are mapped on the ARM7 processor, while the third task T3 is mapped onto the XTENSA processor.

Figure 3.3 shows also the allocation of the communication buffers onto the hardware communication resources. Thus, the communication buffer used for the data exchange between the two tasks mapped on the ARM7 processors T1 and T2 is mapped onto the local memory SRAM of the ARM subsystem. The communication buffers used between the tasks T1 and T3, respectively, tasks T2 and T3, which correspond to the communication between the two processors, are mapped on the global memory.

The result of the mapping process is the system architecture model of the token ring application mapped on the 1AX architecture.

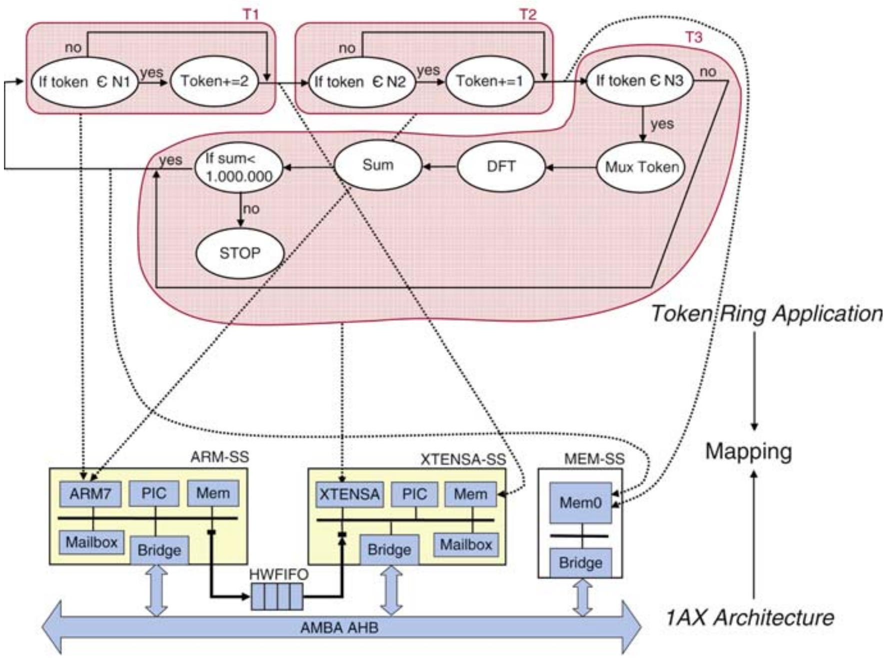


Fig. 3.3 Mapping token ring on the 1AX architecture

3.1.1.2 The Design Space Exploration

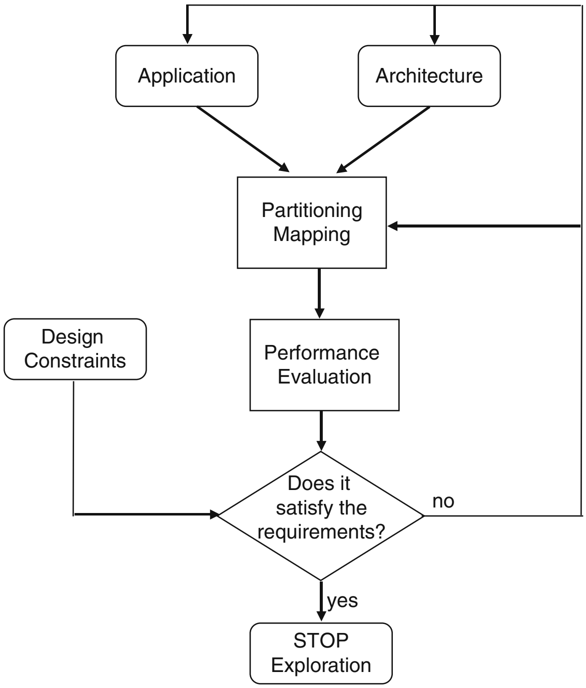
Generally, there are many ways to map an application onto a given hardware architecture. The design space exploration represents the different combinations of mapping parallel software to parallel hardware. In order to be affordable in terms of design cost, the mapping should not require any change of the application code but only of the hardware-dependent code. It also represents the different ways of interaction between the hardware, software, and their configuration and extension.

Usually, the mapping of an application to an MPSoC platform starts from a complex system specification and goes through a vast design space exploration. The application software needs to be adapted to the parallel capabilities of the multiprocessor architectures. Furthermore, to enable fast and flexible exploration of the possible application-to-architecture mappings, it is necessary to automate the hardware–software partitioning of the application [15].

There are two ways for design space exploration: spatial and temporal. The spatial exploration refers to binding application to architecture. It defines the possibilities of mapping tasks on processors, and the communication channels between the tasks to the communication paths available in the MPSoC architecture. The temporal exploration refers to computation and communication ordering. It defines the scheduling policy on each resource and its corresponding parameters, e.g., time division multiple access scheme and the associated slot length, fixed priority scheduling and the associated priorities, or static scheduling and the associated ordering [154].

The goal of design space exploration is to find a best matching between application and architecture based on well-defined criteria or objectives, including the design constraints. The design space exploration is generally represented as an iterative loop with two main phases: performance evaluation and performance optimization in terms of cost and performances (Fig. 3.4).

Fig. 3.4 Design space exploration



The evaluation of the performances for MPSoC may be done using simulation or analysis-based methods [34]. The optimization is necessary when the design constraints and performance requirements are not satisfied. Usually, the optimization consists of application's algorithm, architecture, or partitioning/mapping optimization.

3.1.2 Definition of the System Architecture

The output of the mapping process represents a model at the highest abstraction level, called system architecture (SA) model (Fig. 3.5). The definition of the system architecture given by the Carnegie Mellon University's Software Engineering Institute in its glossary is "representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components" [30].

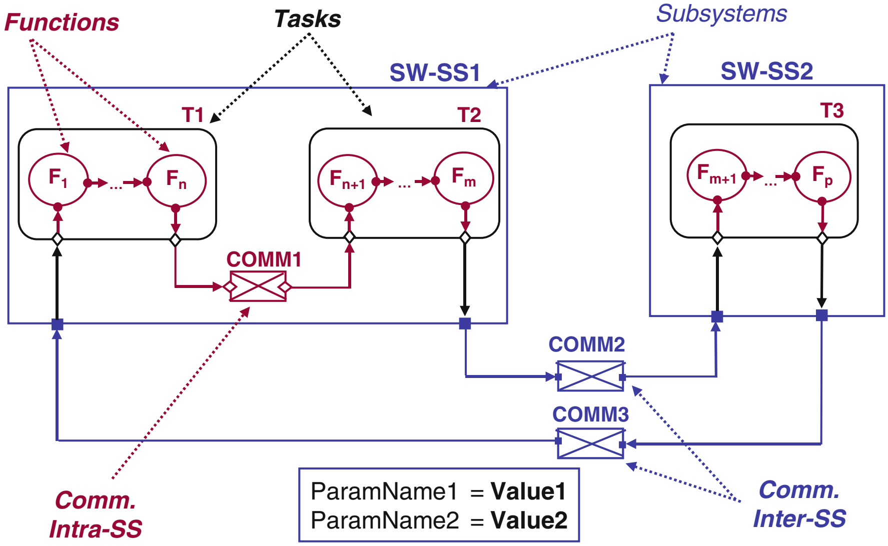


Fig. 3.5 Global view of the system architecture

The system architecture represents a high-level application model combined with partitioning and mapping information. Aspects related to the architecture model (e.g., processing units available in the target hardware platform) are combined into the application model (i.e., multiple tasks executed on the processing units), resulting in a combined architecture/application model. Thus, the system architecture model expresses parallelism in the target application through capturing the mapping of the application functions into tasks and the tasks into subsystems. It makes also explicit the communication units between the tasks to abstract the implementation of the communication protocol used for the data exchange between them.

At the system architecture level, the software is made of a set of functions grouped into tasks. The function is an abstract view of the behavior of an aspect of the application. Several tasks may be mapped on the same software subsystem. The communication between functions, tasks, and subsystems makes use of abstract communication links, e.g., standard Simulink signals or explicit communication units that correspond to specific communication paths of the target platform. The corresponding hardware platform consists of the set of the abstract subsystems.

3.1.3 Global Organization of the System Architecture

The system architecture model is a hierarchical model composed of several component layers. This approach provides insight into how a model is organized and how its parts interact.

The system architecture model may be represented using the hierarchy of concepts depicted in Fig. 3.5. Thus, Fig. 3.5 shows a conceptual representation of the system architecture defining the following concepts: subsystems, tasks, functions, inter-subsystem communication, and intra-subsystem communication.

A subsystem represents a set of tasks that are aimed to be mapped on the same subsystem. Examples of subsystems are SW-SS1 and SW-SS2 in Fig. 3.5. This corresponds to the mapping process of the application tasks on the different computation resources of the target architecture. A task groups a set of application functions. Examples of tasks are T1, T2, and T3. This corresponds to the result of the partitioning process of the application functions into tasks.

The basic element of the system architecture model represents the function. This can be an elementary application function either predefined or user-defined function. Examples of predefined functions are the functions representing mathematical operations (+, –, /, *), constants, conditional structures (if–else), or repetitive structures (do–while). The user-defined functions represent specific functions implemented in diverse programming languages (e.g., C, C++, Matlab). The user-defined functions are also part of the system architecture model.

In order to specify the communication protocol used for the data exchange between the different tasks, communication units are inserted between them in the system architecture model. Later in the software design flow, during the next design steps, the communication units will be replaced by behaviorally equivalent channel implementations with the annotated protocol and device drivers from a real-time operating system targeting to run on the processor.

There are two types of communication units: inter-subsystem and intra-subsystem. The inter-subsystem communication shows the communication between different subsystems, e.g., the communication units COMM2 and COMM3 between the subsystems SW-SS1 and SW-SS2. The intra-subsystem communication specifies the communication between the tasks mapped on the same subsystem, e.g., COMM1 communication unit between tasks T1 and T2 mapped on SW-SS1. The number of the communication units depends on the application partitioning and mapping on the target hardware architecture. The communication between the functions inside the same task is implicit in the system architecture model and it will be translated to communication via local variables inside the task during the next design step.

The system architecture model is annotated with software and hardware architecture parameters to allow the generation and validation of the software stack and of the hardware simulation platforms, and design space exploration.

The system architecture model may be represented using different environments such as Simulink or SystemC. In the following paragraphs, the system architecture will be detailed using as case study the Simulink environment.

Example 2. System architecture model of the token ring application mapped on the 1AX architecture in Simulink Figure 3.6 illustrates a screenshot of the system architecture modeled in Simulink for the token ring application mapped on the 1AX architecture. The top layer of the model's hierarchy represents the two software subsystems available in the 1AX architecture (ARM7-SS and XTENSA-SS) and the

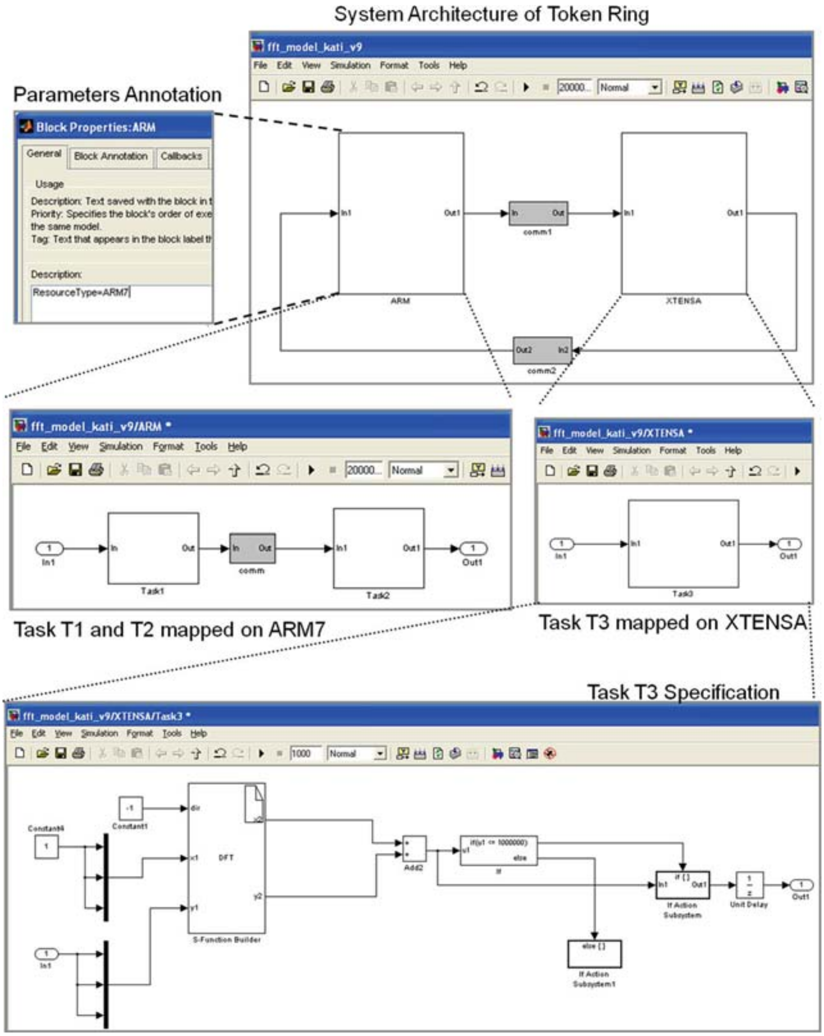


Fig. 3.6 System architecture model of token ring

two inter-subsystem communication units between them (COMM1 and COMM2). The two inter-subsystem communication units allow the data exchange between the two processor subsystems.

The application functions of the token ring application are grouped into three tasks (T1, T2, and T3). Figure 3.6 illustrates that for the token ring application two tasks (T1 and T2) are mapped on the ARM7 processor and the third task (T3) is mapped on the XTENSA processors.

The third task (T3) running on the XTENSA processor computes the DFT function. The DFT function is implemented in an application library developed in

C programming language, representing an example of user-defined function for the token ring application.

The data exchange between the tasks T1 and T2 mapped on the ARM7 processor requires the communication unit COMM.

3.2 Basic Components of the System Architecture Model

The basic components of the system architecture model are the computation and communication components. The computation components consist of the application functions, while the communication makes use of generic I/Os, such as Simulink I/Os or SystemC signals. The detailed description of these components will be illustrated in the following paragraphs using the Simulink environment as representation medium of the system architecture design.

3.2.1 Functions

The application functions can be modeled in Simulink by using Simulink blocks. There are two types of blocks: standard Simulink blocks and user-defined blocks. Blocks are the elements from which Simulink models are built. Every Simulink block has a set of attributes, called parameters, which govern its appearance and its behavior during the simulation. Some types of parameters are common to all blocks (e.g., block name), while other attributes are specific to a particular type of block. Simulink allows users to specify values for many of a block's parameters, thus enabling to customize the behavior to fit the requirements of a particular application. For the standard blocks, Simulink provides predefined continuous and discrete function blocks and a graphical user interface (GUI) to relieve the application model building.

For the user-defined blocks, Simulink provides the capability to integrate in the model user-defined blocks developed in other programming languages such as C, C++, or MATLAB, by using S-functions. To integrate S-functions in Simulink, there are two methods. The first one is to write the S-function block manually. But this method requires also a manual development of a wrapper function, which calls the actual function code. Additionally, the S-function has to be manually compiled using the *mex* utility, in order to generate the MEX file accepted by the Simulink simulation engine.

The second method to use an S-function consists of an automatic generation and compilation of the S-function by using the S-function Builder tool integrated in the Simulink environment. The resulted S-function has a fixed type signature. But the designer has only to set up the configuration panel by specifying the source files of the hand-written function code, the format of the function call, and the input/output arguments passed to the subroutine. The input arguments represent the constant parameters required for the subroutine execution. The output arguments represent the returned value of the subroutine or the non-constant parameters whose values can be changed by the function. Then, based on the configuration, the S-function Builder will automatically create and compile the corresponding S-function.

3.2.2 Communication

The communication between the different application functions is made using Simulink signals. These links or signals connect the different Simulink blocks. They may carry data from one block to one or more other blocks. The data transmission from the source block and its arrival at the destination block happens simultaneously in a rendezvous fashion. The signals may carry different types of data, such as integer, floating point, or boolean. The dimension of the data may vary also from scalar to vector or matrices, but it has to be constant during the execution of the model.

3.3 Modeling System Architecture in Simulink

The system architecture may be described in Simulink or SystemC. This chapter will present the modeling style in the case of using the Simulink environment.

3.3.1 Writing Style, Design Rules, and Constraints in Simulink

The system architecture design in Simulink imposes some limitations and constraints. These design rules include the following:

- Constraints on the selection and configuration of the blocks used for the application modeling
- Constraints on the integration of the application functions implemented in other programming languages such as C/C++
- Constraints regarding the construction of the system architecture model

3.3.1.1 Constraints on the Simulink Standard Blocks

The system architecture model may use only discrete Simulink blocks in order to allow a discrete event simulation.

In case of algebraic loops or feedback path, unit delay blocks have to be inserted in the Simulink loop. These unit delay blocks have to be configured by a sample rate equal to 1 in order to delay their input signals inside the loop. The sample rate represents the number of samples per second. The other blocks are characterized by an inherited sample rate. In case of inherited sample rate blocks, Simulink assigns an inherited sample rate to a block based on the sample rates of the blocks connected to its inputs.

The supported predefined blocks are restricted to a subset of the standard Simulink library. This subset includes the following:

– Mathematical operations, such as sum, multiplication, division, modulo, absolute value
– Logic operations: AND, OR, XOR, unary minus, arithmetic shift
– Discrete blocks: delay, mux, demux, merge, selector, etc.
– Conditional structures: if–then–else and switch-case
– Repetitive structures: for-loop and while-condition-loop
– Sources: constants, extern files, input ports
– Sinks: display block, scope block, extern files, and output ports

3.3.1.2 Constraints on the S-Functions

The system architecture model may include user-defined functions written only in C programming language. The S-functions used to integrate the customized code need to be built by using the S-function Builder tool, which is the fastest and easiest way for the S-function generation compared with the manual implementation.

The arguments of the user-defined function have to respect a well-defined order. Basically, the function call accepts as parameters the input arguments followed by the output arguments, as illustrated in Fig. 3.7. The order of the parameter definition has to be identical with the order of the input/output ports declaration in the configuration panel of the S-function Builder tool. Moreover, the user-defined C function has to return a *void* value.

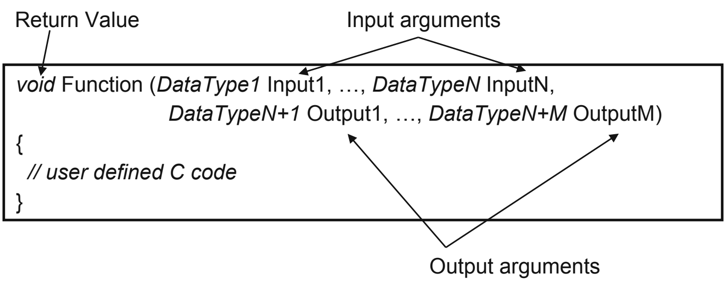


Fig. 3.7 User-defined C-function

Example 3. S-function for the FFT computation in token ring For example, the user-defined C function used for the DFT computation in the token ring application is declared and implemented as shown in Fig. 3.8, where *dir*, *x1*, and *y1* represent the input arguments and *x2* and *y2* represent the output arguments.

The different S-functions are not allowed to share global variables between them, their access being limited to the local data variables. The entire data passing between

```
void DFT(int* dir, double* x1, double* y1, double* x2, double* y2)
{
 int i,k;
 double arg;
 double cosarg,sinarg;
 double x22[3], y22[3];

 for (i=0; i<3; i++)
   {
      x22[i]=0;
      y22[i]=0;
      arg=-(*dir) * 2 *3.141592654 * ((double)i)/3;
      for (k=0; k<3; k++)
         {
           cosarg = cos(k*arg);
           sinarg = sin(k*arg);
           x22[i] = x22[i]+(x1[k]*cosarg - y1[k]*sinarg);
           y22[i] = y22[i]+(x1[k]*sinarg + y1[k]*cosarg);
         }
      *x2+= x22[i];
      *y2+= y22[i]/2;
   }
}
```

Fig. 3.8 DFT function of the token ring

the different S-functions has to be modeled explicitly, with a dedicated Simulink signal used for the connection between the different S-functions.

3.3.1.3 Constraints on the Communication

To transfer data in Simulink, the different blocks are connected by signals. The signal between the blocks may carry data from one source block to multiple destination blocks. This specific feature determines to use communication units which correspond to point–point communication schemes and restricts the global shared memory accesses in the target architecture.

3.3.2 Software at System Architecture Level

The software at the system architecture level consists of a set of application functions grouped into tasks. Figure 3.9 shows three examples of application functions grouped into tasks.

Example 4. Software in the token ring system architecture model The software in the token ring system architecture model is represented by the application functions, e.g., DFT, +, –, /, $*$, if, else, or mathematical constants.

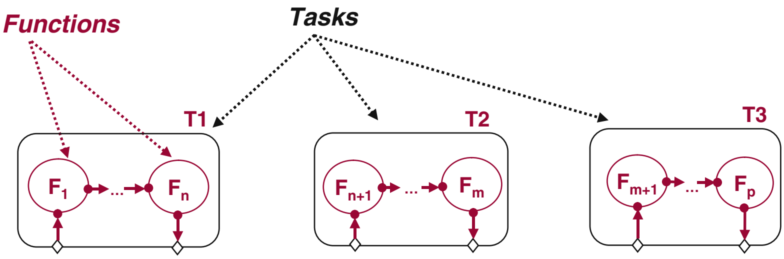


Fig. 3.9 Application functions grouped into tasks

3.3.3 Hardware at System Architecture Level

The hardware at the system architecture level consists of a set of abstract hardware and software subsystems that encapsulate the tasks aimed to be executed on those subsystems, and the different communication units introduced between the subsystems to specify the communication protocol.

Example 5. Hardware in the token ring system architecture model The hardware in the token ring system architecture model is represented by the processor subsystems XTENSA-SS and ARM7-SS, and the inter-subsystem communication units COMM1 and COMM2 that connect the two processor subsystems, as it is illustrated in Fig. 3.10.

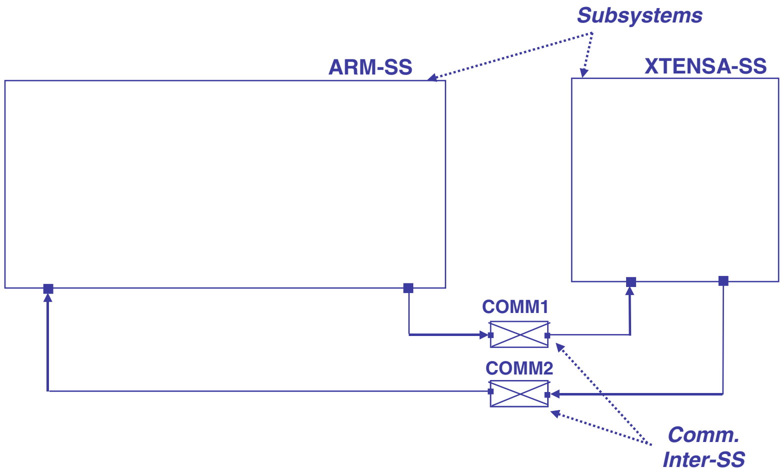


Fig. 3.10 Software subsystems for the token ring application

3.3.4 Hardware–Software Interface at System Architecture Level

The hardware–software interface at the system architecture level consists of a set of links which connect the input/output ports of the different subsystems with the input/output ports of the tasks that are mapped on those subsystems.

Example 6. Hardware–software interface in the token ring system architecture model An example of hardware–software interface in the token ring system architecture model represents the Simulink link that connects the output port of task T3 (out) with the output port of the XTENSA subsystem (out) in Fig. 3.10.

3.4 Execution Model of the System Architecture

The Simulink model is used as a reference model for debugging the application's algorithm. The following sections will describe the adopted configuration in the Simulink simulation engine to validate the system architecture model.

The simulation of the system architecture model uses a variable step discrete solver. This allows validation of the functionality of the application. For the inter-subsystem and intra-subsystem communication units, Simulink uses an abstract simulation model for each of these units based on generic Simulink I/Os.

The system architecture model is used to validate the application's algorithm through functional simulation. It is similar to native code execution on the host machine. The performance estimation at this level uses a simulation-based approach. As the system architecture model represents a high-level application model, the hardware architecture is completely abstracted, including processor subsystems or communication infrastructure. The memory usage is also abstracted; the application uses variables and pointers without taking into consideration details related to aspects such as shared memory or virtual memory. As presented in [12], a relevant metric at this high abstraction level is the simulation time of the application.

The simulation time may give useful information on the efficiency of the application's algorithm in terms of behavioral features. The application's algorithm does not depend on the final operating system that will be running on the target processors but influences the performance after the application's parallelization.

Example 7. Simulation of the token ring system architecture model The simulation of the system architecture model in Simulink allows to verify the functionality of the application, including the DFT computation. The simulation requires 3 s and it stopped when the resulted value after the different computations had become bigger than 1,000,000, conform the initial specification of the application.

3.5 Design Space Exploration of System Architecture

3.5.1 Goal of Performance Evaluation

The MPSoC design process relies on several decisions and constraints related to hardware and software architecture, which can influence the overall performance of the system. Examples of hardware architecture decisions are as follows: number and type of processors, memory size, type of memories (local, global), type of communication network (point to point, bus, network on chip), communication latency, etc. Examples of software architecture decisions are: type of scheduling algorithm used by the operating system for the tasks activation/deactivation, type of communication primitives (blocking or non-blocking semantic), real-time execution requirements, binary code size, synchronization mechanisms between the tasks running on the same processor, etc.

These different decisions influence the overall execution time of the system, cost, and power consumption. Therefore, good decisions are required to be able to control the MPSoC design process.

The goal of performance evaluation at the system architecture level is to allow in an early phase of the design process profiling the communication and computation. This can be accomplished by providing information independently from the system behavior in time.

3.5.2 Architecture/Application Parameters

The system architecture model is annotated with application and architecture parameters that can influence the global performance of the final system. These parameters can be classified into two main categories: specific to subsystems and specific to communication units.

(a) Architecture/Application Parameters specific to Subsystems: The parameters specific to subsystems characterize the different subsystems (hardware and software) from both hardware and software points of view.

Examples of hardware architecture parameters that annotate the subsystems are as follows:

– *ResourceType* which specifies the type of the hardware resource. There are three types of hardware resources: computation resource (processors), storage resource (memory), and I/O resources (I/O peripherals). In the case of a subsystem that represents a processor subsystem, the *ResourceType* specifies the type of the processor cores.
– *NetworkType*. This parameter specifies the type of the network component used to interconnect the different subsystems in the target architecture. Examples of interconnect components are bus and network on chip (NoC).
– *NoCTopology*. This parameter is used when the *NetworkType* is NoC and it specifies the topology of the NoC. Examples of NoC topologies are mesh, torus, tree, or butterfly.
– *NoCRoutingAlgorithm*. This parameter is used when the different subsystems are interconnected by a NoC. The parameter specifies the routing algorithm used by the routers to transmit the received data packet. Example of a routing algorithm is XY or YX.

– *NoCArbitrationAlgorithm*. This parameter is used to specify the type of the arbitration algorithm used inside a NoC router (e.g., round robin, priority based), in order to select the routing request to be treated, when the router receives more than one request simultaneously for packets transmission.
– *ResourceName* which identifies the hardware resource.

Each subsystem which represents a processor subsystem is annotated with software architecture parameters. Examples of software architecture parameters are the following:

– *OSType*, which specifies the name of the operating system running on the target processor. Examples of operating systems are Linux, Mutek, DwarfOS, and eCos.
– *SchedulerType* to identify the type of the OS scheduler (preemptive, cooperative), when the target OS supports different schedulers.
– *SchedulerAlgorithm* to define the algorithm used for the tasks management by the operating system (round robin, priority based), etc.

Example 8. Parameters specific to the subsystems in the token ring system architecture model Examples of architecture/application parameters annotating the subsystems of the token ring system architecture model are *ResourceType* with values “ARM7” for the ARM7 subsystem and “XTENSA” for the XTENSA subsystem, *NetworkType* with the value “AMBA” because in the 1AX architecture different subsystems are interconnected through an AMBA bus, *OSType* with value “DwarfOS” to specify that the target operating system running on each software system is the DwarfOS for both processors.

(b) Architecture/ Application Parameters Specific to Communication Units: The parameters specific to the communication units can be architecture or application parameters.

Examples of hardware architecture parameters that annotate the communication units are as follows:

– *ResourceType* which specifies the type of the communication protocol or storage resource. In the case of an inter-subsystem communication unit, the *ResourceType* specifies the storage resource on which the communication buffer will be mapped. The communication buffer can be mapped in the sender subsystem, receiver subsystem, or in a stand-alone storage resource, such as global memory or hardware FIFO. In the case of an intra-subsystem communication unit, the *ResourceType* specifies the communication protocol implemented in software, such as software FIFO protocol or shared memory.
– *AccessType*. This parameter identifies whether the access to the memory that stores the communication buffer is performed directly by the processor or by using a DMA mechanism, in case that the target hardware architecture provides such kind of mechanism for the memory access.

– *ResourceName* which identifies the storage resource in case that the target hardware architecture provides several storage resources of the same type, e.g., several hardware FIFOs or several global memories.

The communication unit can also be annotated with software architecture parameters, e.g., *CommType,* which identifies the type of the communication library used during the HdS integration. This parameter specifies the communication APIs used in the tasks code after their generation for the data exchange, e.g., *send(...)/recv(...)* APIs when the *CommType* is "MPI"; or *DOL_read(...)/DOL_write(...)* communication APIs when the *CommType* parameter has the value equal to "DOL." The different communication units can be accessed using different communication primitives in the tasks code.

Example 9. Parameters specific to the communication units in the token ring system architecture model Examples of architecture/application parameters annotating the communication units of the token ring system architecture model are *ResourceType* with value equal with "GMEM" for the communication units COMM1 and COMM2 in order to specify that the corresponding communication buffers used for the data exchange between the two processors of the 1AX architecture are mapped onto the global memory. The parameter *ResourceType* has the value "SWFIFO" for the communication unit COMM in order to specify that the communication between the tasks T1 and T2 mapped on the same ARM7 processor follows a software FIFO protocol.

Another parameter annotating the communication units COMM1 and COMM2 represents *ResourceName* with values "MEM0" in order to specify that the communication buffers corresponding to these communication units are mapped on the global memory identified through id *MEM0*. The *CommType* parameter has the value "MPI" for all the communication units, in order to specify that all the tasks use the communication primitives "*send(...)/recv(...)*" for all the data exchanges.

Figure 3.11 illustrates a capture of the communication units' annotation with the architecture parameters for the COMM1 communication unit of the token ring system architecture. The annotation is performed in Simulink by adding parameters in the description field of the block properties of the Simulink block representing the communication unit.

3.5.3 Performance Measurements

At the system architecture level, the performance measurement consists of profiling the communication and computation for each task and/or each processor. As the system architecture has no time notion, the result of profiling is a time-independent data. Examples of metrics that can be measured at this level are as follows: application data size, buffer size required for the intra-subsystem and inter-subsystem communication, the total quantity of exchanged data between the tasks during the

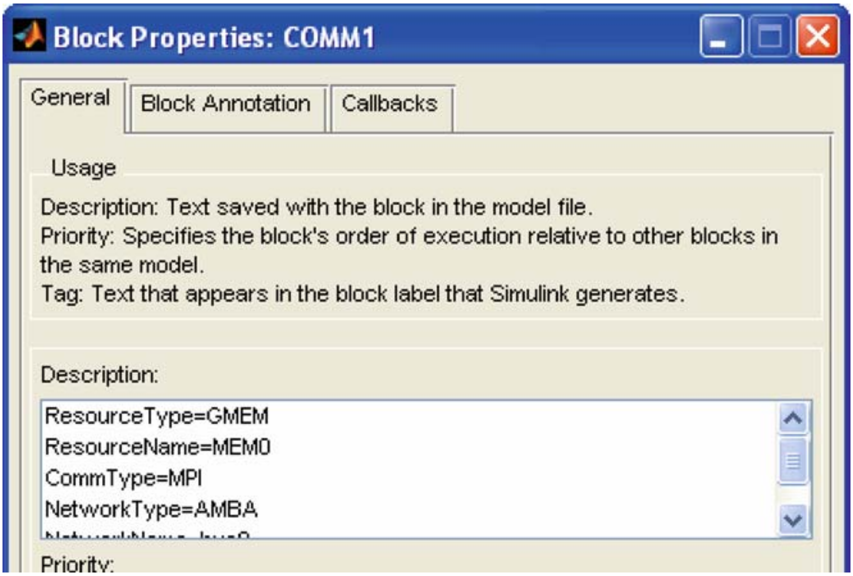


Fig. 3.11 Architecture parameters specific to the communication units

execution, the number of iterations of a function execution, the amount of data transferred between the different processors, etc.

Example 10. Performance measurements in the token ring system architecture model For example, the DFT function of the token ring application was required to be called and computed totally 14 times during the execution of the whole application. The application data exchanged between the ARM and XTENSA processors was 112 bytes during the entire execution. The size of the application data sent by the first task mapped on the ARM processor to the second task running on the ARM processor was 64 bytes.

3.5.4 Design Space Exploration

At the system architecture level, the designer can experiment different partitioning and mapping schemes. The designer can regroup the functions into the tasks in several ways and map these tasks on different subsystems. This exploration influences the total amount of data exchanged between the tasks during the execution, the application data size, or the number of iterations of a specific function. By changing the partitioning and mapping of the application on the target architecture, the number and type (intra-subsystem, inter-subsystem) of communication units may also vary. The designer may adopt different communication protocols and may map the communication buffers onto different storage resources by annotating the communication units with the proper architecture parameters.

Example 11. Design space exploration for the token ring application In the case of the token ring application, the designer may map the buffers required for the inter-subsystem communication onto different communication architecture resources, such as the local memories of both ARM and XTENSA processors or the shared global memory. The designer may also opt for the dedicated hardware FIFO component directly connected to the local buses of both processor subsystems. Regarding the partitioning and mapping, the token ring functions can be grouped forming tasks in different ways, resulting in tasks with different levels of granularity. These tasks may be mapped on the processors in several ways. For example, the DFT computation may be mapped on the ARM processor instead of the XTENSA processor, letting the XTENSA processor to be responsible for the control part of the token ring application. In this case, the number of the intra-subsystem communication units becomes 1 for the XTENSA processor, while the ARM7 subsystem has no intra-subsystem communication unit.

3.6 Application Examples at the System Architecture Level

In the following paragraphs, two examples will be presented at the system architecture level: the Motion JPEG application mapped on the Diopsis RDT architecture and the H.264 encoder application mapped on the Diopsis R2DT architecture.

3.6.1 Motion JPEG Application on Diopsis RDT

This section presents the system architecture design in case of the Motion JPEG (MJPEG) decoder application running on the Diopsis RDT architecture with AMBA bus. This consists of mapping the Motion JPEG decoder application onto the Diopsis RDT platform, and modeling the resulted system architecture.

The first step of the system architecture design represents the functional modeling of the application in Simulink. The development of the MJPEG decoder application in Simulink requires seven S-functions in order to integrate the C code of the main parts of the decoding algorithm.

After the functional modeling, the main functions of the application are isolated into separate tasks. Figure 3.12 shows the application partitioning into tasks. The variable length decoding (VLD) constitutes the first task. The differential pulse code demodulation on the DC component (DPCD), run length decoding on the AC component (RLD), zigzag scan, and inverse quantization (IQ) are grouped into a second task. The inverse discrete cosine transformation (IDCT) makes up the third task and, finally, the display function of the decoded image composes the fourth task.

After the partitioning of the application functions into tasks, the next step represents the mapping of these tasks onto the computation and I/O subsystems provided

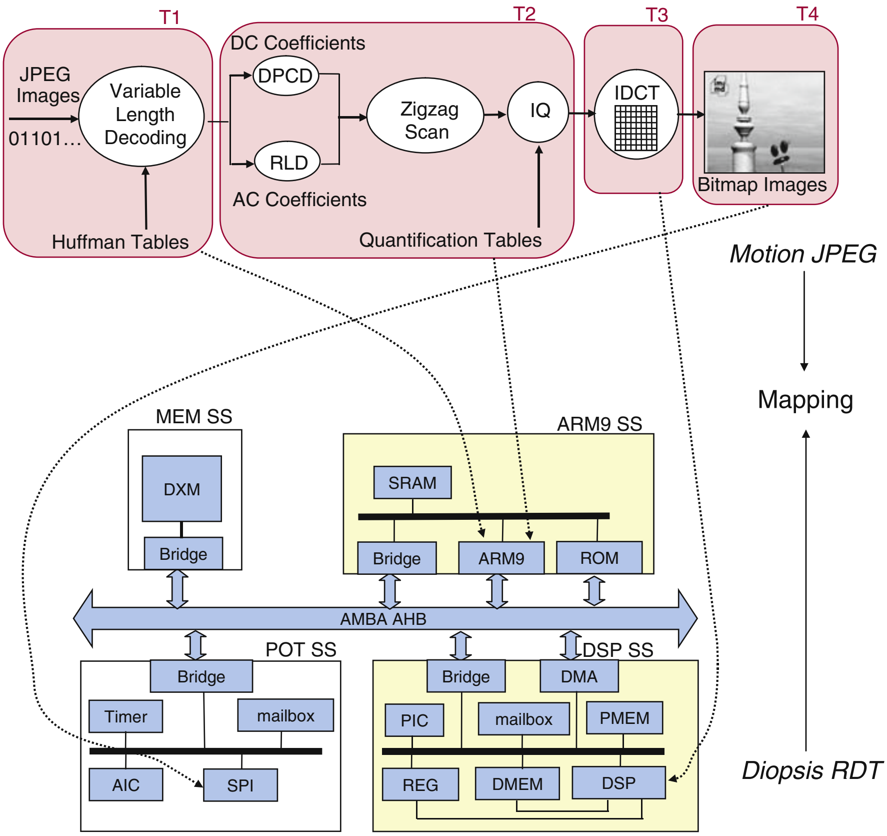


Fig. 3.12 Mapping motion JPEG on Diopsis RDT

by the target architecture. Therefore, the four different tasks are mapped onto the Diopsis RDT architecture (Fig. 3.12). Thus, the first two tasks (*T1* and *T2*) are mapped on the ARM9 processor. The third task (*T3*), performing the IDCT computation, takes 68% of the total execution time of the decoding process, being the most computation-intensive task. Therefore, it was mapped on the DSP processor. The resulting decoded image of the task *T4* is displayed on an LCD panel connected through the SPI peripheral of the POT. Hence, *T4* was mapped on the POT subsystem.

After the mapping process, in order to specify the communication protocol, several communication units are inserted between the tasks mapped on the same processor and between the different subsystems. The communication buffers used for the communication between the tasks can be mapped on the local memories of both processor subsystems or on the global memory. Besides the buffer mapping on the storage resources, the mapping has to specify the end-to-end communication path between the two processors. The result of the mapping represents the system architecture model for the Motion JPEG application mapped on the Diopsis

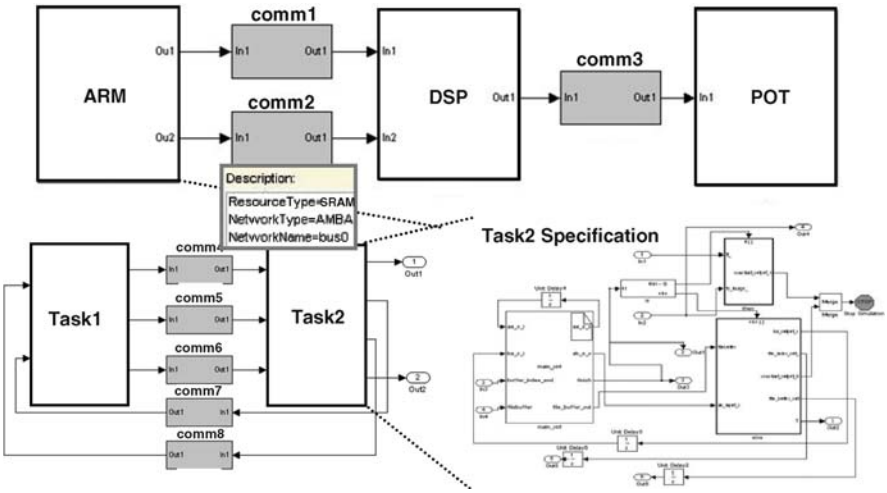


Fig. 3.13 System architecture example: MJPEG mapped on Diopsis

RDT architecture. A screenshot of the system architecture modeled in Simulink is illustrated in Fig. 3.13.

This model includes three subsystems: two software subsystems (the ARM and the DSP subsystem) and a hardware subsystem (the POT). The Simulink hierarchy is able to capture the mapping of the application onto the architecture at a high abstraction level through the decomposition of the system into tasks and subsystems. The Simulink model includes also explicit communication units to capture different communication protocols and resources provided by the architecture.

In this case, the intra-subsystem communication for the tasks mapped on the same subsystem follows a software FIFO protocol (SWFIFO). The inter-subsystem communication between the different subsystems uses data buffers mapped on different storage resources, such as DSP data memory (DMEM), DSP registers (REG), ARM local memory (SRAM), or the distributed external memory (DXM). Later in the design flow, each of the communication units can be mapped on a specific communication path and protocol of the final architecture. The number of communication units depends on the application partitioning and mapping decisions on the target architecture.

The system architecture in Simulink is annotated with architecture information used for the further software refinement and generation of the hardware simulation platform. The hardware architecture parameters used in this example are as follows:

(a) Parameters specific to subsystems:

– *ResourceType* with possible values “ARM9,” “DSP,” and “POT” for a subsystem.
– *NetworkType* parameter, which has the value equal with “AMBA” for the Diopsis architecture with AMBA bus.

– *ResourceName* parameter. This parameter identifies the hardware resource of the target architecture, e.g., the DSP.

On the software side, each processor will execute a tiny operating system, namely, *Dwarfos*. Thus the *OSType* parameter for each software subsystem of the system architecture model has the value "Dwarfos." As the *Dwarfos* operating system supports only round-robin preemptive scheduling, the software architecture parameters *SchedulerType* and *SchedulerAlgorithm* are not required in this example.

(b) Parameters specific to communication units:

– *ResourceType* with possible values or "DXM," "SRAM," "DMEM," "REG" for the inter-subsystem communication and "SWFIFO" in case of an intra-subsystem communication unit.
– *AccessType* parameter, required to specify for an inter-subsystem communication unit whether the DSP will access the local memory of the ARM or the external global memory directly or by initiating a DMA transfer.
– *ResourceName* parameter. This parameter identifies the hardware resource of the target architecture in case of an inter-subsystem communication unit, e.g., the external memory DXM.

The tasks executed by the processors will use the *send(...)/recv(...)* primitives as communication APIs. Therefore, the *CommType* has the value "MPI" to represent the message passing *send(...)/recv(...)* semantics for each communication unit.

To validate the MJPEG algorithm, the system architecture model can be simulated using the Simulink discrete-time simulation engine. The input test image represents a 10-frame bitstream encoded using QVGA YUV 444 format. The simulation time is 15 s on a PC running at 1.73 GHz with 1 GB RAM.

The simulation allowed measuring some performance indicators. Hence, the total number of iterations necessary to decode the 10-frame input image was 36,000. The communication between the ARM and DSP through the communication unit COMM1 requires a buffer of 1 word (4 bytes), the communication unit COMM2 requires 64 words (256 bytes) buffer size, and, finally, the communication unit COMM3 requires 4 words (64 bytes). The total number of words exchanged between the different subsystems during the decoding process of the 10 frames was 2,484 Kwords.

3.6.2 H.264 Application on Diopsis R2DT

The H.264 encoder is a computation-intensive video application and more complex than the token ring or Motion JPEG applications. Hence, the Diopsis R2DT with NoC is used to execute the H.264 encoder.

This section presents the system architecture design in the case of the H.264 main profile encoder application running on the Diopsis platform with two DSP and one ARM9 processors interconnected through a NoC.

The first step represents the functional modeling of the H.264 application in Simulink. The H.264 encoder algorithm is comprised of several functions and two dataflows: a forward path and a reconstruction path. The input frame of a video image sequence (Fn) is processed in units of a macroblock. The reference code used for the H.264 encoder development is the x264 open-source code [172]. The sequential C code is converted to a dataflow model as explained in [71]. The development of the H.264 encoder application in Simulink requires four S-functions in order to integrate the C code of the main parts of the encoding algorithm.

After the functional modeling, the different functions are grouped into tasks. Figure 3.14 illustrates the partitioning of the H.264 functions into tasks. The application functions are grouped into three tasks as follows: the CABAC entropy encoder

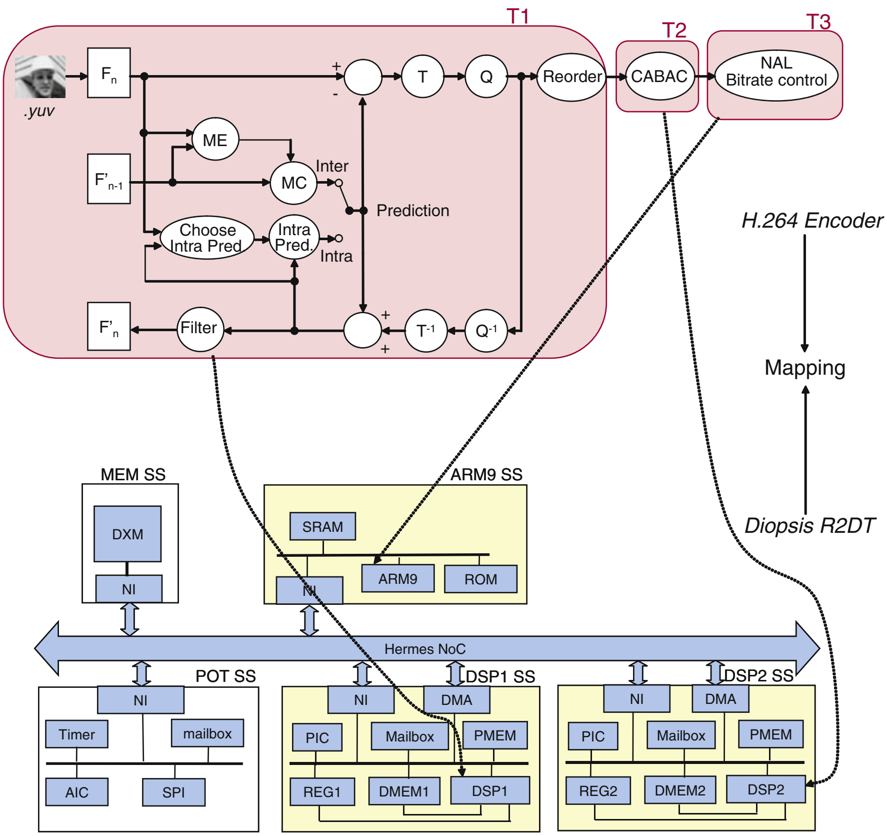


Fig. 3.14 Mapping H.264 on Diopsis R2DT

constitutes the task *T2*; the NAL construction and bitrate controller are grouped into task *T3*; and the other computation and control functions are grouped into task *T1*.

After the partitioning, the tasks are mapped on the available resources of the Diopsis R2DT architecture. Therefore, each task is mapped on a different CPU. As tasks T1 and T2 require a big amount of computation, they are mapped on the DSP processors: task T1 on DSP1, respectively, task T2 on DSP2. The task T3 including the control part for the bitrate is mapped on the ARM9 processor. The communication between these tasks mapped on the three different processors represents inter-subsystem communication and it implies a total of three communication units, one between each pair of processors (DSP1->DSP2, DSP2->ARM9, and ARM9->DSP1).

The resulted system architecture model is illustrated in Fig. 3.15.

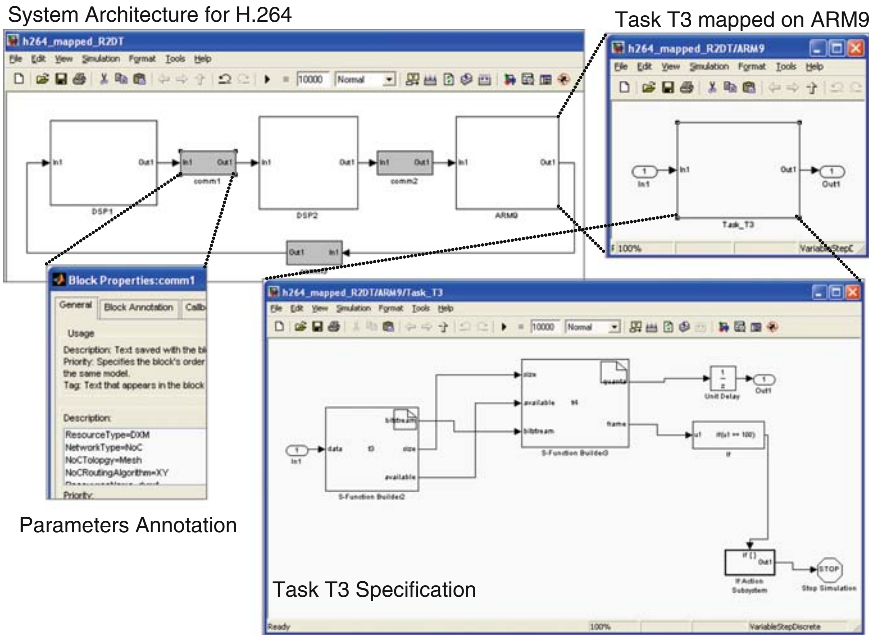


Fig. 3.15 H.264 encoder system architecture model in Simulink

Similar to the system architecture model of the MJPEG application, in order to allow the generation of the next levels, the high-level application model of the H.264 contains the following architecture information annotated as parameters:

(a) Parameters specific to subsystems:

– *ResourceType* for the subsystem in order to differentiate between the “ARM9” and “DSP” subsystems.

– *NetworkType* parameter, which has the value equal with "NoC," as the target architecture will contain a NoC in the Diopsis architecture. This NoC implementation is based on the Hermes NoC [107].
– *NoCTopology*, with possible values "MESH" and "TORUS," as the target architecture support both types of NoC topologies.
– *NoCRoutingAlgorithm* with possible value "XY" for the mesh topology or "NMWF" (non-minimal west-first) algorithm for the torus topology [60].
– *NoCArbitrationAlgorithm* with value "ROUND_ROBIN" to arbitrate the simultaneous packet transmission requests on the router.
– *ResourceName* parameter to identify the hardware resource, e.g., the DSP1 or DSP2 processor as the architecture contains more than one same type of DSP (Atmel magicV VLIW processors).

As in this case the processors will execute single tasks, the software architecture parameter is represented only by the *OSType* parameter with the value "DWARFOS" to specify the OS that will be responsible for the application boot, interrupt management, and the access to the hardware.

(b) Parameters specific to communication units

– *ResourceType* to specify the communication protocol of the inter-subsystem communication unit ("DXM," "SRAM," "DMEM," "REG").
– *AccessType* parameter, required to specify whether the DSP processors will access the local memory of the ARM or the external global memory directly or by initiating a DMA transfer.
– *ResourceName* parameter to identify the hardware resource, e.g., the memory.

As software architecture parameter, the communication units are annotated with the *CommType* parameter with value "MPI" for the communication primitives.

To validate the H.264 encoder algorithm, the system architecture model can be simulated using the Simulink discrete-time simulation engine. The input test video represents a 10-frame video sequence QCIF YUV 420 format. The simulation time is 30 s on a PC running at 1.73 GHz with 1 GB RAM.

The simulation allowed measuring some performance indicators. Thus, the total number of iterations necessary to decode the 10-frame video sequence was equal to the number of frames. This is due to the fact that all the S-functions implemented in Simulink operate at frame level. The communication between the DSP1 and DSP2 processors uses a communication unit that requires a buffer of 288,585 words to transmit the encoded frame from the DSP1 processor to the DSP2 in order to be compressed. The DSP2 processor and the ARM9 processor communicate through a communication unit that requires a buffer of 19,998 words. The last communication unit between the ARM9 and DSP1 processors requires one word buffer size in order to store the quanta value required for the encoder. The total number of words exchanged between the different subsystems during the encoding process of the 10-frame video sequence using main profile configuration of the encoder algorithm was approximately 3,085 Kwords.

3.7 State of the Art and Research Perspectives

3.7.1 State of the Art

Current literature includes several academic and industrial design environments that involve specification of the application mapping on the target architecture at the system architecture level.

The automatic parallelization of a sequential program code is an open research topic. Several research works have already focused on automatic partitioning and tasks mapping, such as [173] and [99]. In [48], the authors propose an automatic partitioning of the application and automatic allocation of the computation resources using genetic algorithms. Reference [123] proposes a programming paradigm that facilitates the translation of sequentially code software algorithms of the multimedia applications into their parallel implementations.

Other research category focuses on finding the best mapping of an application onto the architecture by using different kinds of optimization algorithms and metrics. Examples of these kinds of research works and tools are PISA [20] which defines the mapping process as a multi-objective search problem, Compaan compiler [99] which automatically generates the mapping specification of an application modeled as Kahn process networks onto the Intel IXP network processor, Mat [97] which describes the APOTRES framework for mapping DSP applications on single program multiple data (SPMD) architectures, Bus [27] which presents a framework that automatically partitions a C application code into hardware and software, or Xue [173] which treats the memory and processor allocation problem applying a run-time resource partitioner for multiple applications running on a shared memory MPSoC architecture. The Sesame environment described in [47, 155] defines the optimal mapping problem taking into account three objectives: maximum processing time in the system, total power consumption, and the cost of the architecture. The Sesame environment uses analytical methods to identify a small set of promising mapping candidates, and then uses simulation for performance evaluation. The DOL (distributed operation layer) framework allows multi-objective algorithm mapping onto MPSoC architectures with system-level performance analysis [154].

Besides the partitioning and mapping, other research works are related to the specification, modeling, and simulation of the system architecture. In the DOL environment, the application is specified as Kahn process network [154]. The application, platform, and mapping information are stored into three separate XML files. Many research efforts focus on the standardization of the XML format to facilitate various IPs exchange among different tools and IP providers. The IP XACT proposed by the Spirit consortium is an example of standardization proposal in the form of an XML schema and APIs for representing and manipulating hardware IPs [146].

An example of modeling environment is the well-known Ptolemy [26] for high-level system specification that supports description and simulation of multiple models of computation, e.g., synchronous dataflow (SDF) , which is a special case of dataflow which detects deadlock and boundedness, boolean dataflow (BDF) which is a generalization of SDF that sometimes yields to deadlock or boundedness analysis, dynamic dataflow (DDF) , which uses only run-time analysis to detect deadlock and boundedness, finite state machine (FSM) , which is comprised of a finite number of states, transitions between those states and actions. The Ptolemy environment allows simulation at algorithmic level.

PeaCE [64] is a Ptolemy-based co-design environment that supports hardware and software generation from a mixed dataflow and extended FSM specification. PeaCE also attempts to generate SoC architecture from an algorithm-level model. An extended version, the HOPES framework, is a new model-based programming [13, 58, 105] environment of embedded software, which supports several environments for the initial specification (PeaCE, UML, KPN) [65].

Several other research groups investigate the specification and simulation of multimedia applications using Simulink and PeaCE [83]. In [129] a design flow for data-dominated embedded systems is proposed, which uses Simulink environment for functional specification, and analysis of timing and power dissipation. This approach mainly focuses on an IP-based design with single processor.

Recently, UML is investigated as a system-level language [36]. Reference [73] proposes a UML-based MPSoC design flow that provides an automated path from UML design entry to FPGA prototyping, including the functional verification and the automated architecture exploration.

3.7.2 Research Perspectives

Despite the existing of a huge literature on system architecture design, this is still an open issue for heterogeneous MPSoC. There are three key problems concerning the system architecture design:

– Automatic partitioning of the application functions into tasks
– Automatic mapping of the tasks onto the target architecture
– Automatic mapping of the communication onto the target architecture. This includes communication buffer mapping on the storage resources and specification of the communication path

Programming the complex MPSoC architectures and providing suitable software support (compiler and operating system) seems to be a key issue. This is due to the fact that either system designers or compilers will have to make the application code explicitly parallel to run on these architectures.

A first difficulty found in MPSoC design is how the applications running on these multi-processor architectures are decomposed into several processes/tasks and how these parallel tasks can share the same resources provided by the architectures. In particular, allocation of the computation resources (processing units) and storage resources (memories) is critical, as it dictates both performance and power consumption.

Automatic generation of the system architecture model represents one research perspective. Starting from the specification of the application, specification of the architecture and design constraints, the automatic partitioning, and mapping could find the best configuration. The specification of the application can be considered as being the functional model in Simulink, composed of several functions, similar to dataflow models.

The architecture specification has to include information related to the hardware resources of the target architecture, such as number and types of the available processors, size of the local and external memories, and possible communication paths/protocols between the different processors. The communication paths can be captured using the notion of graph, where the nodes are the hardware resources of the architectures that may be crossed during a data exchange between the processors. Examples of nodes are the CPUs, co-processors, DMA engines, memories, local buses, or the global interconnect components (AMBA, NoC). The edges of the graph represent the connections between them. In this way, the specification of the communication path in the system architecture model (e.g., *AccessType*) is reduced to solve the shortest path problem between two nodes of a graph. The architecture specification can be stored using an XML format [154].

The design constraints may specify limitations for the relation between the application specification in Simulink and the resource components. They include the following:

- The constraints between the functions of the application and the processors of the hardware architecture. For example, certain functions have to be grouped on the same task or several tasks must run on certain processors or certain processors types. One may also specify that certain tasks must be executed on the same processor.
- Application constraints. This concerns real-time execution requirements, e.g., deadline meeting constraints, execution time, or communication latency.

Another method for the system architecture automatic generation may be based on profiling tools. The application specification can be profiled using specific code profiling tools. The profiling tools record summary information of the execution (e.g., number of function calls) and help to expose the performance bottleneck and hotspots. Based on the profiling data and constraints information, analytical methods can propose an efficient application and communication mapping.

3.8 Conclusions

This chapter presented the system architecture design with case studies in Simulink for the token ring application mapped on the 1AX architecture, the Motion JPEG decoder application mapped on the Diopsis RDT MPSoC architecture, and the H.264 encoder application mapped on the Diopsis R2DT architecture.

The hierarchical organization of the system architecture model allowed combining the application model with the specification of the partitioning and mapping of the computation and communication onto the hardware architecture resources.

The simulation at the system architecture level allowed to validate the functionality of the application and to profile the communication requirements of the applications (number of bytes that need to be exchanged during the execution).

Chapter 4
Virtual Architecture Design

Abstract This chapter details the virtual architecture design. The virtual architecture design consists of transforming the application functions into the final application tasks C code and mapping the communication onto the hardware resources available in the target architecture. The key contribution in this chapter represents the virtual architecture definition, organization, and design, using SystemC, for the token ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture, and the H.264 encoder running on the Diopsis R2DT architecture. The simulation of the virtual architecture models allows validating the partitioning and the final application tasks code. Different communication mapping schemes are explored in order to analyze their impact on the global performances.

4.1 Introduction

The virtual architecture design consists of mapping the communication onto the hardware platform resources and generating the final C code for each task. At this phase, the different links used for the communication between the different tasks of the system architecture model are mapped onto the hardware communication resources available in the architecture to implement the specified protocol. The system architecture tasks made of application functions are transformed into the final application tasks code. These tasks codes designed in C are adapted to the communication mechanism through the use of adequate HdS communication primitives. The result of the virtual architecture design represents the virtual architecture model.

4.1.1 Definition of the Virtual Architecture

The second hardware–software abstraction level is called virtual architecture level (VA). The virtual architecture captures the global organization of the system into

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_4,

abstract software and hardware modules or subsystems and abstract hardware/software interfaces. The virtual architecture model may be manually coded or automatically generated by system architecture parser and analysis tools.

The objectives of the virtual architecture design are as follows:

- Validation of the application partitioning and the tasks mapping on the processing subsystems available in the target architecture
- Verification of the final tasks code of the software stack
- Early estimation of the communication requirements

The virtual architecture is comprised of abstract subsystems that are interconnected using abstract communication channels or abstract network components. The abstract hardware or software processing subsystem represents a component which implements the software tasks, respectively, the hardware functions. The abstract communication network represents high-level communication channels, such as message-passing channels, abstract buses, or NoC.

Figure 4.1 illustrates the global view of the virtual architecture, composed of two abstract software subsystems, a memory component and a communication network. The left part of the figure corresponds to the hardware architecture, while the right part represents the software code at the virtual architecture level.

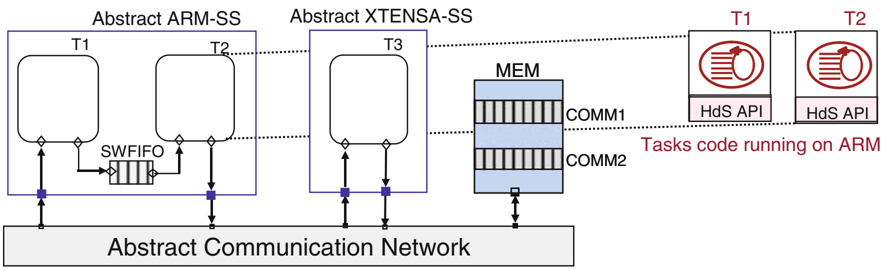


Fig. 4.1 Global view of the virtual architecture

4.1.2 Global Organization of the Virtual Architecture

The virtual architecture model is a hierarchical model. The virtual architecture is composed of abstract subsystems that are interconnected using an abstract communication network, such as abstract bus, abstract NoC, or abstract point-to-point communication channels.

The abstract subsystems may represent a processor subsystem, a hardware subsystem, or a memory subsystem. The software processing subsystem represents a component or module which includes a set of task modules that are aimed to be executed on that processing subsystem, and a set of abstract communication channels for the communication between the task modules inside the same subsystem. The hardware processing subsystem contains a single task module which implements the hardware functions.

The task modules abstract the hardware/software interface. Each task module can be characterized by two elements: container and ports. The container represents the task code. The task code is represented by a sequential C code which implements the application functions that were grouped together to form the task. The task code also contains communication primitives (HdS API) which allows accessing the ports of the task modules. The ports of the task module represent logic ports of the task, which serve to allow the software code to access the communication channels used for the data exchange with another tasks. The logic ports of the task modules are connected to the ports of the subsystem that encapsulates them, or to the intra-subsystem communication channels.

At the virtual architecture level, the intra-subsystem communication units become abstract communication channels inside the processor subsystem. The inter-subsystem communication units become abstract communication architecture. They also determine the memory modules that serve as storage resources for the communication buffer mapping and the type of global interconnect component. The type of the communication protocol and the topology of the network infrastructure are implemented according to the annotation of the system architecture model.

Example 12. Virtual architecture of the token ring application Figure 4.1 shows a conceptual representation of the virtual architecture for the token ring application mapped on the 1AX architecture.

The virtual architecture contains two abstract subsystems (ARM-SS and XTENSA-SS), corresponding to the ARM, respectively, XTENSA processors and the global memory module (MEM). All the subsystems are interconnected by an abstract AMBA bus. The different software subsystems encapsulate the application tasks and the communication channels for the data exchange between the tasks mapped on the same processor. For instance, the ARM-SS subsystem includes the two task modules (T1 and T2) that were mapped on this processor, and a SWFIFO communication channel used for the communication between T1 and T2. SWFIFO represents an abstract communication channel which implements a FIFO communication protocol. The XTENSA-SS subsystem includes the T3 task module.

The inter-subsystem communication units COMM1 and COMM2 are mapped on the global memory (MEM).

The virtual architecture model may be represented using different design languages, such as SystemC [62] or SpecC [54]. In the following paragraphs, the virtual architecture will be detailed using the SystemC design language.

4.2 Basic Components of the Virtual Architecture Model

The basic components of the virtual architecture model are the software and the hardware components. The software components allow for the description of the pure software elements, while the hardware components represent the components of the execution model [164]. The software components consist of the tasks code and HdS APIs, while the hardware components represent the abstract subsystems and the abstract communication network. The detailed description of these components will be illustrated in the following paragraphs.

4.2.1 Software Components

The application software is refined to tasks C code that contains the final application code and makes use of HdS API (Fig. 4.2). The tasks code represents sequential code, which implements a set of application functions. The communication primitives of the HdS API access explicit communication components. Each data transfer specifies an end-to-end communication path. For example, the functional primitives *send_mem(ch,src,size)/recv_mem(ch,dst,size)* may be used to transfer data between two processors using a global memory connected to the system bus, where *ch* represents the communication channel used for the data transfer, *src/dst* the source/destination buffer, and *size* the number of words to be exchanged. Thanks to the HdS APIs, the tasks code remains unchanged at the following abstraction levels (transaction-accurate architecture and virtual prototype).

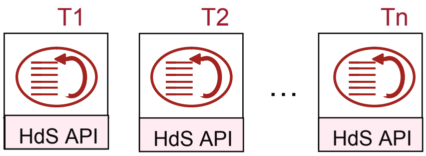


Fig. 4.2 Software components of the virtual architecture

Example 13. Software components for the token ring application at the virtual architecture level For the token ring application, the software is represented by the sequential C code corresponding to tasks T1, T2, and T3. This code implements the equivalent behavior of the different Simulink functions in C and contains the communication primitives *send(ch, src,size)/recv(ch,dst,size)* for the data exchange between the diverse tasks.

4.2.2 Hardware Components

The software tasks are executed using an abstract model of the hardware architecture that provides an emulation of the HdS API. The hardware platform is comprised of those components that provide the implementation of these HdS APIs. Thus, it includes the abstract subsystems, the abstract communication architecture (interconnection component), and the storage resources.

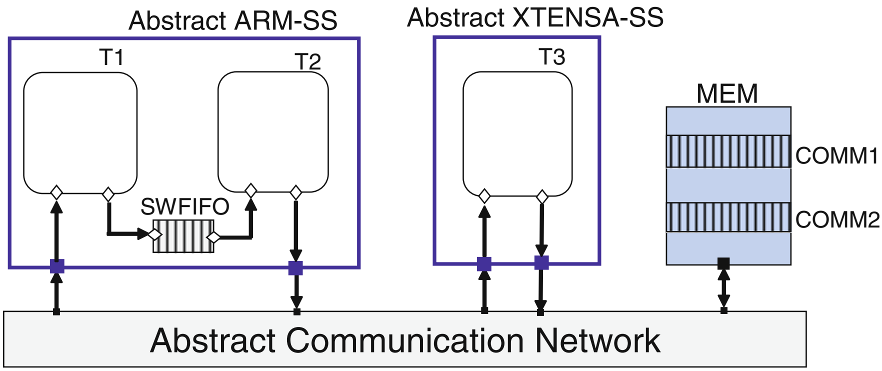


Fig. 4.3 Hardware components of the virtual architecture

Example 14. Hardware components for the token ring application at the virtual architecture level For the token ring application, the hardware is represented by the software subsystems (ARM-SS and XTENSA-SS), the global memory MEM, and the abstract communication network (an abstract AMBA bus). Figure 4.3 shows these hardware components.

4.3 Modeling Virtual Architecture in SystemC

The virtual architecture model is described using the SystemC language and is generated according to the parameters specified in the initial Simulink model. SystemC allows modeling a system at different abstraction levels from functional to pin-accurate register transfer level. The virtual architecture is modeled using transaction level modeling (TLM) techniques that allow analyzing SoC architectures in an earlier phase of the design, software development, and timing estimation [62].

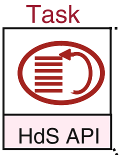
4.3.1 Software at Virtual Architecture Level

At the virtual architecture level, the Simulink functions of the application are transformed into C program code for each task. This step is very similar to the code generation performed by Real Time Workshop (RTW) [94].

The software at the virtual architecture level represents a multitasking C code description of the initial Simulink application model.

Each data link of the Simulink model requires a memory space called buffer memory to deliver data from the input block to the output blocks. To reduce the required memory size, the task code generation has to apply buffer memory optimization techniques, such as copy removal or buffer sharing [66].

The task C code is made of two parts: computation and communication (Fig. 4.4).




```
int B[10], C[20], D[10];
void task_code( )  {
      while(1) {
          recv(CH1, B, 10);       // Communication
           F1(B,C);               // Computation
           F2(C,D);
           …
           send(CH2, D, 10);     // Communication
          }
}
```

Fig. 4.4 Software at the virtual architecture level

The computation part describes the behavior of the various Simulink functions that are grouped in the task, including local memory declarations. The Simulink blocks within a task are scheduled statically according to their data dependency and generated into a task C code. The communication between functions inside a task is translated into local memory elements. To implement the external communication between the tasks, during the task code generation the function calls of the communication primitives are instantiated from an HdS API template library, preserving the invocation order of the blocks. Then, the allocated memory spaces are mapped onto the arguments of these functions. Before inserting the communication primitives, the data dependency between the tasks is checked during the task code generation in order to perform deadlock prevention.

Example 15. Software task code for the token ring application Figure 4.5 illustrates the C code of the task T2 of the token ring application at the virtual architecture level.

The task code contains the declaration of the local variables (*in*, *out*, *var*, and *var2*), the *send_data/recv_data* communication primitives, and the computation code. The tasks code starts with a receive operation to read the input token, then it performs some computation, and finally it sends the new value of the token to the next node.

The semantic of the communication primitives is the following: the first parameter represents the logic port which is connected to a communication channel, the second parameter is the local memory from where the data are transferred in case of a *send* operation or the local address where the data are stored in case of a *receive* operation, and the last parameter defines the size in words of the data to be sent or received.

In the computation code, the C code represents the equivalent behavior of the Simulink functions. Thus, the input data stored in the local variable *var* represents the input token. The destination of the token is calculated by a modulo 3 operation of the input tokens. If the result of this operation is 0, the destination of the token is task T2. In this case the task increments the token with one unit. Otherwise, task T2 is not the destination of the token and it increments the token with value 2, conform

```
#include <Task2.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifdef VIRTARCH
void Task2::behavior (void) {
#else
void Task2(void) {
#endif

int in,out;                                 //local variables
int var, var2;

for (;;) {
          recv_data(&In1_Task2, &in, 1);    //Communication API
          var = in % 3;                     // Computation
          var = abs(var);
          if (var == 0)
                    var2 = in + 1;
          else
                    var2 = in + 2;
          out = var2;
          send_data (&Out1_Task2, &out, 1); //Communication API
#ifdef VIRTARCH
          wait();
#endif
          }
}
```

Fig. 4.5 Task T2 code

the application specification. Then, the task forwards the token to the next node of the ring.

The multitasking C code generation from the system architecture model needs to handle a large subset of predefined Simulink blocks, such as mathematical operations (sum, multiplication, division, modulo, etc.), logical operations (AND, OR, XOR), discrete blocks (delay, mux, demux, merge), conditional structures (if–then–else), and repetitive structures (for-loop, while-condition-loop). The generation has to support also user-defined C code integrated in the Simulink model as S-functions. For the S-functions, the task code represents a function call of the user-written C function. The semantics of the argument passing are identical to those of the definition in the configuration panel of the S-function Builder tool in Simulink.

The resulted tasks code at the virtual architecture level is independent of the target processor, communication protocol, and abstraction level. This can be achieved by using HdS APIs that hide many details of the underlying implementation of the architecture and represent the abstraction of the hardware [167].

4.3.2 Hardware at Virtual Architecture Level

The hardware at the virtual architecture level consists of a set of hardware and software subsystems that encapsulate the tasks aimed to be executed on those subsystems, and the abstract communication network introduced to implement the communication protocol.

The hardware is refined to a set of abstract SystemC modules (SC_MODULE) for each subsystem. The SC_MODULE of the processor includes the tasks modules that are mapped on the processor and the communication channels for the intra-subsystem communication between the tasks inside the same processor. The communication channels between the tasks mapped on the same processor are implemented using standard SystemC channels. The tasks modules are implemented as SystemC modules (SC_MODULE).

For the inter-subsystem communication, the hardware architecture integrates also the resources addressed explicitly by the HdS APIs. Typical examples are memories that serve to store the communication buffers. The interconnection between the different components uses an abstract model of the communication network that allows the data transfer from the source to the destination module.

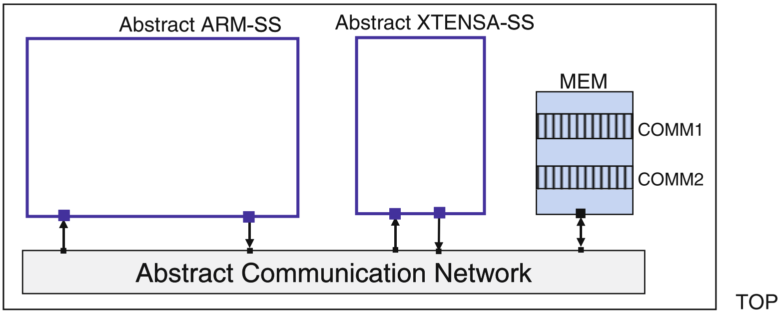
Example 16. Hardware code for the token ring application at the virtual architecture level Figure 4.6 details the top module for the token ring application running on the 1AX architecture. The top module is an SC_MODULE which includes the declaration and instantiation of the ARM-SS (*vARM7* in Fig. 4.6), XTENSA-SS (*vXTENSA*), abstract bus (*bus*), global memories (*gmem*), and a global clock (*clk*). It also interconnects these different components and fixes the addresses of the communication buffers used for the data exchange between the processors (the inter-subsystem communication units).

Thus, the communication buffer used between tasks T3 running on XTENSA and T2 running on ARM7 processors is mapped in the global memory at address 0×0. The communication buffer required for a data transfer between task T1 mapped on the ARM7 processor and T3 executed on the XTENSA is mapped on the global memory at address 0×1000.

Figure 4.7 shows the SystemC code for the ARM-SS component. The ARM-SS is an SC_MODULE which encapsulates the instances of the two tasks T1 (*vTask1* in Fig. 4.7) and T2 (*vTask2* in Fig. 4.7) and the software FIFO channel for the communication between them (*ch1* in Fig. 4.7).

This ARM-SS has three ports: an input port, namely, *In1_ARM7*; an output port, namely, *Out1_ARM7*; and a clock port (*clk*). The output port of the ARM-SS is connected to the output port of task T2, as task T2 sends the data to a task mapped on the other processor. The input port of the ARM-SS is connected to the input port of task T1, because task T1 needs external data from a task running on the other processor.

At the virtual architecture level, the tasks code uses HdS APIs, whose implementation depends on the hardware platform. The hardware platform includes all the

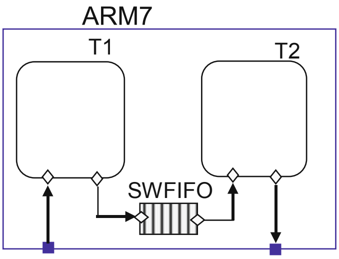



```
ifndef _TOP_H
#define _TOP_H

#include <systemc.h>
#include "XTENSA.h"
#include "ARM7.h"
#include "global_bus.h"
#include "mem.h"

SC_MODULE(TOP) {                                  //TOP MODULE

sc_in<bool> clk;
XTENSA *vXTENSA;                                  // XTENSA-SS MODULE
ARM7 *vARM7;                                      // ARM-SS MODULE
global_bus_module *bus;                           // BUS CHANNEL
mem *gmem;                                        // GLOBAL MEMORY MODULE

SC_CTOR(TOP) {
          vXTENSA = new XTENSA("XTENSA");         //INSTANTIATION
          vARM7 = new ARM7("ARM7");
          bus = new global_bus_module("BUS");
          gmem = new mem("mem",0x2000);

          gmem->port(*bus);                       //CONNECTION
          gmem->port.set_map(0x0,0x1FFF);
          vXTENSA->In1_XTENSA(*bus);
          vXTENSA->vTask3->In1_Task3.set_connect_address(0x0);
          vARM7->Out1_ARM7(*bus);
          vARM7->vTask2->Out1_Task2.set_connect_address(0x0);
          vARM7->In1_ARM7(*bus);
          vARM7->vTask1->In1_Task1.set_connect_address(0x1000);
          vXTENSA->Out1_XTENSA(*bus);
          vXTENSA->vTask3->Out1_Task3.set_connect_address(0x1000);
          vARM7->clk(clk);
          vXTENSA->clk(clk);
          }
};
```

Fig. 4.6 SystemC code for the top module

components accessed by the HdS APIs and the resources to implement the required communication paths.

Example 17. Communication primitives implementation for the token ring application at the virtual architecture level Figure 4.8 shows an example of implementation of the *send_data(...)/recv_data(...)* communication primitives that allow to write or read to/from a software FIFO communication channel.




```
#ifndef _ARM7_H
#define _ARM7_H

#include "Task1.h"
#include "Task2.h"

SC_MODULE(ARM7) {                         // ARM-SS MODULE

Task1 *vTask1;                            // TASK T1 MODULE
Task2 *vTask2;                            // TASK T2 MODULE
SWFIFO_Channel ch1;                       // SOFTWARE FIFO CHANNEL
AMBA_Port Out1_ARM7;                      // PORTS
AMBA_Port In1_ARM7;
sc_in<bool> clk;

SC_CTOR(ARM7) {
          vTask1 = new Task1("Task1");   //INSTANTIATION
          vTask2 = new Task2("Task2");

          vTask1->Out1_Task1(ch1);       //CONNECTION
          vTask2->In1_Task2(ch1);
          vTask2->Out1_Task2(Out1_ARM7);
          vTask1->In1_Task1(In1_ARM7);
          vTask1->clk(clk);
          vTask2->clk(clk);
          }
};

#endif
```

Fig. 4.7 SystemC code for the ARM-SS module

The FIFO channel is derived from a SystemC channel (sc_prim_channel) and has a blocking implementation. Therefore, if the sender wants to put data into the FIFO, but the FIFO is full, the sender will be blocked until there is enough available space in the buffer. This blocking implementation is employed by calling the *wait ()* statement in the implementation of the *send_data* primitive if there is not enough space available. The *wait()* call suspends the execution of the task. In the same manner, if a task calls a *recv_data* primitive, but there is not enough data stored in the FIFO buffer, the receiver will be blocked until the FIFO contains the requested number of elements. The FIFO buffer can be characterized by size and depth. The size represents the number of elements stored in the buffer. The depth

```
class SWFIFO_Channel: public sc_prim_channel,       //SOFTWARE FIFO CHANNEL
public swfifo_if
{
int buffer[30000];
unsigned int sizemax;
int buffer_size;

public:
      SWFIFO_Channel() {
                    sizemax=SIZEMAX_swfifo;
                    buffer_size =0;
                    }
      virtual void recv_data(const SWFIFO_Port& port, void* dst, int size);
      virtual void send_data(const SWFIFO_Port& port, void* src, int size);
      virtual void init(const SWFIFO_Port& port, int size);
};


void SWFIFO_Channel::send_data(const SWFIFO_Port& port,       // SEND_DATA
void* src, int size)
{
 while((sizemax - buffer_size) < (unsigned)size)
          wait();

 for (int i=0; i<size; i++)
          buffer[buffer_size+i]=(((int*)src)[i]);

 buffer_size = buffer_size + size;
}
```

Fig. 4.8 Example of implementation of communication channels

represents the number of bits necessary to store one element. Each element occupies the same number of bits. In this example (Fig. 4.8), the FIFO buffer has a size equal to 30,000 and depth equal to 1 word (32 bits).

At the virtual architecture level, all the modules are connected to the same clock signal. Typically the clock is created in the main function of the top level and passed down through the module hierarchy to the rest of the system. This allows a subset of components or the entire system to be synchronized by the same clock. The clock signal has a set of attributes, such as default time unit, period, duty cycle, first edge, and first value. The default time unit is assumed to be 1 ns. The period represents the number of default time units required by the clock signal to make a complete transition from true (high) to false (low) and back from false (low) to true (high). The duty cycle is the ratio of the high time to the entire clock period. Example, if the period of a clock signal is equal to 20 default units and the duty cycle is 0.25, this means that the clock would stay in true state for 5 time units and false for 15 time units. The first edge represents the offset time from 0 of the first edge expressed in time units. The first value represents the starting value of the clock (true or false).

Example 18. sc_main for the token ring application at the virtual architecture level Figure 4.9 presents the main function (*sc_main*) for the token ring application at the virtual architecture level. This includes the initialization of the top module, the declaration of the global clock signal, the connection of the clock signal to the clock

```
int sc_main(int argc, char ** argv)
{
          TOP top_module("TOP");            //TOP MODULE INSTANTIATION

          sc_clock s_clk("s_clk",20,0.5,0);  // CLOCK SIGNAL
          top_module.clk(s_clk);

          sc_start(-1);                     //START SIMULATION
          return 0;
}
```

Fig. 4.9 SystemC main function

port of the top module, and the launch of the simulation. The clock has a period of 20 ns and duty cycle 0.5.

4.3.3 Hardware–Software Interface at Virtual Architecture Level

The hardware–software interface defines the software–hardware interaction and how the software can access the hardware. At the virtual architecture level, the hardware/software interface consists of a set of task modules. The task module is a SC_MODULE which encapsulates the software code within a SystemC clocked thread (SC_CTHREAD). The software code can access the hardware through the ports of the task module.

Example 19. Task module for the token ring application at the virtual architecture level Figure 4.10 illustrates an example of task module for the task T2. The task module contains the declaration of the logic ports. The type of the ports depends on the type of the communication channel which is accessed. For instance, the input port of task T2 *In1_Task2* is connected to the software FIFO channel, thus the port has the type SWFIFO_Port. Task T2 writes to task T3 running on the XTENSA processor via the AMBA bus. Therefore, the type of the output port *Out1_Task2* is AMBA_Port.

Task T2 executes a clocked SystemC thread, namely, the *behavior* function. The *behavior* function is defined in the task software code, as illustrated in Fig. 4.5. A clocked SystemC thread represents a thread of execution which is sensitive only to the positive or negative edge of a clock signal.

4.4 Execution Model of the Virtual Architecture

The virtual architecture level allows debugging the task code. The following sections will describe the simulation model in SystemC and the adopted configuration to validate the virtual architecture model.

The executable model is obtained by compiling the task code and the hardware platform together. The resulted executable model uses the SystemC scheduler

```
#ifndef _Task2_H
#define _Task2_H

#include <systemc.h>
#include "swfifo.h"
#include "amba.h"

SC_MODULE (Task2) {                                    //TASK T2 MODULE

          sc _in<bool> clk;

          SWFIFO_Port In1_Task2;
          AMBA_Port Out1_Task2;

          void behaviour();

          SC_CTOR(Task2) {

                    SC_CTHREAD(behavior,clk);   // THREAD

          }
};

#endif
```

Fig. 4.10 Example of hardware/software interface

to activate and deactivate the execution of the different tasks. The processor and memories are SystemC modules. The abstract network component (bus, NoC, or point-to-point communication channels) and the software FIFO channels are derived from the SystemC channels. The software tasks are SystemC clocked threads. A clocked thread has its own thread of execution which may accept only positive or negative edge clock event in its sensitivity list. When the simulation starts, the clocked threads are automatically activated.

The simulation at the virtual architecture level allows validating the tasks C code of the refined software and the hardware–software partitioning. It represents a native execution of the software onto the simulation host machine. High simulation speed is usually attained, but abstracting the hardware architecture it lacks some accuracy.

The simulation at the virtual architecture level allows avoiding communication deadlock due to improper scheduling of the communication operations between the different tasks. The debug of the software code may be done using standard C debuggers such as gdb or by tracing waveforms. SystemC provides functions to create a VCD (value change dump) or ASCII WIF (waveform intermediate format) file that contains the values of the variables and the signals as they change during the simulation. The waveforms can be viewed using standard waveform viewers that support the VCD and WIF formats, such as gtkwave.

4.5 Design Space Exploration of Virtual Architecture

4.5.1 Goal of Performance Evaluation

The goal of the performance evaluation at the virtual architecture level is to allow profiling the communication and computation requirements to improve the overall performances of the system. The objective is to provide through simulation statistical information, such as utilization of the architecture model components (busy/idle times), the degree of the contention in a system, profiling information (the time spent in different executions), critical path analysis, or average bandwidth between the architecture components.

Based on the application requirements and the communication traffic resulted after the virtual architecture simulation, the designer can fix some hardware and software architecture decisions. Examples of hardware architecture decisions are the topology of the interconnect component that will be included in the hardware platform at the next abstraction level (NoC topology) or the communication scheme between the different subsystems fixing the mapping of the communication buffers onto the storage resources. Examples of software architecture decisions are application partitioning into tasks and mapping onto the processing subsystems and the semantic of the communication primitives used in the final application tasks code.

These different decisions influence the overall execution time of the system, cost, and power consumption. Therefore, good decisions are required to be able to control the MPSoC design process.

Example 20. Goal of performance evaluation for the token ring application at the virtual architecture level For example, in case of the token ring application, the designer fixes at the virtual architecture level the partitioning of the application into three tasks and the mapping of the FFT computation onto the XTENSA processor. Also, the communication between the processors is decided to be performed via the global memory.

4.5.2 Architecture/Application Parameters

The virtual architecture model has to fix some parameters that can influence the global performance of the final system. The parameters represent a subset of those specified at the system architecture level. The virtual architecture validate some decisions taken at the system architecture level, such as partitioning and mapping, while other parameters are preserved in order to be validated at the next levels. The preserved parameters can be specific to subsystems or communication units, as it will be detailed in the following paragraphs.

(a) Architecture/Application Parameters Specific to Subsystems: The parameters specific to subsystems characterize the different subsystems from hardware and software points of view. The hardware architecture parameters that characterize the subsystems at this level will be validated at the following abstraction levels:

- *NetworkType* to specify the type of the network component used to interconnect the different subsystems, such as AMBA bus or network on chip (NoC).
- *NoCTopology* to specify the bus or NoC topology (mesh or torus).
- *NoCRoutingAlgorithm* to specify the routing algorithm used by the routers to transmit the received data packet in case of a NoC network component.
- *NoCArbitrationAlgorithm* to specify the type of the arbitration algorithm inside a NoC router (e.g., round robin, priority based).

The software architecture parameters that characterize each processor subsystem are as follows:

- *OSType*, which specifies the name of the operating system running on the target processor (e.g., Linux, Mutek, DwarfOS, eCos).
- *SchedulerType* to identify the type of the scheduler (preemptive, cooperative).
- *SchedulerAlgorithm* to define the algorithm used for the tasks management by the operating system (round robin, priority based), etc.

(b) Architecture/Application Parameters Specific to Communication Units. The communication primitives used for the data exchange in the tasks code are fixed at the virtual architecture level. Therefore, the parameters that characterize the communication units and will be validated at the next abstraction levels rely on the hardware architecture. Example of this kind of parameters is the *AccessType* which identifies the type of the access to the memory (directly or through DMA) making the communication path end to end.

4.5.3 Performance Measurements

At the virtual architecture level, the performance measurement consists of profiling the communication and computation requirements for each task or for each processor.

The virtual architecture has the notion of time due to the clock signals. Therefore, by simple annotation of the virtual architecture model with adequate execution delays, if such delay information is available, the simulation at this level can estimate the total clock cycles spent on communication or computation by the task or processor. But, the accuracy of the estimation is not yet cycle accurate at this level, since not all the hardware components or hardware features (e.g., final bus arbitration scheme, interconnect topology, peripherals) are explicit in the model. The execution time represents these estimated clock cycles required to run an application on the MPSoC architecture. The simulation time represents the time needed to simulate the behavior of the application running on the architecture with their interaction.

Examples of metrics that can be measured at the virtual architecture level are the tasks code and data size, the buffer size required for the intra-subsystem and inter-subsystem communication, the total quantity of exchanged data between the tasks during the execution, the number of iterations of a function execution, the amount of data transferred between the different processors, the amount of data passing through the global interconnect component, the buffer size requirements in the worst-case scenario for the storage resources in order to support the communication mappings specified at the system architecture level, or the amount of read/write operations performed at the storage modules.

By tracing the waveforms of the signals or variables during the simulation, other metrics can be measured, e.g., the cycles spent by a task on computation and communication, or the current task executed during the simulation at a certain moment of time.

Example 21. Performance measurements for the token ring application at the virtual architecture level For example, the total simulation time of the token ring application was 3 s to execute 68 clock cycles of period 20 ns, required to run the entire application. But in this example, the model is not annotated with accurate information regarding the operating system and the communication overhead. Thus, the estimation is only message-level accurate.

In the case of the token ring application, the total code size and data size of the tasks code running on both processors is 32,223 bytes, respectively, 12 bytes. The total number of bytes passed through the bus during the simulation is 3,136 bytes.

Figure 4.11 shows the waveforms captured during the simulation of the token ring application, e.g., at time 13,100 ps, the current task running on the ARM processor was T1, while task T2 was blocked on communication.

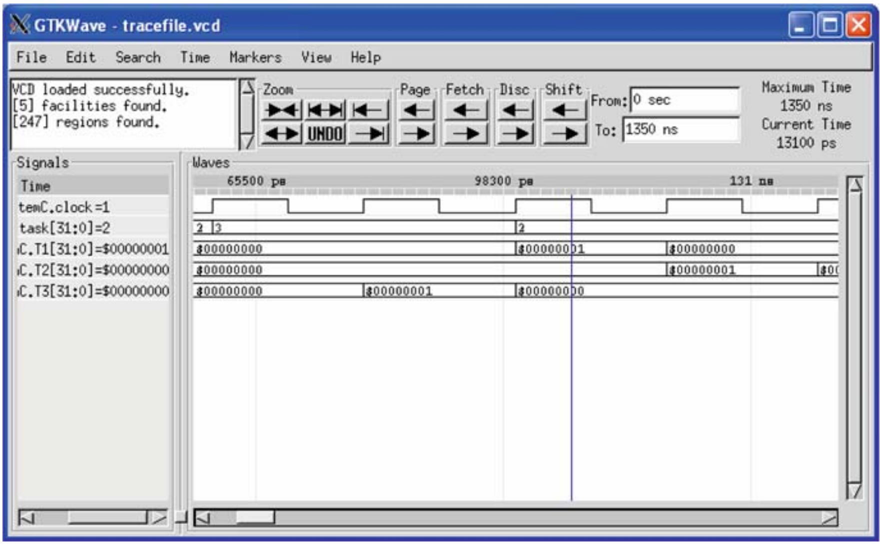


Fig. 4.11 Waveforms traced during the token ring simulation

4.5.4 Design Space Exploration

At the virtual architecture level, the design space exploration covers architecture exploration, more precisely communication architecture exploration. The designer can experiment different communication mapping schemes and different communication primitives. The designer may adopt different communication protocols and may map the communication buffers onto different storage resources. The different communication and synchronization schemes have advantages and disadvantages in terms of performance (latency, throughput), resource sharing (multitasking, parallel I/O), and communication overhead (memory size, execution time). Also, the tasks code can be generated using different tools, such as Real Time Workshop.

Example 22. Design space exploration for the token ring application at the virtual architecture level In the case of token ring application, the designer may map the buffers required for the inter-subsystem communication onto different architecture resources, such as the local memories of both ARM and XTENSA processors, or the shared global memory, or on the hardware FIFO.

4.6 Application Examples at the Virtual Architecture Level

The following sections detail the virtual architecture model for the two case studies considered in this book: the motion JPEG decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 encoder application running on the Diopsis R2DT architecture with Hermes NoC.

4.6.1 Motion JPEG Application on Diopsis RDT

This section presents the virtual architecture design in case of the Motion JPEG (MJPEG) decoder application running on the Diopsis RDT platform. The virtual architecture design consists of two steps: software design and hardware design.

First, the C code for each task was generated from the Simulink system architecture model based on the annotation with the software architecture parameters. In the system architecture model, the value attributed to parameter *CommType* is equal with "MPI." Therefore, the generated task code uses *send_data(...)/recv_data(...)* for the communication primitives. Moreover, the C code was optimized by applying buffer sharing and copy removal memory optimization techniques.

In order to evaluate the efficiency of the software task code, a comparison with the single task code generation from Simulink using Real Time Workshop (RTW) is given. Table 4.1 resumes the code and data size of the generated application code. The code library contains the user-defined C-functions commonly used by all the code generator tools and independent of the software design method. The application task code obtained by applying memory optimization techniques is more

Table 4.1 Task code generation for motion JPEG

	Library		RTW		Multitask code	
Size (bytes)	Code	Data	Code	Data	Code	Data
MJPEG	6818	32	8225	72	8032	494

efficient in terms of code size than the code generated using RTW. But the multitasking representation requires communication buffers. Therefore, the data size is bigger than in the case of RTW.

The second step of the virtual architecture design represents the hardware design. The hardware design consists of building the software development platform in SystemC considering the hardware architecture parameters that annotate the system architecture model. Figure 4.12 illustrates a conceptual view of the virtual architecture of the Diopsis RDT architecture with AMBA bus.

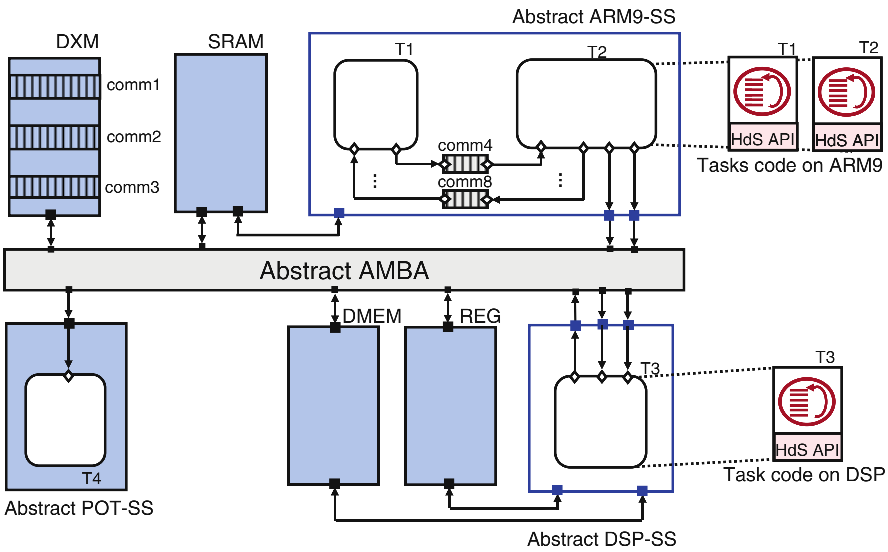


Fig. 4.12 Global view of Diopsis RDT running MJPEG

The virtual architecture platform contains all the components that are accessible by the software through the *send_data(...)/recv_data(...)* HdS APIs. The inter-subsystem communication units are partially mapped on the memory modules (DXM, REG, SRAM, DMEM), attached as slave components to the AMBA bus. The impartiality comes from the abstraction of the hardware architecture and making implicit several hardware characteristics (e.g., local bus, DMA, bus bridges). The address space of the components are automatically assigned and computed by using a template that contains a predefined address size for each component. The communication buffers between the different subsystems are mapped on the corresponding memory modules based on the protocol specified at the system architecture level.

As showed in Fig. 4.12, the virtual architecture contains the following components: abstract ARM9-SS, abstract DSP-SS, abstract POT-SS, and the storage resources: DXM, local memory SRAM of the ARM9 processor, data memory DMEM of the DSP processor, and data register REG of the DSP. All the components are interconnected using an abstract AMBA bus. The ARM9-SS contains the two task modules that are running on it (T1 and T2) and the five intra-subsystem communication channels (*comm4*, *comm5*, *comm6*, *comm7*, and *comm8*). According to the system architecture annotation, all these communication channels are implemented as FIFO channels. The DSP-SS includes the task module T3. The POT-SS includes the task module T4 responsible with the display of the decoded image. The three communication units between the subsystems, more precisely *comm1*, *comm2*, and *comm3*, may be mapped on different storage resources. In a first case, the *ResourceTyp*e of each communication unit has the value "DXM." Hence, the system architecture model is specified to use the external memory as buffer storage for the communication between the different subsystems. Therefore, all three were mapped on the external memory DXM.

The abstract AMBA bus is implemented as a simple bus which transfers data initiated by a master. It allows connection of several master and slave subsystems, but only one data transfer request can be accomplished in time. The arbitration of the simultaneous requests for a data transfer through the bus is performed in a queue manner. As shown in Fig. 4.13, the bus is made of two components: virtual arbiter and address decoder.

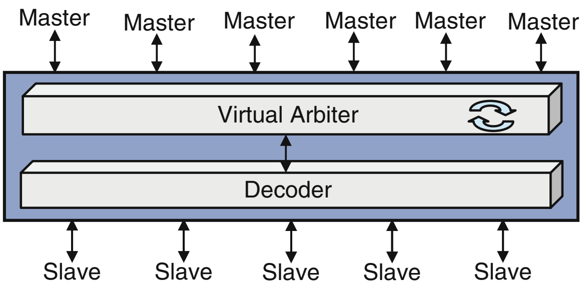


Fig. 4.13 Abstract AMBA bus at virtual architecture level

The arbiter, also called scheduler, controls the data traffic. If data are to be transferred, the requesting master subsystem sends a message to the scheduler. The scheduler checks if the slave subsystem is ready for the data transfer. If it is ready, the scheduler puts the request into a FIFO queue. Otherwise, it waits until the availability of the slave component and checks its status by polling. This mechanism allows avoiding blocking the bus for a data transfer to a destination or source which is not yet ready to receive or send data. The request message contains an identification code of the target subsystem, which represents the address of the slave component. The decoder is responsible to identify the slave subsystem. As soon as the bus is available, the access to the bus is granted to the requesting subsystem, which can perform the data transfer to the destination address. Having completed the data transfer, the bus becomes free for the next request in the scheduler's queue. The AMBA bus allows transfers in burst mode, which means that the master may transfer the whole data message within one access grant to the bus [6]. Through polling the status of the destination subsystem, the virtual architecture bus provides synchronization mechanism between the different subsystems similar to semaphores. At this level, the bus does not yet implement the arbitration strategy of the target AMBA bus protocol.

The functionality of the software code was verified by execution using the hardware platform. The software code was compiled with the architecture platform. During the execution, the tasks are scheduled by the SystemC simulation engine.

Besides the tasks code verification, the simulation model also allowed to gather important early performance measurements, e.g., total number of messages transferred through the AMBA bus. The data transfer between the ARM9 and DSP processor subsystems is performed in messages of 64 words for the IDCT coefficients and in 1 word for the decoding pattern; the data transfer between the DSP-SS and POT-SS is performed in messages of 16 words.

Table 4.2 shows the results for the different communication schemes. Using as communication units only the DXM, the bus was accessed to transfer 216,000 messages during the decoding process of the 10 frames. If the communication units are mapped on different resources, for example, *comm1* is mapped on DXM, *comm2* on REG, and *comm3* uses DMEM memory to store the communication buffer, the global bus was accessed to transfer 144,000 messages during the simulation. In the third scheme, *comm1* and *comm2* are mapped on SRAM, while *comm3* remains mapped on DMEM. Thus, all the communication units make use only of local memories SRAM and DMEM. In this case, the execution required 108,000 messages to be transferred via the AMBA bus. In all the communication schemes, the communication units between the two tasks running on the ARM9 processor *comm4*, *comm5*, *comm6*, *comm7*, *comm8*, and *comm9* implements the software FIFO protocol.

Table 4.2 Messages through the AMBA bus

Comm. unit	Comm1	Comm2	Comm3	Comm4-Comm8	Total messages AMBA	Execution time (ns)
	DXM	DXM	DXM	SWFIFO	216,000	4,464,060
MJPEG	DXM	REG	DMEM	SWFIFO	144,000	3,720,060
	SRAM	SRAM	DMEM	SWFIFO	108,000	2,232,020

This simulation model was accurate enough to verify the functionality of the tasks code and ensure that there is no communication deadlock in the scheduling of the data transfer between the tasks. The simulation time required to decode the 10 image frames encoded using QVGA YUV 444 format was approximately 14 s on a PC running Linux OS at 1.73 GHz in all the cases of the communication schemes.

The total execution time required by the whole decoding process is illustrated in Table 4.2. These numbers were estimated without annotation of the code with execution delays. Therefore, the accuracy of the estimation relies on message level. As the DXM communication scheme supposes all the data exchange through the AMBA bus, it requires the highest number of execution cycles, in a total time of approximately 4,464,060 ns, due to the conflicts that appear on the shared bus when simultaneous bus requests occur. In the mixed communication scheme with DXM, REG, and DMEM, the total execution time is estimated to be 3,720,060 ns, while the last communication scheme guarantees the fastest execution with 2,232,020 ns. The numbers for the required execution time are obtained by calling *sc_simulation_time()* at the end of the execution in the top module of the SystemC platform. All the subsystems are interconnected to the same clock signal with a period of 20 ns and duty cycle 0.5.

Figure 4.14 presents a screenshot during the SystemC simulation of the MJPEG decoder application at the virtual architecture level.

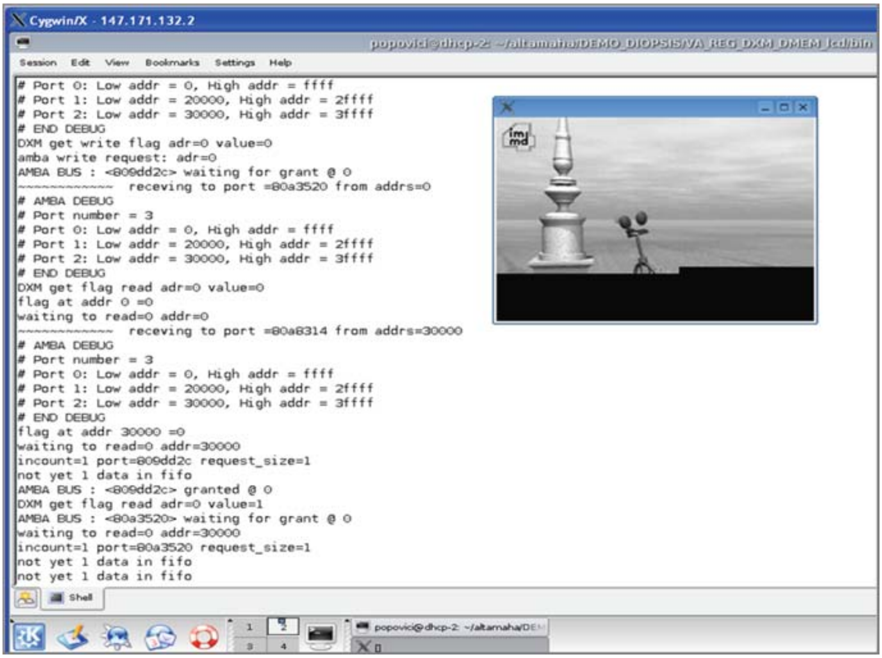


Fig. 4.14 Virtual architecture simulation for motion JPEG

4.6.2 H.264 Application on Diopsis R2DT

This section presents the virtual architecture design in case of the H.264 encoder application running on the Diopsis R2DT platform. The virtual architecture design is accomplished in two steps: software design and hardware design.

The software design consists of generating the C code for each task from the system architecture model using the software architecture parameters. The *CommType* parameter annotating each subsystem determines the communication primitives supported by the operating system. Similar to the Motion JPEG example, the *CommType* is equal with "MPI." Therefore, the generated task code uses *send_data(...)/recv_data(...)* for the communication primitives. The code is optimized in terms of data memory requirements.

Table 4.3 shows the task code and data size of the software at the virtual architecture level. The first column represents the code and data size of the functions that are independent of the design method. The second column shows the code and data size for the generation using Real Time Workshop. Real Time Workshop generates single-tasking code, while the software at the virtual architecture level represents multi-tasking code. The last column represents the results for the software design method with memory optimization techniques.

Table 4.3 Task code generation for H.264 encoder

	Library		RTW		Multitask code	
Size (bytes)	Code	Data	Code	Data	Code	Data
H.264	270,994	132	296,305	148	366,060	148

The second step of the virtual architecture design represents the hardware architecture design. The hardware design consists of building the software development platform in SystemC considering the hardware architecture parameters that annotate the system architecture model. Figure 4.15 illustrates a conceptual view of the virtual architecture for the Diopsis R2DT architecture with Hermes NoC.

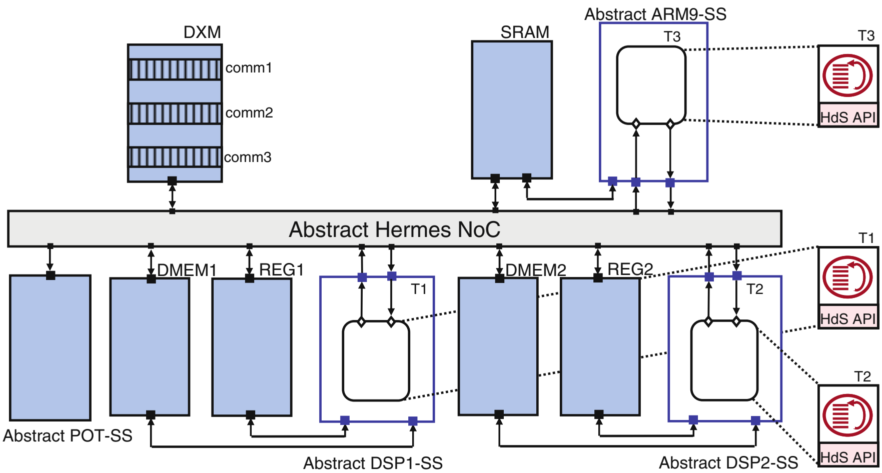


Fig. 4.15 Global view of Diopsis R2DT running H.264

The virtual architecture platform contains all the components that are accessible by the software through the *send_data(...)/recv_data(...)* HdS APIs. Thus, the platform contains the following modules: four abstract subsystems, namely the ARM9-SS, DSP1-SS, DSP2-SS, and POT-SS, and the local and global memory modules: DXM shared by all the subsystems, SRAM local memory of the ARM9-SS, DMEM1 and REG1 memories of the DSP1-SS, respectively, DMEM2 and REG2 memories of the DSP2-SS. All these components are interconnected using an abstract NoC model. The DSP1-SS contains the task module of T1. The DSP2-SS includes the task module of T2. Finally, the ARM9-SS encapsulates the task module of the third task T3. The three communication units between the different processors, more precisely the *comm1*, *comm2*, and *comm3*, may be mapped on different storage resources, according to the system architecture specification. Figure 4.15 shows an example of mapping of the communication units onto the DXM global memory.

At this level, each local memory has allocated an address space of 4 MB. The global memory has an address space of 256 MB.

The NoC at the virtual architecture level represents an abstract NoC where information like topology, routing algorithm, arbitration, or buffer size information are omitted. Communication architecture is modeled like a crossbar, where any set of communication events may occur simultaneously.

Figure 4.16 details the model of the Hermes NoC at the virtual architecture level.

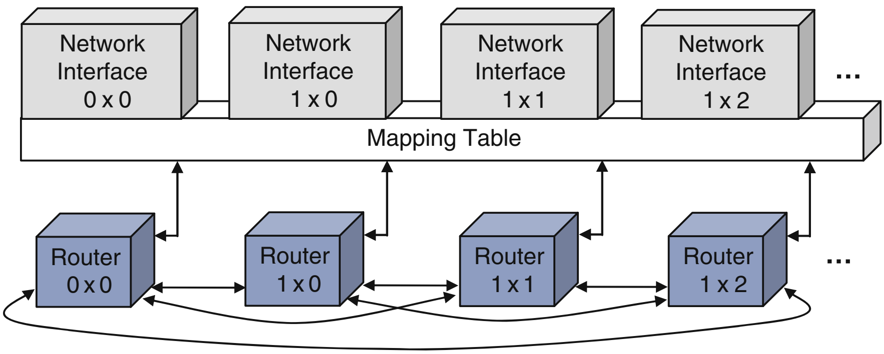


Fig. 4.16 Abstract Hermes NoC at virtual architecture level

The NoC is comprised of three basic elements, which are the network interface (NI), the mapping table (MT), and the router. The network interface is responsible for providing *send/receive* operations for the communicating subsystems, encapsulating these requests in packets, capturing and interpreting packets arriving from the NoC, and delivering them to the subsystems. The mapping table is responsible for storing and informing the correspondence between the IP cores range address and the NoC physical address, i.e., IP core address between 0×00400000 and $0\times007FFFFF$ corresponds to the NoC physical address 0×0. The router is in charge of transporting packets from the source network interface to the destination network interface.

The Hermes NoC for the Diopsis R2DT architecture involves five routers at the virtual architecture level. Each router is connected to the corresponding network interface and the other four routers. The network interfaces connect the following IP cores to the NoC: ARM9-SS, POT-SS, DXM, DSP1-SS, and DSP2-SS. One network interface is associated with each subsystem. Therefore, SRAM and ARM9 share the same network interface with address 1×0. The local memories REG1 and DMEM1 share the network interface with address 1×1 with the DSP1 processor core. The network interface with address 1×2 connects the REG2, DMEM2, and DSP2 components to the NoC. The network interface corresponding to the DXM has address 0×0. Finally, the network interface connecting the POT-SS has address 0×1.

The functionality of the software code was verified by execution using the hardware platform. The software code was compiled with the architecture platform. During the execution, the tasks are scheduled by the SystemC simulation engine. The simulation model is accurate enough to verify the functionality of the tasks code and ensure that there is no communication deadlock in the scheduling of the data transfer between the tasks.

Besides the tasks code verification, the simulation model allowed also to gather important early performance measurements, e.g., the number of words exchanged between the tasks through the network component. The virtual architecture simulation allows capturing information regarding the communication values through the NoC. Such values are the amount of data exchanged between the different subsystems, the storage elements worst-case size requirement for the communication buffer, the number of operations (send/receive) originated from each access point of the NoC, the amount of read/write operations performed at the storage elements, and the NoC area based on the number of routers.

Figure 4.17 shows these numbers in case of different communication mapping schemes. Hence, when all the communication buffers are mapped on the DXM memory, as shown in Fig. 4.15, the NoC was accessed to transfer 6,171,680 words during the encoding process of the 10 frames. In another case, *comm1* is mapped on DXM, *comm2* on REG2, and *comm3* on DMEM1. This case required 5,971,690 words to be transferred through the NoC. A third case maps *comm1* on DMEM1, *comm2* on DMEM2, and *comm3* on SRAM and it generates 3,085,840 words to be operated by the NoC.

Table 4.4 shows some results captured during the simulation of the H.264 encoder application in the case of the first communication scheme with all the buffers mapped on the DXM memory. The first and the second columns represent the correspondence between the different cores connected to the NoC and the NoC addresses. The third column represents the total number of reads and writes requested over the NoC. Based on these values the designer may define a better mapping of the hardware or the size of the packets. The fourth and the fifth columns (packets and mega bytes sent) allow evaluating the real amount of the communication injected in the NoC through each network interface. The DXM was the core that

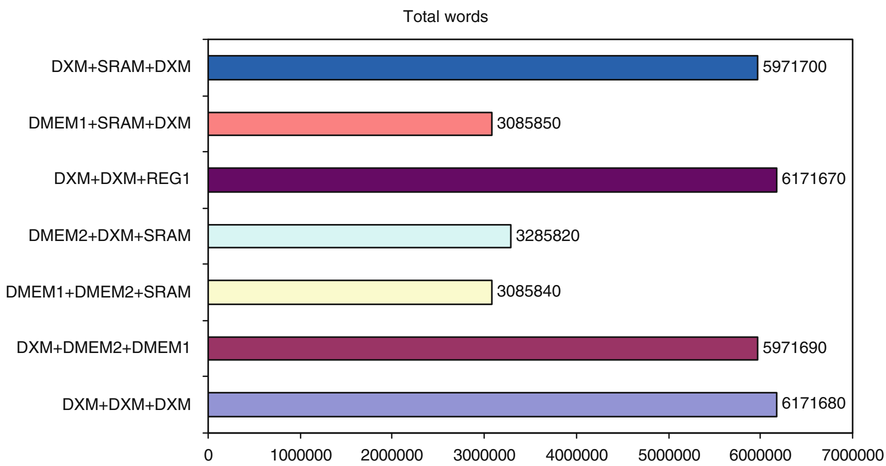


Fig. 4.17 Words transferred through the Hermes NoC

Table 4.4 Results captured in Hermes NoC using DXM as communication scheme

H.264	NoC address	Read/write requests	Total packets sent	Mega bytes sent
DXM	0×0	0	83,352	17,324
ARM9-SS	1×0	2,426	4,853	68
DSP1-SS	1×1	39,260	78,522	16,167
DSP2-SS	1×2	41,663	83,327	2,090

inserted the biggest amount of data in the NoC. The DXM packets are originated from read requests and confirmation packets.

In the third communication scheme, the simulation time required to encode the 10 image frames using QCIF YUV 420 format was approximately 32 s on a PC running Linux OS at 1.73 GHz. The execution time of the encoding process without accurate execution delay annotation implies 2,546,540 ns or 127,327 clock cycles with a common clock used by all the modules with the following settings: a period of 20 time units, a duty cycle of 50%, the first edge will occur at 0 time units, and the first value is true. All the modules are synchronized by the same clock. The default time unit is assumed to be 1 ns.

4.7 State of the Art and Research Perspectives

4.7.1 State of the Art

The concept of the virtual architecture is used in several academic and industrial MPSoC design environments. There are several modeling and simulation environments of the virtual architecture.

For example, in the ROSES hardware/software co-design tool, the virtual architecture is defined as a system made of an abstract netlist of virtual components. A virtual component consists of an internal component (or module) and its wrapper for adaptation to different communication protocols, abstraction levels, or specification languages. The virtual components are interconnected by virtual channels through virtual ports [32].

A similar definition of the virtual architecture is given in [143]. They define the virtual architecture as a system in which processing elements communicate via abstract channels.

Reference [79] defines the virtual architecture as an intermediate phase of the SoC design flow, where the functionality of the system is mapped to the architecture in an abstract manner to enable architecture optimization across heterogeneous computational components. The virtual architecture is annotated with timing characteristics of the target architecture, thus it allows fast exploration of different design alternatives. In this approach, the timing-related aspects are captured by the communication channels.

Reference [57] identifies the abstraction levels based on the communication refinement from abstract message passing down to cycle-accurate bus functional implementation. In their work, the communication design starts from a virtual architecture model. The virtual architecture is defined as a system comprised of processing elements that communicate via abstract channels with untimed synchronous or asynchronous message-passing semantics. The virtual architecture presented in this book is similar to the one defined in [57], but it contains also explicit mapping of the communication buffer onto the storage components and explicit abstract interconnect component.

Other research works focus on automatic generation of the virtual architecture. Thus, [111] introduces the ESPAM tool, which automatically generates C/C++ software code for each processor from an application specification in form of KPN. The code contains the main behavior of a process, together with the blocking read/write synchronization primitives, and the memory map of the system. The resulted code is similar to the tasks code at the virtual architecture presented in this book.

Another example of code generation is the MCS (MATLAB-to-C synthesis) product of Catalytic Inc., which provides automatic functionally equivalent C code generation from MATLAB models for the algorithm developers [31]. The generated C code mimics the MATLAB code file structure and the function hierarchy. A graphical user interface provides viewing and cross-probing between MATLAB and C code. The tool offers support for all the common data types, all the operators, and a large number of built-in functions. But it does not include the testbench, visualization, or plotting capabilities of MATLAB.

4.7.2 Research Perspectives

Future research perspectives of the virtual architecture concern the following aspects:

– Automatic generation of the hardware architecture and the software code
– Automatic annotation of the software and hardware code with timing information for accurate performance estimation
– Formalization of fleeting from system architecture to virtual architecture

One of the main challenges in the SoC development flow is the consistency between different levels of abstraction of the system to be implemented [19]. The quality of the design can be preserved by automatic generation of the abstraction levels, including the virtual architecture generation. The automatic generation of the virtual architecture implies generation of the software code and the hardware platform. The generation is achievable due to the annotation with architecture attributes of the initial specification in the form of the system architecture.

The software and hardware architectures are natively executed on a simulation host, without using a software simulator such as instruction set simulator (ISS). Therefore, to obtain an accurate estimation of the execution time for an application, such as number of cycles spent by the processor on computation or waiting for the communication, the virtual architecture code has to be orchestrated with additional timing information, like the number of cycles required by the processor to compute a function. The automatic annotation of the generated code (software and hardware) with timing information can be accomplished by inserting *wait(delay)* statements in the SystemC code of the architecture.

The passing from the system architecture level to the virtual architecture level needs to be done in a rigorous way which ensures the right preservation of the initial specification in terms of design constraints. This may be achieved through a formalization of the system architecture, virtual architecture, and the formalization of the conversion from the high level to the more detailed lower level. The considered aspects could be the model of computation and the model of execution that characterize each abstraction level, and the definition of the rules that guarantee a correct translation from one model to another.

4.8 Conclusions

This chapter defined the virtual architecture design. It presented the software representation as final application tasks code and the hardware organization in abstract subsystems interconnected through an abstract network component.

The virtual architecture design was performed using SystemC for three case studies: token ring mapped on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture, and H.264 encoder running on the Diopsis R2DT architecture.

The simulation of the virtual architecture model allowed to verify the final code of the application tasks and the partitioning of the application. It also gave important statistics regarding the communication requirements. These include the total number of bytes exchanged between the subsystems during the execution of the application, the amount of data passing through the interconnect component (bus, NoC), and the buffer size requirements in the worst-case scenario for the storage resources in order to support the communication mapping.

Chapter 5
Transaction-Accurate Architecture Design

Abstract This chapter details the transaction-accurate architecture design. The transaction-accurate architecture design consists of integrating the OS and the communication software component with the application tasks code and adapting the software to specific communication synchronization protocol. The key contribution in this chapter represents the transaction-accurate architecture definition, organization, and design, using SystemC, for the token ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture and the H.264 encoder running on the Diopsis R2DT architecture. The simulation of the transaction-accurate architecture model allows validating the execution of the application tasks code upon an OS and early performance validation of the communication mapping scheme. Different interconnect components, communication mapping schemes, and IP cores positioning over the interconnect component are explored in order to analyze the performances of the various communication paths.

5.1 Introduction

The transaction-accurate architecture design consists of software adaptation to specific communication protocol implementation. At this phase, aspects related to the communication protocol are detailed, for example, the synchronization mechanism between the different processors running in parallel becomes explicit. The software code is adapted to the synchronization method, such as events or semaphores. The adaptation is performed through an integration of the tasks codes with the OS and the communication components of the software stack. The result of the transaction-accurate architecture design represents the transaction-accurate architecture model.

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_5,

5.1.1 Definition of the Transaction-Accurate Architecture

The third abstraction level of the hardware–software architecture is called transaction-accurate architecture level (TA). The transaction-accurate architecture details the local architecture of each subsystem and makes explicit the communication protocol. On the software side, the tasks code is integrated with an operating system and communication library to form the software stack. Each processor subsystem executes a software stack. The transaction-accurate architecture model may be manually coded or automatically generated by different tools.

The objectives of the transaction-accurate architecture are as follows:

– Early verification of the tasks code execution upon an operating system
– Early performance validation of the communication mapping scheme

The transaction-accurate architecture is composed of processor and hardware subsystems that are interconnected using an explicit interconnection component, such as bus or NoC. The processor subsystems include the local components of the subsystem, such as local memories, peripherals, and network interfaces, and an abstract model of the processor cores.

Figure 5.1 illustrates a global view of the transaction-accurate architecture, composed of two abstract processor subsystems, one memory hardware subsystem, and the network component. The left part of the figure corresponds to the hardware architecture, while the right part represents the software stack at the transaction-accurate architecture level running on one of the processor subsystems.

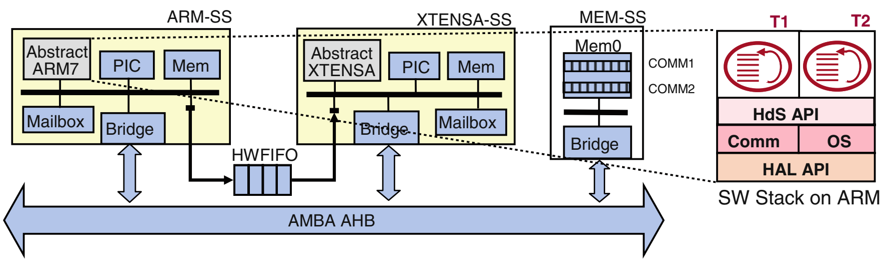


Fig. 5.1 Global view of the transaction-accurate architecture

5.1.2 Global Organization of the Transaction-Accurate Architecture

The transaction-accurate architecture model is a hierarchical model. The transaction-accurate architecture is composed of software and hardware subsystems that are interconnected using an explicit network component, e.g., bus, NoC, or dedicated hardware components like the hardware FIFO.

The software subsystem represents the processor subsystem. The hardware subsystem represents a memory subsystem or a dedicated hardware subsystem that accelerates the computation of specific application functions.

Each subsystem integrates local components that are interconnected using a local and simple bus. Usually the processor subsystems are made of one or more abstract computation models of the processor cores, local memories such as program code memory, data memory, or dedicated registers, network interfaces for the connection with the external world, and other processor-specific peripherals. The selection of these components relies on the target architecture and the software requirements at this level.

Each abstract processor model executes a specific software stack made of the tasks code, operating system, and communication library. The software stack uses hardware abstraction layer primitives (HAL APIs) for the interaction with the hardware part of the system. In fact, the abstract processor with the implementation of the HAL APIs represents the hardware–software interface.

At the transaction-accurate architecture level, the intra-subsystem communication units become communication channels implemented by the communication and operating system components of the software stack. Therefore, the communication between the tasks running on the same processor is managed totally by the OS and the communication software libraries.

The inter-subsystem communication units are mapped on full end-to-end communication paths through the architecture. Hence, the communication protocol and the synchronization between the processors become explicit. The different communication paths are characterized by different performance indicators, such as throughput of the buses, delay of the communication path, or overhead of the HdS layer (device drivers, resource sharing mechanism).

The adopted communication path and the topology of the network infrastructure are implemented according to the annotation of the system architecture model and the performance estimation through the simulation of the virtual architecture model.

Example 23. Transaction-accurate architecture for the token ring application Figure 5.1 shows a conceptual representation of the transaction-accurate architecture for the token ring application mapped on the 1AX architecture.

Figure 5.1 illustrates that for the token ring application running on the 1AX architecture, the transaction-accurate architecture contains two processor subsystems, corresponding to the ARM, respectively, XTENSA processors and the global memory subsystem. All these subsystems are interconnected by an explicit AMBA bus.

The ARM-SS processor subsystem includes an abstract ARM module, local memory, programmable interrupt controller (PIC), mailbox for the communication synchronization, and bridge for the interface to the AMBA bus, all interconnected through a local bus.

The local architecture of the XTENSA-SS subsystem is similar to the ARM-SS subsystem, but only it includes an abstract model for the XTENSA processor instead of the ARM7 processor. The global memory subsystem includes the global memory and the bridge for the connection with the global bus.

The communication through a FIFO between the tasks T1 and T2 mapped on the ARM-SS is implemented by the software components of the ARM software stack.

At the transaction-accurate architecture level, the inter-subsystem communication units COMM1 and COMM2 are mapped on full communication path. Therefore, a data sent by the ARM and received by the XTENSA processor using as storage buffer the global memory follows the data path illustrated below:

ARM -> BUS_ARMSS -> BRIDGE_ARMSS -> AMBA -> BRIDGE_MEMSS -> BUS_MEMSS -> MEM -> BUS_MEMSS -> BRIGDE_MEMSS -> AMBA -> BRIDGE_XTENSASS -> BUS_XTENSASS -> XTENSA
where

– *BUS_ARMSS* represents the local bus of the ARM-SS
– *BRIDGE_ARMSS* is the bridge of the ARM-SS
– *BUS_MEMSS* is the local bus of the MEM-SS
– *BRIDGE_MEMSS* is the interface of the global memory to the AMBA bus
– *BUS_XTENSASS* specifies the local bus of the XTENSA-SS
– *BRIDGE_XTENSASS* represents the bridge inside the XTENSA-SS.

This kind of data transfer requires synchronization mechanism between the two processors using the mailbox components. Thus, when the data to transmit is stored in the global memory, the ARM sends an event to the mailbox of the XTENSA to notify that there is available data. After checking the appropriate register status of the mailbox, the XTENSA processor may transfer the data from the global memory.

Other path of communication between the processors offered by the architecture involves the following route:

ARM -> BUS_ARMSS -> HWFIFO -> BUS_XTENSASS -> XTENSA

The communication through the hardware FIFO does not require explicit synchronization because the hardware resource manages also the synchronization between the processors.

The transaction-accurate architecture model may be represented using different design languages, such as SystemC [62] or SpecC [54]. The following paragraphs will present the transaction-accurate architecture using SystemC as design language.

5.2 Basic Components of the Transaction-Accurate Architecture Model

The basic components of the transaction-accurate architecture model are the software and hardware components. The software components consist of the tasks code, operating system, communication library, and HAL APIs, while the hardware components represent the detailed subsystems and explicit communication network.

5.2.1 Software Components

At the transaction-accurate architecture level, a software stack is build for each processor subsystem. This software stack is comprised of the previously generated tasks code enriched with an OS and communication library (Fig. 5.2). The HdS software represents the assembly of the OS, communication library, and HAL APIs. The HdS refines the communication APIs (HdS APIs) to custom hardware-specific low-level APIs (HAL APIs) and is responsible for the tasks and hardware resources management. The HAL APIs abstract the underlying hardware architecture. Their implementation is not yet defined for the target processor, allowing to keep the software code still processor independent. Based on the OS and communication libraries, the proposed approach sets aside flexible building and configuration of the software stack. Therefore, it allows easy customization for specific architectures and/or applications. At this level, the data transfers use explicit addresses, e.g., *read_mem(addr, dst, size)/ write_mem(addr, src, size)*.

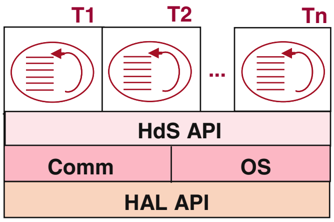


Fig. 5.2 Software components of the transaction-accurate architecture

Example 24. Software components for the token ring application at the transaction-accurate architecture level For the token ring application, a software stack is executed by each processor (ARM7 and XTENSA). The software stack running on the ARM7 is made of two application tasks code (T1 and T2), OS, and communication library. The software stack running on the XTENSA is made of the task code of T3, OS for the interrupt management, and communication software component. For both processors, the software stack has the same OS running, namely DwarfOS, the same communication library that implements the primitives *send_data(...)/ recv_data(...)*, and are based on the same HAL APIs (*read_mem(...)/ write_mem(...),ctx_swich(...)*).

5.2.2 Hardware Components

The hardware architecture at the transaction-accurate level represents a more detailed platform than the virtual architecture level. It includes the components explicitly used by the HAL APIs. The different subsystems of the architecture are detailed with explicit peripherals and abstract computation model for the processor cores. Design decisions such as subsystems positioning over the global interconnect component, NoC size definition, NoC topology, NoC routing algorithm, and communication buffer size are implemented at the transaction-accurate architecture level.

Example 25. Hardware components for the token ring application at the transaction-accurate architecture level For the token ring application, the hardware platform has a detailed local architecture for each subsystem (Fig. 5.3). Thus, the ARM-SS and XTENSA-SS contain an abstract ARM, respectively, XTENSA processor, a local memory, an interrupt controller, a local bus, and a bridge for the interface with the AMBA. The global memory subsystem contains the global memory and the bridge for the connection to the AMBA. The hardware FIFO is connected directly to the local bus of each processor subsystem.

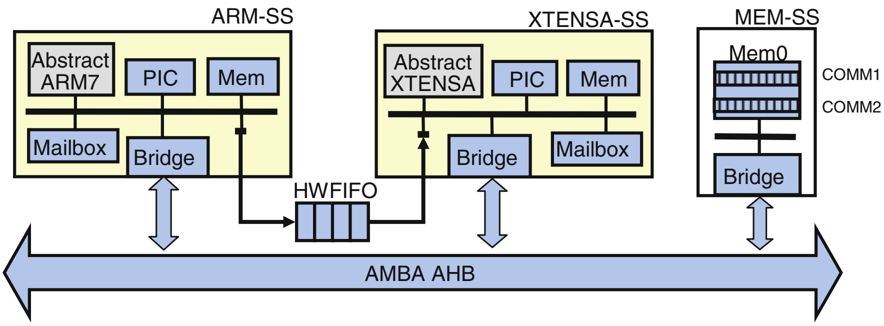


Fig. 5.3 Hardware components of the transaction-accurate architecture

5.3 Modeling Transaction-Accurate Architecture in SystemC

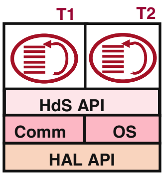
The transaction-accurate architecture model is described using SystemC TLM language and is designed according to the annotated architecture parameters of the initial system architecture model and the results of the virtual architecture model simulation.

5.3.1 Software at Transaction-Accurate Architecture Level

The software design at the transaction-accurate architecture level consists of integration of the tasks code with an OS and a communication implementation for each processor subsystem. In the following examples, the considered operating system is called DwarfOS, a tiny operating system which supports a set of basic services, such as interrupts management, FIFO software communication protocol, a cooperative scheduling policy based on static priority, and application tasks initialization [63, 126]. The communication primitives are based on blocking message-passing interface semantic. The synchronization is made using events. At this level, the generated tasks are dynamically scheduled by the OS scheduler according to the availability of data for read operations or the availability of space for write operations.

The tasks C code remains unchanged from the virtual architecture level and it uses HdS APIs such as *send_data(...)/recv_data(...)*. Compared with the virtual architecture, the implementation of these APIs is not anymore handled by the SystemC architecture. The implementation relies on the OS and communication libraries. Hence, the tasks are blocked on communication and scheduled by the OS scheduler and not by the SystemC scheduler as at virtual architecture level.

The OS and communication components make use of HAL APIs. At this level, the implementation of the HAL APIs is not yet defined for the target processors. Therefore, the software code is still processor independent at the transaction-accurate architecture level, but it is adapted to specific hardware communication implementation such as synchronization. Figure 5.4 shows a part of the software code at the transaction-accurate architecture level.



main

```
extern void task_T1 ();
extern void task_T2 ();

void __start (void) {…
    create_task (task_T1);
    create_task (task_T2);
    …
}
```

Communication SW

```
void recv(ch, dst, size) {   // HdS API
   switch (ch.protocol){
   case FIFO:
        if  (ch.state==EMPTY)
          __schedule();
        …
        break;
    } …
```

C code of Task_T2

```
void task_T2( ) {
    while (1) {
        recv (CH1, B, 10);
        …
        send (CH2, C, 20);
       }
}
```

OS

```
void __schedule(void) {
  int old_tid = cur_tid;
  cur_tid = new_tid;
  __cxt_switch( old_tid.cxt, cur_tid.cxt ); //HAL API
  …
}
```

Fig. 5.4 Software at the transaction-accurate architecture level

The HAL APIs, i.e., *__ctx_switch(...)* gives to the operating system, communication, and application software an abstraction of the underlying architecture. Furthermore, the HAL APIs ease OS porting on a new hardware architecture.

There are different categories of HAL APIs [174]:

– Kernel HAL APIs, such as task context management APIs (e.g., context creation, deletion or context switch APIs, task initialization), stack pointer and program counter management APIs (*get/set_IP()*, *get/set_SP()*), or processor mode change APIs (*enable_kernel/ user_mode()*)

– Interrupt management APIs, e.g., APIs which enable/disable interrupt request from an interrupt source (*vector_enable/disable(vector_id)*), configure interrupt vector (*vector_configure(vector_id, level, up)*), mask/unmask interrupt for a processor (*interrupt_enable/disable()*), implement the interrupt service routines (*interrupt_attach/ detach(vector_id, isr)*) or HAL APIs that acknowledge to the interrupt source that the interrupt request has been processed (*clear_interrupt(vector_id)*)
– I/O HAL APIs, which configure the I/O devices and allow their access. For example, to configure an MMU device, the following I/O HAL APIs may be required: APIs for page management (*enable/disable_paging()*), address translation (*virtual_to_physical()*), TLB (translation lookaside buffer) management, such as set a TLB entry (*TLB_add()*) or get TLB entry virtual/physical page frame (*get_TLB_entry()*). Other I/O HAL API examples can be considered the APIs for the cache memory management, such as *Instruction/Data_Cache_Enable/Disable()*
– Resource management APIs, such as APIs for power management Power management (check battery status, set CPU clock frequency) or APIs to configure the timer (*set/reset_timer()*, *wait_cpu_cyle()*)
– Design time HAL APIs, which facilitate the software design process, more precisely the simulation. Example of such kind of API is the *consume_cpu_cyle()* to simulate the advance of the software execution time.

Example 26. Software code for the token ring application at the transaction-accurate architecture level Figure 5.5 illustrates an example of software code for the token ring at the transaction-accurate architecture level.

Figure 5.5 shows the main file. The main file contains the function "thread_main" which represents the first function executed on the processor after boot. The main file is responsible to initialize the application tasks and the software communication channels. It includes the OS-dependent header files, it declares the software FIFO communication channels, it attaches the interrupt service routines to the interrupt numbers, and it initializes the tasks in the list of tasks ready for execution for the operating system. As illustrated in Fig. 5.5, for the token ring application, the initialization file of the ARM7 processor declares the two tasks running on the ARM7 and the software FIFO used for the communication between them. It also attaches the interrupt service routine of the mailbox to the interrupt number 0.

Figure 5.6 shows a fragment of code implementing the communication primitive *recv_data(...)*. If the protocol of the communication channel is based on a FIFO mechanism, the implementation checks the status of the FIFO. If the FIFO is empty, the scheduler of the OS is called (*__schedule(...)*).

The communication primitives access the logic ports of the tasks that are declared in the header files of each task. Figure 5.7 shows the header file of task T2 running on the ARM7 processor in case of the token ring application.

Task T2 has two logic ports:

```
#include <config.h>                    // OS dependent header files
#include <support/os_types.h>
#include <comm/os_comm.h>
#include <comm/event.h>
#include <stdio.h>

extern void Task1( );
extern void Task2( );

unsigned char SWFIFO_buf1[4];                  // software channels
unsigned char SWFIFO_stat_send1 = OS_EVENT;
unsigned char SWFIFO_stat_recv1 = OS_NO_EVENT;

void thread_main( ) {
   int id;

   vector_attach(UNIX_IRQ, 0, _mailbox_isr, NULL);
   vector_enable(UNIX_IRQ);

   id=thread_create(Task1,0);               // tasks initialization
   id=thread_create(Task2,0);               // for scheduling
   return;
}
```

Fig. 5.5 Initialization of the tasks running on ARM7

```
void recv_data (ch,dst,size){      //implementation of recv_data HdS API
…
  switch (ch.protocol){
          case FIFO:
            if  (ch.state == EMPTY)
                    __schedule();  // OS scheduler
…
```

Fig. 5.6 Implementation of *recv_data(...)* API

– One input port (*In1_Task2*) bonded to the software FIFO channel that connects task T1 and T2 and it was declared in the main file of the ARM7 processor as pointed up in Fig. 5.5.
– One output port (*Out1_Task2*) for the external communication with the task T3 running on the XTENSA processor.

The logic ports have type *port_t,* as illustrated in Fig. 5.8. The *port_t* represents the data structure which implements the logic port in case of the DwarfOS. It combines the following fields: communication protocol associated with the port, status of the local synchronization register, status of the remote synchronization register, destination buffer used to store the data to be exchanged, list of tasks that are wait-

```
#ifndef _Task2_H
#define _Task2_H

#include <support/os_types.h>
#include <comm/os_comm.h>
#include <comm/event.h>
#include <stdio.h>

extern unsigned char SWFIFO_buf1[4];          //software fifo channel
extern unsigned char SWFIFO_stat_send1;       //status of sender
extern unsigned char SWFIFO_stat_recv1;       // status of receiver

extern port_t Out1_Task1;

port_t In1_Task2 = {OS_SWFIFO_PROTOCOL,       // SOFTWARE FIFO protocol
                    &SWFIFO_stat_recv1,       // local synchronization
                    &Out1_Task1,              // remote port
                    SWFIFO_buf1,              // buffer address
                    NULL,
                    OS_DEFAULT};

port_t Out1_Task2 = {OS_GFIFO_PROTOCOL,       // GLOBAL FIFO protocol
                    (void*)0x300808,          // mailbox local register
                    (void*)0x700808,          // mailbox remote register
                    (void*)0x40500000,        // buffer address
                    NULL,
                    OS_DEFAULT};
```

Fig. 5.7 Example of task header file

Fig. 5.8 Data structure of tasks' ports

```
typedef struct {
            protocol_t protocol;
            void *l_status;
            void *r_port;
            void *d_buffer;

            thread_t *requesting_thread;
            unsigned char specific;
} port_t;
```

ing for the port to acquire a synchronization event, and a specific field which stores special protocol characteristics.

The input port of task T2 is characterized by a software FIFO protocol and has the synchronization and buffer associated with the software FIFO channel. The output port of task T2 notes a global FIFO protocol with the communication buffer mapped onto the external memory at the address 0×40500000 and the synchronization making use of the registers of the local and remote mailbox corresponding to the communication channel. The local mailbox represents the mailbox corresponding to the ARM processor accessed at address 0×300808. The remote mailbox stands for the mailbox of the XTENSA-SS with address 0×700808.

```
void __schedule (void){
   int old_tid = cur_tid;
   cur_tid = get_new_tid();         //get new task ready for execution

   __ctx_switch (old_tid,cur_tid);  //context switch HAL API
   ...
```

Fig. 5.9 Implementation of the *__schedule()* service of OS

Figure 5.9 shows a portion of the OS scheduler implementation. The scheduler searches for a new task in status ready for execution. If there is a new ready task, the scheduler performs a context switch, by calling the HAL API *__cxt_switch(...)*. During the context switch, the OS saves the status and registers (program counter, stack pointer, etc.) of the processor running the current task and loads those of the new task.

5.3.2 Hardware at Transaction-Accurate Architecture Level

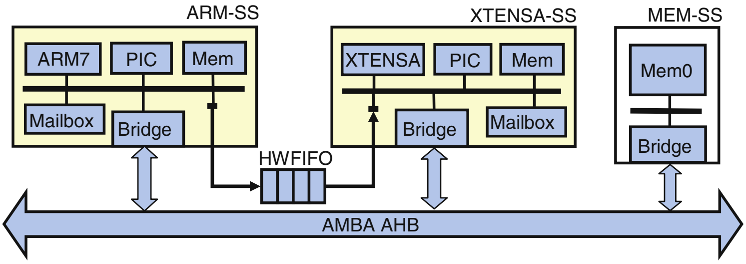
The hardware at the transaction-accurate architecture level consists of a set of hardware and software subsystems interconnected using an explicit communication network. The hardware architecture implements the communication protocol, including buffer mapping, synchronization mechanism used by the processors, and the entire communication path for inter-subsystem communication.

The different subsystems represent SystemC modules (SC_MODULE) which include the local components. A top module includes the declaration, instantiation, interconnection, and address space allocation of these subsystems. Each subsystem incorporates the local hardware modules. The local components are also SystemC modules.

The transaction-accurate architecture makes use of a library of transaction-accurate components. This library implements parametric hardware components such as mailbox, bridge, network interface, interrupt controller, interrupt signals, buses, and abstract execution model for distinct types of processor.

Example 27. Hardware code for the token ring application at the transaction-accurate architecture level Figure 5.10 details the top module for the token ring application running on the 1AX architecture.

The top module is an SC_MODULE which includes the declaration and the instantiation of the ARM-SS (*varm7-ss* in Fig. 5.10), XTENSA-SS (*vxtensa_ss*), AMBA bus (*vAMBA*), global memory subsystem MEM-SS (*vgmem_ss*), and hardware fifo (*vhwfifo*). It also interconnects these different subsystems by linking the bridges of each subsystem to the AMBA bus. A 4 MB address space is allocated to each processor subsystem. Thus, the ARM-SS has the address space 0×800000–0×BFFFFF and the XTENSA-SS has the address space 0×400000–0×7FFFFF. The global memory is identified between addresses 0×40000000–0×40FFFFFF.




```
#include "XTENSA_SS.h"
#include "ARM7_SS.h"
#include "AMBA.h"
#include "MEM_SS.h"
#include "HWFIFO.h"

SC_MODULE (TOP)
{
 public:
  AMBA *vAMBA;
  MEM_SS *vgmem_ss;
  XTENSA_SS *vxtensa_ss;
  ARM7_SS *varm7_ss;
  HWFIFO *vhwfifo;

  SC_CTOR(TOP)
  {                                                      //AMBA BUS
    vAMBA = new AMBA("AMBA");

                                                         //MEMORY SUBSYSTEM
    vgmem_ss = new MEM_SS("MEM",0x1000000);

    vgmem_ss->bridge->port(*vAMBA);
    vgmem_ss->bridge->port.set_map(0x40000000,0x40FFFFFF);

                                                         //XTENSA SUBSYSTEM
    vxtensa_ss = new XTENSA_SS("XTENSA_SS","../sw/XTENSA/XTENSA.bin");
    vxtensa_ss->bridge->port(*vAMBA);
    vxtensa_ss->bridge->port.set_map(0x400000,0x7FFFFF);

                                                        //ARM7 SUBSYSTEM
    varm7_ss = new ARM7_SS("ARM7_SS","../sw/ARM7/ARM7.bin");
    varm7_ss->bridge->port(*vAMBA);
    varm7_ss->bridge->port.set_map(0x800000,0xBFFFFF);

    vhwfifo = new HWFIFO("HWFIFO", 0x200, 1, 1);        //HWFIFO
    vhwfifo->inport(1, varm7_ss->bus->lport);
    vhwfifo->outport(1, vxtensa_ss->bus->lport);
  }
};
```

Fig. 5.10 SystemC code for the top module

Figure 5.11 shows the SystemC module of the ARM7 subsystem of the 1AX architecture.

The ARM7 subsystem includes a local bus (*sys_bus*), an abstract execution model of the processor core (*ArmUnixCore*), a local memory (*mem*), a bridge

```
#include "ARM7_SS.h"
extern int debug_flag;

ARM7_SS::ARM7_SS(sc_module_name name, char *bin)          // ARM7-SS
:sc_module(name)
{
    sys_bus = new TlmBus("sys_bus");                      // local bus

    core = new ArmUnixCore("ARM7Core",bin,debug_flag);    // abstract ARM7 core
    core->rw_port(*sys_bus);

    mem = new Sram("mem0",0x300000);                      // local memory
    mem->port(*sys_bus);
    mem->port.set_map(0x0,0x2FFFFF);

    bridge = new AhbIf("bridge");                         // bridge
    bridge->master(*sys_bus);
    bridge->slave(*sys_bus);
    bridge->slave.set_map(0x400000,0x7fffffff);

    pic = new Pic<1>("pic",0x20);                         // PIC
    pic->port(*sys_bus);
    pic->port.set_map(0x300000,0x30001f);

    sync = new Sync("sync",0x400);                        // mailbox
    sync->port(*sys_bus);
    sync->port.set_map(0x300800,0x300bff);

    TlmIntrSig *sig_sync = new TlmIntrSig("sig_sync");   //interrupt signals
    sync->intr(*sig_sync);
    pic->in_irq[0](*sig_sync);
    s1 = new TlmIntrSig("sig_intr1");
    s2 = new TlmIntrSig("sig_intr2");
    pic->out_fiq(*s1);
    pic->out_irq(*s2);
    core->nIrqPort(*s2);
}
```

Fig. 5.11 SystemC code for the ARM7-SS module

(*bridge*) for the connection to the AMBA bus, a programmable interrupt controller (PIC) (*pic*), the mailbox synchronization component (*sync*), and some interrupt signals (*sign_sync*, *s1*, and *s2*) . The local peripherals have associated address space. Thus, the local memory is addressable between addresses 0×0–$0\times2FFFFF$, the PIC between addresses 0×300000–$0\times30001F$, and the mailbox between addresses 0×300800–$0\times300BFF$. Each processor subsystem has the local address space between 0×0 and 0×400000. The accesses to addresses higher than 0×400000 will be forwarded by the local bus to the bridge for external access through the AMBA bus.

As illustrated in Fig. 5.12, the transaction-accurate architecture of the 1AX architecture contains a global clock used by all the processors. This clock has a period of time 1 unit, where a time unit represents1 ns.

```
sc_clock SystemClock("SystemClock", 1, SC_NS);    //SYSTEM SYSTEMC CLOCK
```

Fig. 5.12 SystemC clock

5.3.3 Hardware–Software Interface at Transaction-Accurate Architecture Level

The hardware–software interface at the transaction-accurate architecture level is represented by the abstract model of each processor core and the implementation of the HAL APIs. This is responsible to guarantee the software access to the hardware and implements the interaction between hardware and software.

The abstract model of the processor defines an execution environment of the software stack [138]. This is implemented as a SystemC module which interacts with the software. The abstract processor is modeled as a bus functional model, which allows operations onto the local bus, such as read and write operations [142].

The implementation of the HAL APIs allows a simulation model of the OS and inter-processor communication on the host machine [11]. For example, the implementation of the HAL API *ctx_switch (old_tid, cur_tid)* to perform a context switch between two tasks relies on the APIs provided by the operating system running on the host machine (Windows, Linux, UNIX, etc.). Figure 5.13 exemplifies the implementation of the context switch on the host machine running Linux OS that uses *sigsetjmp* and *siglongjmp* APIs to save and switch the context of a task.

```
void __ctx_switch(int old_tid, int new_tid)
{
          sigjmp_buf old_buf, new_buf;

          old_buf = task[old_tid].buf;
          new_buf = task[new_tid].buf;

          if(!sigsetjmp(old_buf, 1))                        //LINUX APIs
                              siglongjmp(new_buf, 1);
}
```

Fig. 5.13 Implementation of the *__ctx_switch* HAL API

5.4 Execution Model of the Transaction-Accurate Architecture

The full hardware–software executable model is based on co-simulation between SystemC for the hardware components including the abstract processors and the native execution of the software stacks [110]. The main advantage of the native simulation is the simulation speed.

A native software simulation usually runs as a separate process on the host machine. In order to control and access the OS/HAL simulation model, the IPC (inter-process communication) is used. The SystemC simulation environment is used as co-simulation backplane for the software and hardware simulators. Thus, each software stack is a SystemC thread which creates a Linux process for the software execution. At the beginning of the simulation, the SystemC platform launches a GNU standard debugger (gdb) Linux process for each software stack in order to start its execution. The software stack interacts with the corresponding SystemC abstract processor module through the Linux IPC layer. The hardware–software interface uses Linux shared memory (IPC Linux *shm*) for the interaction, data, and synchronization exchange between the software and the hardware.

The abstract processor module represents a bus functional model, shortly BFM. The BFM has two sides: one is facing the native HAL simulation model (i.e., a Linux process) and the other is a pin-level interface of the processor. Figure 5.14 shows an example of BFM for a native OS/HAL simulation model. In this example, the native HAL simulation model uses the shared memory, signals, and semaphores as IPC mechanisms for the external access.

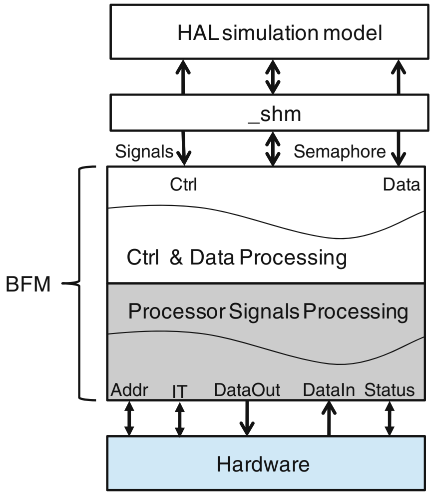


Fig. 5.14 Hardware–software co-simulation

The BFM is used to transform a functional memory access (e.g., read a data item from a specific physical address) to a sequence of memory accesses. Thus, the BFM transfers external access from the HAL simulation model to the hardware simulation engine (e.g., SystemC hardware simulation). This transfer is performed by polling the IPC interface of the HAL simulation model for read/write access. If there is a requested read/write data operation, the BFM transforms the access request into signal transitions on the processor's pin interface. Besides the data transfer, the BFM also transmits a processor interrupt to the HAL simulation model. When an interrupt arrives at the processor's interrupt pins, the BFM sends a signal (e.g., Linux signal) to the HAL simulation model, more precisely to the Linux process.

The simulation at the transaction-accurate architecture level allows validating the integration of the tasks code with the OS and the communication protocol and debug of the HdS access to the hardware resources (e.g., access to the AMBA bus, interrupt lines assignment, OS scheduling). On the software side, it makes possible the debug of the access of the OS functions to the hardware resources through the HAL APIs, e.g., *read(...)*/ *write(...)* from/to the memory, explicit synchronization using mailboxes, or the interrupt service routines. On the hardware side, it gives more precise statistics on the communication and computation performances, such as the number of exchanged data bytes during the application execution, network congestion, or estimation of the processors' cycles spent on communication.

Example 28. Execution model for the token ring application at the transaction-accurate architecture level Figure 5.15 shows the execution model of the software stacks running on the ARM7 and XTENSA processors in the case of the 1AX architecture. This represents a co-simulation between the gdb Linux processes of each software stack *gdb1* and *gdb2* (one gdb for each software stack) and one SystemC Linux process for the whole hardware platform simulation. The interface between the three Linux processes is performed using the Linux IPC shared memory.

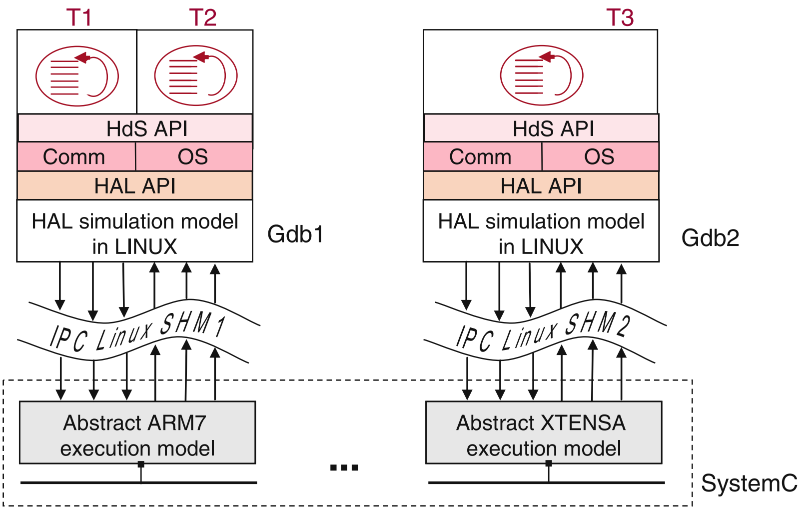


Fig. 5.15 Execution model of the software stacks running on the ARM7 and XTENSA processors

5.5 Design Space Exploration of Transaction-Accurate Architecture

5.5.1 Goal of Performance Evaluation

The goal of the performance evaluation at the transaction-accurate architecture level is to allow profiling the communication requirements and improve the overall performances of the system. The objective is to provide through simulation statistical information, such as utilization of the global interconnect component or the degree of contention in the network component, and validate the communication protocol and the execution of the tasks under the control of a dedicated operating system.

Based on the communication traffic resulted after the transaction-accurate architecture simulation, the designer can fix hardware and software architecture decisions. Examples of hardware architecture decisions are the entire end-to-end communication path used for the data exchange between the processors, the size of the NoC in number of routers, the positioning of the IP cores over the NoC, the final topology of the interconnect component, the routing algorithm used in a NoC, the buffer size inside the NoC routers or the communication protocol between the different subsystems fixing the mapping of the communication buffers onto the storage resources, and the synchronization mechanism. Examples of software architecture decisions are operating system used for the scheduling of the tasks running on the same processing units, implementation of the communication primitives, and synchronization mechanism managed by software.

These different decisions influence the overall execution time of the system, cost, and power consumption. Therefore, good decisions are required to be able to control the MPSoC design process.

5.5.2 Architecture/Application Parameters

The transaction-accurate architecture validates some hardware and software architecture characteristics specified at the system architecture level, such as the following:

– Integration of the tasks code with the OS and communication libraries
– Implementation of the communication protocol: buffers mapping, synchronization mechanism, and end-to-end data path between the processors
– Adaptation of the software to specific hardware communication implementation
– Type of the scheduling algorithm for the tasks
– Type of global interconnection algorithm with its configuration parameters such as topology, buffer size, routing algorithm, arbitration algorithm

The transaction-accurate architecture still keeps the implementation of the communication protocol independent of the type of processor cores. Therefore, the *CPUCoreType* represents an architecture parameter that will be considered only at the next abstraction level, the virtual prototype level. This will determine the adaptation of the software to a particular CPU through the explicit implementation of the low-level processor-specific HAL software layer.

5.5.3 Performance Measurements

At the transaction-accurate architecture level, the performance measurement consists of profiling the interconnect component and the communication and computation requirements for each processor.

Using annotation of the transaction-accurate architecture model with adequate execution delays, the simulation at this level can estimate the total clock cycles spent on communication or computation by each processor. The achieved precision can be cycle accurate only for the inter-subsystem communication, since all the hardware components of the communication path are explicit. The accuracy of the software execution is transaction level.

On the hardware side, the transaction-accurate architecture may give more precise statistics on the communication architecture such as the number of conflicts on the shared global bus due to the simultaneous access requests in the case of a bus-based architecture topology. For a NoC-based architecture topology, useful information deduced during the simulation are related to the amount of NoC congestion, number of routing requests, number of transmitted packets, the average amount of transmitted bytes per packet, or the number of times some routers failed to transmit the packet due to conflicts. For both topologies (bus and NoC), the transaction-accurate architecture simulation allows extracting the total amount of transmitted bytes through the global interconnect component and the amount of data transferred between the different processors.

Example 29. Performance measurements for the token ring application at the transaction-accurate architecture level For example, the total simulation time of the token ring application was 12s to run the whole application and the bus was required 108 times to transfer data. But in this example, the model is not annotated with accurate information required for an accurate estimation due to operating system and communication overhead.

5.5.4 Design Space Exploration

At the transaction-accurate architecture level, the design space exploration consists of communication mapping exploration. The designer can experiment different communication mapping schemes, different communication protocols, and diverse global interconnect components in distinct configurations. For example, the designer may adopt a bus such as STBus or AMBA bus or a NoC such as Hermes or STNoC. Moreover, the NoC may support different topologies (mesh, torus, hypercube, ring, tree), the routers may be positioned in different dimensions (2D, 3D), the number of routers is configurable, and the IP cores may be located through different access points to the NoC. Thus, the NoC offers flexibility and scalability in terms of number of routers, number of network interfaces, and interconnected IP cores.

Example 30. Design space exploration for the token ring application at the transaction-accurate architecture level At this level, the designer can still map the communication buffers onto different storage resources provided by the architecture, such as the local memories of both ARM and XTENSA processors, or the shared global memory, or on the hardware FIFO in case of the 1AX architecture running the token ring application. These different communication mapping schemes involve different communication paths and synchronization mechanisms between the processors.

5.6 Application Examples at the Transaction-Accurate Architecture Level

The following paragraph presents the transaction-accurate architecture model for the two case studies: the Motion JPEG decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 encoder application running on the Diopsis R2DT architecture with Hermes NoC in torus and mesh topologies.

5.6.1 Motion JPEG Application on Diopsis RDT

The transaction-accurate architecture design consists of two steps: software and hardware design. The software design consists of linking the tasks code with an operating system and communication library. For the Motion JPEG application, in order to produce an executable software code, the tasks code is compiled with the DwarfOS operating system and the communication library that implements the *send_data(...)/recv_data(...)* communication primitives. The tasks are scheduled by the OS. The communication between the tasks of the same processor is implemented by the OS and communication library.

The hardware architecture of the Diopsis RDT tile contains the components that can be accessed by the HAL APIs (Fig. 5.16).

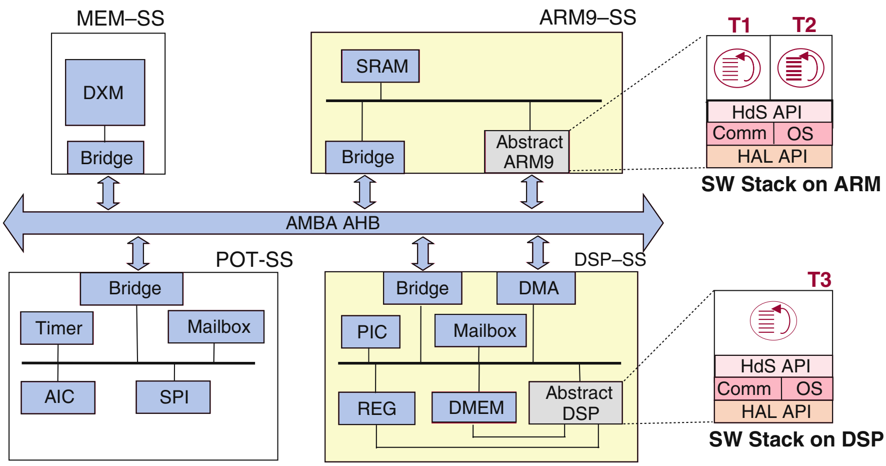


Fig. 5.16 Transaction-accurate architecture model of the Diopsis RDT architecture running motion JPEG decoder application

The ARM subsystem includes the abstract processor core, local data memory (SRAM), local bus, and bridge for the connection with the AMBA bus. The DSP subsystem includes the DSP core, data memory (DMEM), registers (REG), DMA, interrupt controller (PIC), mailbox, local bus, and the bridge for external connection. The POT includes the system peripherals of the RISC processor, e.g., timer, interrupts controller (AIC), synchronization component (mailbox), but also the I/O components like the serial peripheral interface (SPI).

The AMBA bus implementation is based on the implementation at the virtual architecture level. The main components of the AMBA bus are illustrated in Fig. 5.17.

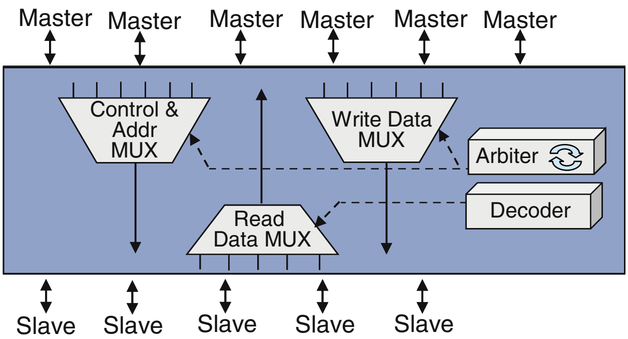


Fig. 5.17 AMBA bus at transaction-accurate architecture level

The synchronization between the different subsystems connected to the global bus is handled explicitly through the operating system and dedicated hardware components. The AMBA supports burst mode transfer at this level and fully models the arbitration strategy.

The assignment of addresses and mapping of the communication buffers into the memories with the corresponding interrupt mechanism used for synchronization is performed during the hardware platform design. The address space of the components is different from the virtual architecture platform, because the generated platform at the transaction-accurate level is more detailed and fully implements the communication protocol.

The full hardware–software executable model is based on co-simulation between SystemC for the hardware components including the abstract processors and native execution of the software stacks. Each software stack is a UNIX process created and launched at the beginning of the simulation by the SystemC platform, in order to start their execution. The software stack interacts with the corresponding SystemC abstract processor module through the Unix IPC layer. Besides the software debug, the execution model at this level also provides more precise idea on performances that allows some architecture experimentation, as detailed in the next section. The simulation of the 10 QVGA frames at the transaction-accurate level takes 5 min 10s.

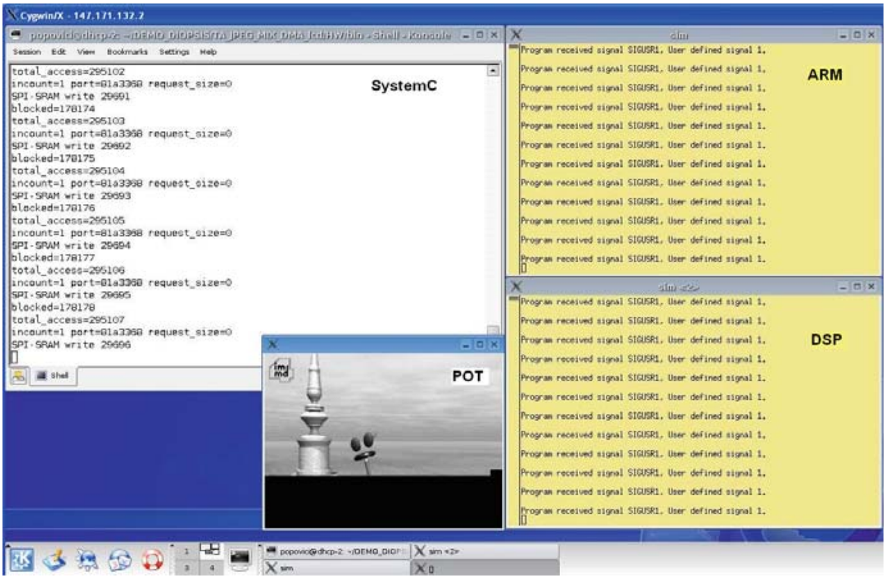


Fig. 5.18 MJPEG simulation screenshot

Figure 5.18 shows a screenshot taken during the simulation, which captures the execution of the two software stacks running on the ARM, respectively, DSP, and the SystemC simulation of the platform with the POT displaying the decoded image.

Using transaction-accurate simulation, in this book, three experiments are conducted with different communication schemes between the DSP and RISC. The results are summarized in Table 5.1. In the first scheme, the data exchange is made only via DXM. This generated 5,256,000 transactions to the DXM. The second communication scheme makes use of DXM and REG communication units between the processors and DMEM between the DSP and the POT. This generated 4,608,000 transactions to the DXM, 72,000 to the register, and 576,000 to the DMEM. The third case uses the SRAM as communication unit between the processors and DMEM between the DSP and POT and needs 4,680,000 transactions to

Table 5.1 Memory accesses

	Transactions (kB)					
Communication scheme	DXM	SRAM	REG	DMEM	Total cycles	–
DXM+DXM+DXM	5,256 k	0	0	0	8,856 k	100%
DXM+REG+DMEM	4,608 k	0	72 k	576 k	7,884 k	89%
SRAM+SRAM+DMEM	0	4,680 k	0	576 k	3,960 k	45%

the SRAM and 576,000 to the DMEM. One transaction to the memory means one read/write operation of one word (4 bytes) to the memory.

Starting from quantitative estimators provided by ATMEL Inc., the number of clock cycles, needed by the ARM and DSP to access data buffers of length N words located in different memories, can be estimated. The DMA engine of the DSP needs 14+(N–1) cycles for DXM read, 10+(N–1) for DXM write, 5+(N–1) for SRAM read, and 8+(N–1) for SRAM write. A data movement between REG and SRAM driven by the DSP core costs N/4 cycles plus a movement to/from the SRAM driven by the DMA engine. The ARM processor is not natively equipped with a DMA engine. The cost of an ARM isolated access is $11\times N$ for DXM read and $8\times N$ for DXM write. Forcing the compiler to use the assembler instruction which moves blocks of 8 registers, the cost of burst can be reduced to $11\times (N/8)+N$ for DXM read and $2\times N$ for DXM write. On the Diopsis tile, the ARM processor runs at a clock frequency which is double of the AMBA bus used as a unit of measure. This factor 2 can be taken into account in the estimate of time of the ARM access to the SRAM. The DSP data memory can be accessed by the ARM in $6\times(N/8)+N$ cycles for write and $8\times N$ cycles for read.

The performance estimation results are summarized in Table 5.1. The overall number of cycles required for the communication using AMBA burst mode is approximately 8,856 k when all the data transfer is made via DXM; 7,884 k in the second case using REG, DXM, and DMEM storage resources; and 3,960 k in the third case using the SRAM and DMEM local memories. Thus, if the software code makes use of the existing hardware resources, an improvement in communication performance can be obtained. This improvement corresponds to 11% in the second communication mapping case and 55% in the third case. The communication protocol is specified in the initial Simulink model by annotating the communication units.

5.6.2 H.264 Application on Diopsis R2DT

The transaction-accurate architecture of the Diopsis R2DT tile with Hermes NoC is illustrated in Fig. 5.19.

The tasks code is combined with the DwarfOS operating system and the implementation of the *send_data(...)/recv_data(...)* communication primitives to build each software stack running on the processors. The processors execute a single task on top of the operating system. The OS is required for the interrupt service routines and the application boot.

The hardware platform is comprised of the detailed three processor subsystems (ARM9-SS, DSP1-SS, and DSP2-SS), one global memory subsystem (MEM-SS), and the peripherals on tile subsystem (POT-SS). The different subsystems are interconnected through an explicit Hermes NoC available in torus and mesh topologies.

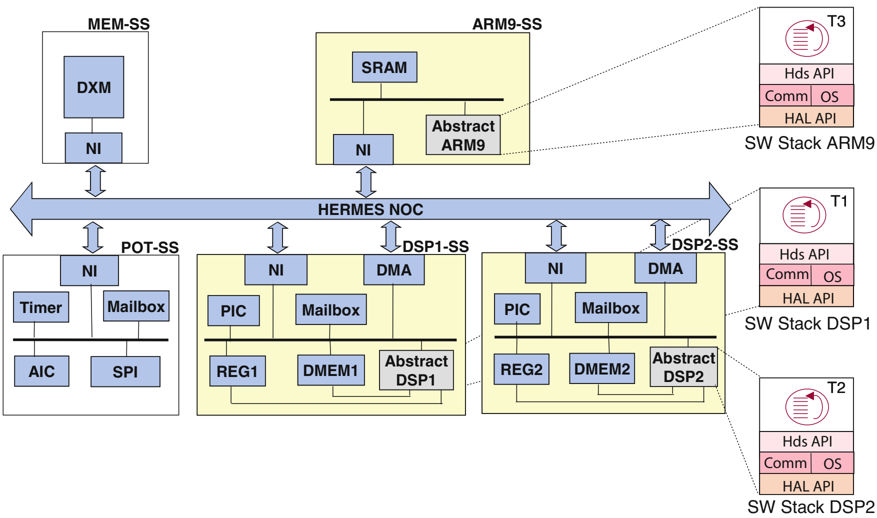


Fig. 5.19 Global view of the transaction-accurate architecture for Diopsis R2DT with Hermes NoC running H.264 encoder application

Figure 5.19 presents the transaction-accurate architecture of the Diopsis R2DT tile with NoC running the H.264 encoder application. The local architectures of each subsystem are detailed, including network interfaces, local bus, data memories and registers, abstract processor models, synchronization components, interrupt controller, or DMA engines.

The Hermes NoC at the transaction-accurate architecture adds more architectural details such as topology, routing algorithm, and router buffer size. The Hermes NoC model is comprised of the same basic elements as at the virtual architecture level: network interface, mapping table, and routers but with a more detailed implementation. Topology (e.g., mesh, torus), routing algorithm (e.g., pure XY, west first), arbiter algorithm (e.g., round robin, priority based), and buffer size (e.g., number of flits) can be varied. The packet structure in this model is comprised of destination address, size, and body fields, similar to the one assumed in the synthesizable NoC description. The Hermes NoC allows at the transaction-accurate architecture level extracting information from the system communication architecture like *(i)* number of routing requests; (*ii*) number of packets inserted into the NoC; (*iii*) amount of bytes exchanged; (*iv*) the average of bytes per packet; (*v*) the number of packets transmitted; and (*vi*) number of routing request failed due to NoC congestion.

At the transaction-accurate architecture level, the DMA components belonging to the DSP subsystems become explicit and have direct link to the interconnect component. Thus, the Hermes NoC for the Diopsis R2DT architecture requires seven access points: five for the different subsystems, as previously presented in the virtual architecture model, and two additional for the DMA components.

The different subsystems can be mapped over the NoC in different ways. The following paragraphs describe with details an example of IP cores mapping scheme. Thus, in a first scheme, the network interfaces connect the following IP cores to the NoC:

– The ARM9-SS is connected to the network interface with address 1×0.
– The network interface with address 2×1 connects the DSP1-SS.
– The network interface with address 1×1 connects the DMA of the DSP1-SS.
– The network interface with address 1×2 connects the DMA of the DSP2-SS.
– The network interface with address 2×2 connects the DSP2-SS to the NoC.
– The network interface corresponding to the MEM-SS has address 0×0.
– The network interface connecting the POT-SS has address 0×1.

The NoC was adopted in two topologies: mesh and torus. In both cases, the NoC has nine routers (3×3). Each router is connected to the corresponding network interface and the neighbor routers.

Figure 5.20 shows the NoC employing a 2D mesh topology, a pure XY routing algorithm, and a round-robin arbiter algorithm at each router and wormhole as packet-switching strategy.

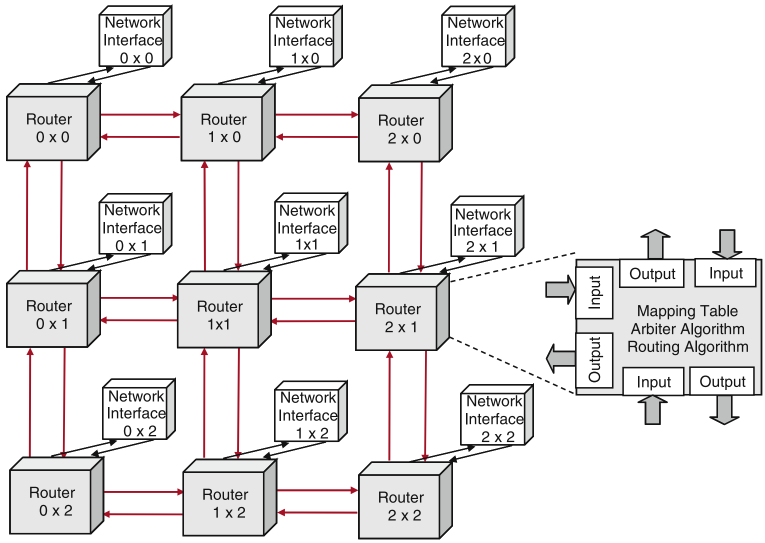


Fig. 5.20 Hermes NoC in mesh topology at transaction-accurate level

Table 5.2 shows the results captured during the transaction-accurate architecture mesh model simulation in case of the H.264 encoder application. The first and the second columns represent the correspondence between the different subsystems and the NoC access points. A routing request is performed at least once per packet

Table 5.2 Mesh NoC routing requests

IP core	NoC @	Total (%)	Local (%)	North (%)	South (%)	East (%)	West (%)
MEM-SS	0×0	20.00	6.39	6.18	0.00	7.43	0.00
POT-SS	0×1	20.63	7.22	3.19	7.22	2.99	0.00
	0×2	3.19	0.00	0.00	0.00	3.19	0.00
ARM9-SS	1×0	21.04	7.43	7.22	0.00	0.00	6.39
DSP1-SS (DMA)	1×1	10.21	0.00	0.00	0.00	2.99	7.22
DSP2-SS (DMA)	1×2	3.19	0.00	0.00	0.00	3.19	0.00
	2×0	6.18	0.00	0.00	0.00	0.00	6.18
DSP1-SS (NI)	2×1	9.17	2.99	0.00	6.18	0.00	0.00
DSP2-SS (NI)	2×2	6.39	3.19	0.00	3.19	0.00	0.00

per router that it will cross. Depending on the application, the NoC structure, routing algorithm, NoC congestion state, the routing request can occur as many times as needed inside a router. For the H.264 encoder simulation with 10-frame QCIF YUV 420 format, 96,618,508 routing requests were issued. The third column of Table 5.2 presents the percentage of the routing requests at each router, while the other columns detail this information related to the router port (local to the corresponding network interface, north, south, east, or west). These results were captured in the case of mapping all the communication buffers onto the external memory.

Figure 5.21 shows the amount of data that traverses each router in the mesh NoC for the H.264 encoder application by using external memory for the communication

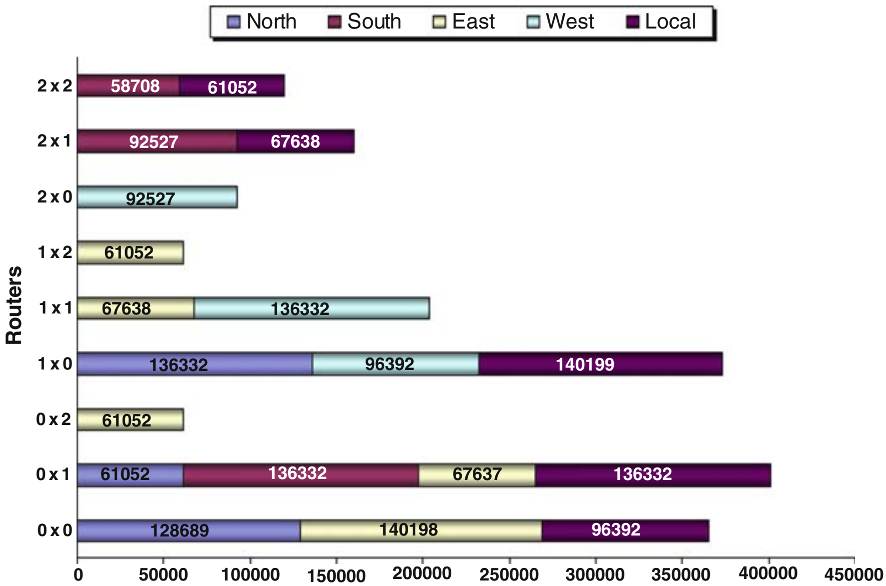


Fig. 5.21 Total kilobytes transmitted through the mesh

between the processors. The local port of each router inserts packets to the NoC, while the remaining ports transfer them inside the NoC. The value assigned to the local port of the router 0×0 (MEM SS) corresponds to response packets due to read requests or confirmation packets due to write requests. Block transfer operations (amount of operation that will be transferred in one packet) permit to optimize the amount of data exchanged inside the NoC by minimizing the amount of control data.

In the second topology, the adopted NoC was a 2D torus topology using a deadlock free version of the non-minimal west-first routing algorithm proposed by Glass and Ni [60]. Figure 5.22 presents the Hermes 3×3 torus NoC.

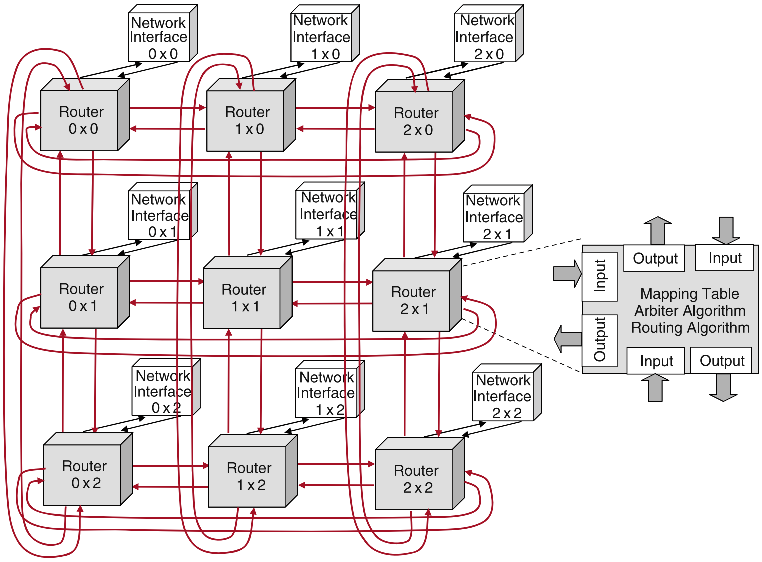


Fig. 5.22 Hermes NoC in torus topology at transaction-accurate level

The H.264 encoder simulation with 10-frame QCIF YUV 420 format using torus NoC topology involved approximately 78,217,542 routing requests, representing 19% of reduction when compared to the mesh NoC. This was possible because the 2D torus topology has the longest minimum paths that are only half of those in 2D meshes. Also, torus networks have better path diversity than meshes, which, if exploitable by the routing algorithm, leads to diminished network congestion, thus reducing routing requests.

Table 5.3 presents these results. The first columns represent the correspondence between the IP cores and the network interfaces, while the others show the distribution of the routing requests along the local, north, south, east, and west ports of each router. The results were captured in the case of mapping all the communication buffers onto the external memory.

Table 5.4 sums up the amount of data transferred through the torus NoC during the H.264 encoder simulation. The third column of the table represents the amount of data and control information exchanged (e.g., operation request, confirmation

Table 5.3 Torus NoC routing requests

IP core	NoC @	Total (%)	Local (%)	North (%)	South (%)	East (%)	West (%)
MEM-SS	0×0	25.67	8.90	4.28	4.34	8.14	0.00
POT-SS	0×1	20.00	7.86	0.00	7.86	0.00	4.28
	0×2	4.33	0.00	0.00	0.00	0.00	4.33
ARM9-SS	1×0	16.28	8.14	7.86	0.00	0.00	0.28
DSP1-SS (DMA)	1×1	7.86	0.00	0.00	0.00	0.00	7.86
DSP2-SS (DMA)	1×2	0.00	0.00	0.00	0.00	0.00	0.00
	2×0	8.62	0.00	0.00	0.00	8.62	0.00
DSP1-SS (NI)	2×1	8.57	4.28	0.00	4.28	0.00	0.00
DSP2-SS (NI)	2×2	8.68	4.34	4.34	0.00	0.00	0.00

Table 5.4 Torus NoC amount of transmitted data (bytes)

	NoC @	Local	North	South	East	West
MEM-SS	0×0	110,724,784	80,393,092	68,768,172	127,341,684	0
POT-SS	0×1	122,941,472	264	122,941,856	0	80,393,360
	0×2	0	0	0	0	68,768,436
ARM9-SS	1×0	127,342,228	1,229,941,340	132	0	4,399,692
DSP1-SS (DMA)	1×1	0	0	0	132	122,941,208
DSP2-SS (DMA)	1×2	0	0	0	132	0
	2×0	0	0	0	106,325,092	528
DSP1-SS (NI)	2×1	80,393,908	396	40,196,920	264	0
DSP2-SS (NI)	2×2	68,768,964	66,128,700	528	0	0

response). The other columns of the table show the amount of data transmitted per router port.

Figure 5.23 shows a screenshot captured during the simulation of the H.264 encoder running on the Diopsis R2DT architecture with torus NoC.

In order to analyze the communication performances, the AMBA bus is also experimented as global interconnect instead of the Hermes NoC. The average throughput of the interconnect component in order to execute the H.264 in real time (25 frames/s) was 235 MB/s for the NoC and 115 MB/s for the AMBA.

The NoC allows various mapping schemes of the IPs over the NoC with different impact on performances. In this book, two different mappings of the IP cores over the mesh and torus NoC are experimented: scheme A, detailed in the previous paragraphs and scheme B with the MEM-SS connected at network interface with address 1×1 (both x- and y-coordinates are 1). Figure 5.24 summarizes the correspondence between the network interface and the IP core in case of these two IP mapping schemes.

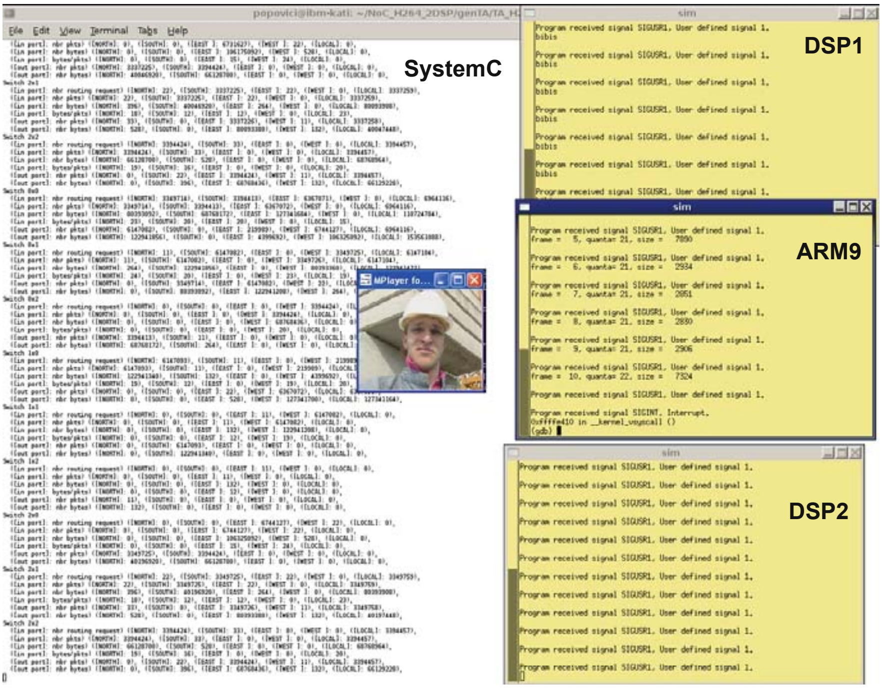


Fig. 5.23 Simulation screenshot of H.264 encoder application running on Diopsis R2DT with torus NoC

Scheme A

Y \ X	0	1	2
0	MEM-SS	POT-SS	–
1	ARM9-SS	DMA1	DMA2
2	–	DSP1–SS	DSP2-SS

Scheme B

Y \ X	0	1	2
0	DMA1	–	–
1	ARM9-SS	MEM-SS	DSP2-SS
2	POT-SS	DSP1-SS	DMA2

Fig. 5.24 IP core mapping schemes A and B over the NoC

Table 5.5 presents the results of the transaction-accurate simulation: estimated execution cycles of the H.264 encoder, the simulation time using the different interconnect components on a PC running at 1.73 GHz with 1 GB RAM, and the total routing requests for the NoC. These results were evaluated for the two considered IP mapping schemes shown in Fig. 5.24 (A and B) and for three communication buffer mapping schemes: *DXM+DXM+DXM*, *DMEM1+DMEM2+SRAM*, and *DMEM1+SRAM+DXM*. The AMBA had the best performance, as it implied the fewest clock cycles during the execution for all the communication mapping schemes. The mesh NoC attained the worse performance in case of mapping all the

Table 5.5 Execution and simulation times of the H.264 encoder for different interconnect, communication, and IP mappings

Communication mapping scheme	Interconnect	IPs mapping over NoC	Execution time at 100 MHz (ns)	Simulation time (min)	Execution cycles	Simulation cycles/second	NoC routing requests	Average interconnect latency (cycles/word)
DXM+DXM+DXM	Mesh	Scheme A	64,028,725	36 min	3,201,436	1,995	96,618,508	25
DXM+DXM+DXM	Torus	Scheme A	46,713,986	28 min 29s	2,335,699	1,990	18,115,966	16
DMEM1+DMEM2+SRAM	Mesh	Scheme A	28,573,705	12 min 54s	1,428,685	1,730	18,512,403	10
DMEM1+DMEM2+SRAM	Torus	Scheme A	26,193,039	12 min	1,309,652	1,593	15,213,557	9
DMEM1+SRAM+DXM	Mesh	Scheme A	26,233,039	14 min 55s	1,594,237	1,430	18,467,386	11
DMEM1+SRAM+DXM	Torus	Scheme B	26,193,040	14 min 48s	1,309,652	1,639	24,753,488	10
DXM+DXM+DXM	Mesh	Scheme B	35,070,577	18 min 34s	1,753,529	1,555	24,753,610	9
DXM+DXM+DXM	Torus	Scheme B	35,070,587	19 min 8s	1,753,529	1,527	14,479,723	9
DMEM1+DMEM2+SRAM	Mesh	Scheme B	31,964,760	17 min 8s	1,595,238	1,574	13,144,538	13
DMEM1+DMEM2+SRAM	Torus	Scheme B	31,924,752	16 min 14s	1,595,238	1,475	12,674,692	13
DMEM1+SRAM+DXM	Mesh	Scheme B	31,964,731	18 min 38s	1,598,237	1,466	13,118,044	15
DMEM1+SRAM+DXM	Torus	Scheme B	31,924,750	16 min 42s	1,596,238	1,819	78,217,542	14
DXM+DMX+DXM	AMBA	–	17,436,640	8 min 24s	871,832	1,846	–	9
DMEM1+DMEM2+SRAM	AMBA	–	17,435,445	7 min 18s	871,772	1,527	–	9
DMEM1+SRAM+DXM	AMBA	–	17,435,476	7 min 17s	871,774	1,482	–	9

communication buffers onto the *DXM* and similar performance with the torus in case of using local memories.

This is explained by the small numbers of subsystems interconnected through the NoC. In fact, NoCs are very efficient in architectures with more than 10 IP cores interconnected, while they can have a comparable performance results with the AMBA bus in less complex architectures. Between the NoCs, the torus has better path diversity than mesh. Thus, torus reduces network congestion and decreases the routing requests. Also, scheme A of IP cores mapping provided better results than scheme B for the *DMEM1+DMEM2+SRAM* buffer mapping. For the other buffer mappings the performance of scheme A was superior to scheme B. In fact, the ideal IP cores mapping scheme would have the communicating IPs separated by only one hope (number of intermediate routers) over the network to reduce latency.

Comparing with the virtual architecture, the transaction-accurate interconnects fully implement the bus, respectively, the NoC protocol. Thus it provides accurate characteristics. Therefore, the simulation of the transaction-accurate interconnects requires higher simulation time compared with the virtual architecture. But, during both design steps, the NoC needs more time for the application simulation than buses due to its high complexity.

5.7 State of the Art and Research Perspectives

5.7.1 State of the Art

Current literature offers large set of references dealing with transaction-accurate architecture design and software native execution using an abstract hardware platform.

ChronoSym [11] presents a fast and accurate SoC co-simulation that allows verification of the integration of the tasks code with the operating system. It is based on an OS simulation model and annotation of the software with execution delays. The abstract execution model of the processors in the transaction-accurate architecture presented in this book is similar to the timed bus functional model used in the ChronoSym approach, but it is not annotated for accurate estimation.

Reference [25] presents an abstract simulation model of the processor subsystem. In this work, the processor subsystem is not defined as a set of hardware components, but it is viewed from a software point of view. Thus, the processor subsystem is made of execution, access, and data unit elements to allow early validation of the MPSoC architecture and native time-accurate simulation of the software.

Reference [56], based on the work described in [23], resumes a hardware–software interface modeling approach in SystemC at the transaction-accurate architecture level. This work uses the concept of required and provided services in the modeling of the hardware–software interfaces. The hardware–software interface is assembled using software, hardware, and hybrid elements.

Reference [74] illustrates a configurable event-driven virtual processing unit (VPU) to capture timing behavior of multiprocessor multithreaded platforms through flexible timing annotation. The VPU enables investigation of the mapping of the application tasks with respect to time and space and early design space exploration.

Reference [138] deals with abstract modeling of the embedded processors using TLM. This work develops a high-level abstract processor model that allows fast simulation, acceptable accuracy in simulated timing, and exposing the structure of the software architecture (e.g., drivers and interrupts). This approach is similar to the abstract execution model of the processor belonging to the transaction-accurate architecture.

Reference [19] details the Synopsys System Studio design tool that allows a SoC design flow from system level to implementation by passing through several abstraction levels. One of the intermediate refinement steps corresponds to the development at the platform level, which represents a TLM platform of the hardware that allows starting the development of the software. The software development itself uses a specific development and simulation kernel such as RTLinux, together with an interface layer to the virtual processors on the platform.

References [67, 68] present a simulation model of μTRON-based RTOS kernels in SystemC. They developed a library of APIs that supports preemption, task priority assignment, or scheduling RTOS services by native execution and a SystemC wrapper to encapsulate the OS simulation model into the bus functional model (BFM) of the hardware platform. Their approach is similar to the presented approach, but they do not give details on the hardware side.

Reference [143] presents a communication design flow based on automatic TLM model generation. They allow generation and refinement of bus-based communication architectures, including bus bridges and transducers. But they do not address software code adaptation to specific communication protocol implementation, in order to optimize the overall communication performance.

Reference [77] proposes a hardware procedure call (HPC) protocol to abstract the platform-dependent details of the TLM communication between the different subsystems, by providing an additional layer for the software modeling on top of transaction-level models.

5.7.2 Research Perspectives

The most important research perspective regarding the transaction-accurate architecture design consists of annotating the software code with execution delays for accurate software performance estimation and annotating the hardware code for accurate communication architecture performance estimation. This could be managed by applying a similar approach with the timed bus functional model used in ChronoSym [11].

Other research perspective represents the automatic generation of the transaction-accurate architecture. The generation could be made possible by applying a service-based modeling of the hardware–software interface as described in [56]. The composition of the services easies the automatic generation tools to reduce design time. The generation can be performed from the system architecture or virtual architecture. Generation from the system architecture enables generation of different detail levels from the same specification (virtual architecture, transaction-accurate architecture, and virtual prototype). The generation from the virtual architecture enables gradual refinement of the hardware/software architecture based on the performance estimation performed at this level.

On another proposed research perspective refers to the design at the transaction-accurate architecture level of more complex multi-tile architectures such as Tile64 [157] or AM2000 [4] running massive parallel applications.

5.8 Conclusions

This chapter defined the transaction-accurate architecture design. It presented the software organization as final application tasks code running upon a real-time OS and the hardware organization in detailed subsystems interconnected through an explicit network component.

The transaction-accurate architecture design was performed using SystemC for three case studies: token ring mapped on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture, and H.264 encoder running on the Diopsis R2DT architecture.

The simulation of the transaction-accurate architecture model allowed to verify the integration of the final application tasks code with an OS and communication software adapted to the synchronization protocol. It also gave more precise information on the interconnect model. This includes the number of conflicts in the global bus, the amount of NoC congestion, the number of transmitted bytes through the bus or NoC, the number of routing requests, the number of times some routers failed to transmit the packet due to conflicts inside the NoC, or the average bytes per packet.

The transaction-accurate architecture design also allows exploration of different IP cores mapping over the NoC in order to analyze their impact on the overall performances.

Chapter 6
Virtual Prototype Design

Abstract This chapter details the virtual prototype design. The virtual prototype design consists of integrating the HAL implementation into the software stack and establishing the final memory mapping. The verification of the software is performed by using classical co-simulation with instruction set simulators (ISS). The key contribution in this chapter represents the virtual prototype definition, organization, and design using SystemC for the token ring application running on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture, and H.264 encoder running on the Diopsis R2DT architecture. The Motion JPEG application is executed using ISS on different types of single processor (ARM7, ARM9, and DSP) and the H.264 encoder is simulated using ISS running on both multi-processor architecture with three ARM7 processors and single processor (ARM7 and ARM9). The simulation of the virtual prototype model allows to verify the software binary and the memory mapping.

6.1 Introduction

The virtual prototype design consists of software adaptation to specific target processors and memory mapping. This includes the integration of the processor-dependent software code into the software stack, more precisely the HAL integration with the tasks code, OS, and communication software components. The result of the virtual prototype design represents the virtual prototype model.

6.1.1 Definition of the Virtual Prototype

The lowest MPSoC abstraction level is called virtual prototype (VP) . The software stack is fully explicit, including the HAL layer to access the hardware resources and it is detailed to ISA (instruction set architecture) level to be adapted for a specific processor. The hardware architecture incorporates an ISS for each processor to

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_6,

execute the final binary code. At the virtual prototype level the communication consists of physical I/Os, e.g., *load/store*.

According to [70], the virtual prototype has the following objectives:

– Measure system performance and analyze its bottlenecks
– Find out optimization points from the bottleneck analysis by using traces and profile data
– Allow full software stack and memory mapping validation before the real hardware is available
– Evaluate architectural decisions of both hardware and software sides

The virtual prototype is characterized by three issues: timing accuracy, simulation speed, and development time. The virtual platform has to be accurate enough to analyze system performance including hardware–software interaction, fast enough to execute the software and it has to be available earlier than the real-chip development. Unfortunately, these criteria are difficult to be accomplished simultaneously: accurate platforms usually require detailed information, thus they impose slow simulation speed and substantial time to develop.

The simulation at the virtual prototype level allows performance validation and it corresponds to classical hardware/software co-simulation models with instruction set simulators [134, 140] for the processors. The simulation performed at this level is cycle accurate. It allows validating the memory mapping of the target architecture and the final software code. It also provides precise performance information such as software execution time, computation load for the processors, the number of clock cycles spent on communication. The hardware platform includes all the hardware components such as cache memories or scratch pads.

Figure 6.1 illustrates a global view of the virtual prototype, composed of ISS for the processors and the other hardware components, such as local resources of the processor subsystems, memory subsystem, and the network component. The left part of the figure corresponds to the hardware architecture, while the right part represents the software stack at the virtual prototype level running on one of these processor subsystems.

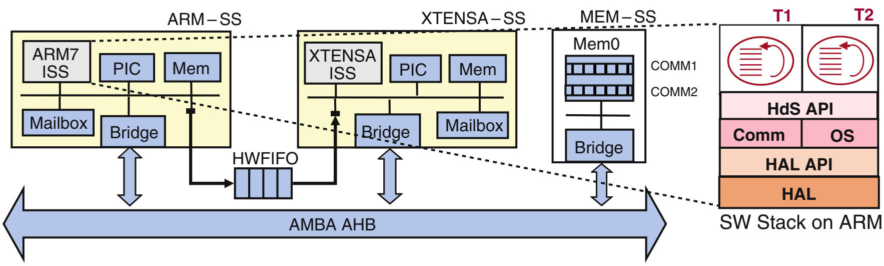


Fig. 6.1 Global view of the virtual prototype

6.1.2 Global Organization of the Virtual Prototype

The virtual prototype model is a hierarchical model. The virtual prototype is comprised of detailed software and hardware subsystems interconnected through a global interconnect component. The software subsystems incorporate an instruction set simulator (ISS) for each processor to execute the final binary code and cycle-accurate components for the rest of the architecture. The ISS is a software environment which can read microprocessor instructions and simulate their execution. Most of these tools can provide simulation results like values in memory and registers, as well as timing information (e.g., clock cycle statistics).

Example 31. Virtual prototype for the token ring application mapped on the 1AX architecture Figure 6.1 shows a conceptual representation of the virtual prototype for the token ring application mapped on the 1AX architecture. Figure 6.1 illustrates that for the token ring application running on the 1AX architecture, the virtual platform contains two processor subsystems, corresponding to the ARM, respectively, XTENSA processors and the global memory subsystem. All the subsystems are interconnected by an explicit AMBA bus. The processor subsystems encapsulate the ISS for the ARM7 processor, respectively, XTENSA processor ISS.

The software stack represents the final software code adapted to specific processor implementation. The communication consists of physical I/Os, e.g., *load/store.*

6.2 Basic Components of the Virtual Prototype Model

The basic components of the transaction-accurate architecture model are software and hardware components. The software components consist of the tasks code, operating system, communication library, and HAL, while the hardware components represent detailed subsystems with ISS for the processor.

6.2.1 Software Components

At the virtual prototype level, the software stack running on each processor is completely detailed and represents the final binary of the software. The binary image will run on the hardware simulation platform or on the physical architecture board if this is available.

As shown in Fig. 6.2, the software stack is comprised of all the software components: application tasks code, communication implementation, operating system, HAL, and the APIs to pass from one component to another. Thus, the software stack is fully explicit, including the HAL layer to access the hardware resources and it is detailed to ISA (instruction set architecture) level for a specific processor. The HAL represents a thin low software layer, totally dependent on the target processor core. The HAL allows the software to access and configure the hardware peripherals.

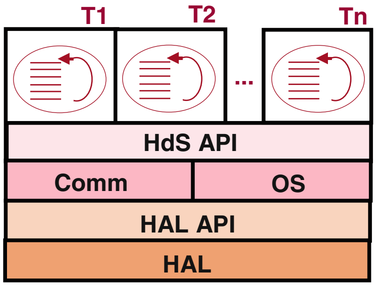


Fig. 6.2 Software components of the virtual prototype

Example 32. Software components for the token ring application at the virtual prototype level For the token ring application, both software stacks running on the ARM7, respectively, XTENSA processor are made of the application tasks code (T1 and T2 for the ARM7 processor and T3 for the XTENSA), the DwarfOS as operating system, communication library and the HAL-specific implementation for the ARM7, respectively, XTENSA processors. For both software stacks, the data and program code are mapped explicitly on the memory, conforming to the final memory mapping.

6.2.2 Hardware Components

The components of the hardware platform are those at the previous abstraction levels but detailed with cache memories, scratch pads, memory management units, and special registers. The hardware architecture contains all the resources required to verify the final software stack. Therefore, it contains the local components of each processor and hardware subsystem. In order to execute the software stack, the virtual platform contains an instruction set simulator (ISS) corresponding to each processor core.

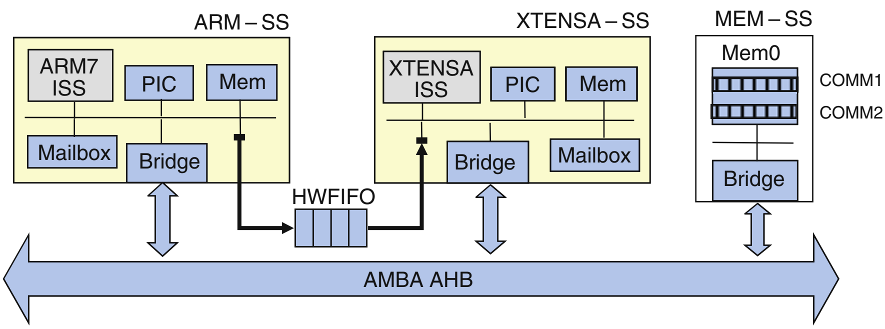


Fig. 6.3 Hardware at virtual prototype level

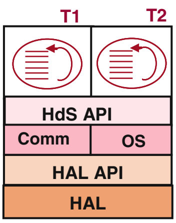
Example 33. Hardware components for the token ring application at the virtual prototype level For example, in the case of the token ring application running on the 1AX architecture, the hardware platform contains ISS encapsulated in the processor subsystems, specific to the ARM7, respectively, XTENSA processors, as it is illustrated in Fig. 6.3

6.3 Modeling Virtual Prototype in SystemC

The virtual prototype is modeled according to the annotated architecture parameters of the initial system architecture model and the results of the virtual architecture and transaction-accurate architecture models simulation.

6.3.1 Software at Virtual Prototype Level

The software design at the virtual prototype level consists of developing the final software binary that will run on each processor of the hardware platform. The binary image is obtained from the final software stack. This software stack contains all the software components: those verified at the transaction-accurate architecture level, namely, the application tasks code, operating system, and communication library, and an additional low-level component, more precisely the HAL. Figure 6.4 shows the software organization at this level and the fragment of the HAL code which performs a context switch for the ARM7 processor.



OS

```
void __schedule(void) {
  int old_tid = cur_tid;
  cur_tid = new_tid;
  __cxt_switch( old_tid.cxt, cur_tid.cxt ); // HAL API
...}
```

HAL code for ARM7 CPU

```
__ctx_switch           ; r0 old stack pointer, r1 new stack pointer
STMIA r0!,{r0-r14}     ; save the registers of current task
LDMIA r1!,{r0-r14}     ; restore the registers of new task
SUB pc,lr,#0           ; return
END
```

Fig. 6.4 Software at the virtual prototype level

Reference [174] defines the HAL as all the software that is directly dependent on the underlying hardware. The HAL can be implemented in assembly language interpretable by the processor or specific C code. In fact, the HAL includes two types of software code:

– Processor-specific software code, such as context switch, boot code, or code for enabling and disabling the interrupt vectors
– Device drivers, which represent the software code for the configuration and access to the hardware resources, such as MMU (memory management unit), system timer, on-chip bus, bus bridge, I/O devices, resource management, such as tracking system resource usage (check battery status) or power management (set processor speed)

Generally, the HAL provides the following kinds of services:

– Context switch, boot code, or code for enabling and disabling the interrupt vectors
– Integration with an ANSI C standard library to provide the familiar C standard library functions, such as *printf()*, *fopen()*, *fwrite()*. An example of such a kind of library is the newlib library, which is an open-source implementation of the C standard library .newlib for use on embedded systems [112]
– Device drivers to provide access to each device of the hardware platform
– The HAL API to provide a consistent interface to HAL services, such as device access, interrupt handling, and debug facilities
– System initialization to perform the initialization of the tasks for the processor before the execution of the *main()* function of the application
– Device initialization to instantiate and initialize each device in the hardware platform before the execution of the *main()* function of the application

The device drivers, that are part of the HAL, can provide access for the following classes of hardware devices:

– Character-mode devices, which represent hardware peripherals that send and/or receive characters serially, such as a UART (universal asynchronous receiver/transmitter) device
– Timer devices, which are hardware peripherals that count clock ticks and can generate periodic interrupt requests
– File subsystems, which provide a mechanism for accessing files stored within physical devices. Depending on the internal implementation, the file subsystem driver may access the underlying devices either directly or by using a separate device driver. For example, a flash file subsystem driver may access a flash memory by using dedicated HAL APIs for the flash memory devices
– Ethernet devices to provide access to an Ethernet connection for a networking stack such as the NicheStack TCP/IP Stack [84]

– DMA devices that are peripherals that perform bulk data transactions from a data source to destination. Sources and destinations can be memory or another device, such as an Ethernet connection
– Flash memory devices, which are non-volatile memory devices that use a special programming protocol to store data

To create the complete binary software image, the designer has to develop the configuration and build files (e.g., makefile) which select and configure the library components (OS, communication, HAL) and control the compilation and linking of the different software components. Using a cross-compiler, the final target binary is created for each processor that can be executed on the target processor of the virtual platform.

Example 34. Software code for the token ring application at the virtual prototype level Figure 6.5 presents an example of HAL code performing a context switch between two tasks running on the ARM7 processor in the case of the token ring application. Instead of using a simulation model of the HAL APIs as it was employed at the transaction-accurate architecture level, the virtual prototype gives the final implementation of the HAL API *__ctx_switch(...)* by using an explicit HAL software code. The context switch needs two basic operations to be performed: store the registers of the current task and load the registers of the new task.

```
__ctx_switch              ; r0 old stack pointer, r1 new stack pointer
STMIA r0!,{r0-r14}        ; save the registers of current task
LDMIA r1!,{r0-r14}        ; restore the registers of new task
SUB pc,lr,#0              ; return
END
```

Fig. 6.5 HAL implementation for context switch on ARM7 processor

Figure 6.6 illustrates the low-level implementation of the HAL API that sets the context for a task and initializes the stack required for the execution. The implementation is given for both ARM7 and XTENSA processors.

Figure 6.7 illustrates another example of low-level code implementation of the HAL APIs that enable and disable the IRQ interrupts for the ARM processor. The interrupts are enabled and disabled by reading the CPSR (Current Program Status Registers) flags and updating bit 7 corresponding to bit I.

Figure 6.8 shows the implementation of the interrupt vector enabling and disabling for the Xtensa processor.

In order to select properly the libraries of all these software components (OS, communication, HAL) for the compilation of the software stack, a makefile is required. Figure 6.9 details a makefile used for cross-compilation of the software stack running on the ARM7 processor of the 1AX architecture. The makefile contains the path to the application tasks code, target OS (*los-kernel*), communication library (*los-comm*), and the HAL library corresponding to the ARM7 processor (*lib/arm7*). It also identifies the compiler to be used, which in this case represents the *arm-elf-gcc* cross-compiler provided by GNU [61].

Xtensa
```
void _set_context(thread_context_t buf, fcall_t function, void *stack)
{
        _setjmp(buf);
        buf[JB_PC] = (int)(function);
        buf[JB_SP] = (int)(stack + STACK_SIZE);
}
```

ARM7
```
_set_context:
        stmia    r0!, {r0-r10, fp, ip}        @ we save r0-r12
        stmia    r0!,  {r2}                   @ sp
        stmia    r0!,  {r1}                   @ lr
        stmdb    sp!, {r4-r5}
        mrs      r5, cpsr                     @ we get the cpsr
        mrs      r4, spsr                     @ and the spsr
        stmia    r0!, {r4-r5}                 @ we can save them
        ldmdb    sp!, {r4-r5}
        mov      pc, lr                       @ and we branch
```

Fig. 6.6 HAL implementation for Set_Context on ARM7 and XTENSA processors

```
__inline void enable_IRQ(void)                          //HAL API
{
        inttmp;
        __asm
        {
                MRS tmp, CPSR
                BIC tmp, tmp,#0x80
                MSR CPSR_c, tmp
        }
}

__inline void disable_IRQ(void)                         //HAL API
{
        inttmp;
        __asm
        {
                MRS tmp, CPSR
                ORR tmp, tmp, #0x80
                MSR CPSR_c, tmp
        }
}
```

Fig. 6.7 Enabling and disabling ARM interrupts

It also includes the path to the linker script *ldscript* used to coordinate the linking process of the different object files obtained after the compilation. The *ldscript* guides also the loading process of the software image into the memory, by specifying explicitly the addresses where to load the program and data code of the software stack.

More details about the memory mapping will be given in the next paragraphs.

```
void vector_enable(int irq_type){
          if(intr_init==0) {
                    _xtos_set_interrupt_handler(0, _irq_dispatch);
                    intr_init =1;
          }
          _xtos_ints_on ( 1L << (0));
}

void vector_disable(int irq_type){
          _xtos_ints_off ( 1L << (0));
}
```

Fig. 6.8 Enabling and disabling XTENSA interrupts

```
CC      =  arm-elf-gcc                                              # cross compiler
OBJDUMP= arm-elf-objdump
OSHOME =  /home/popovici/dwarfos

INCDIR = .
SRCDIR = .
OBJDIR = .
BINDIR = .

FLAGS  = -Wall -D_SIMULINK_ -DSTACK_SIZE=0x1000 -g -I$(INCDIR)
FLAGS += -I$(OSHOME)/include -I$(OSHOME)/include/libc
FLAGS += -DARCH_ARM7 -T$(OSHOME)/lib/arm7/ldscripts
FLAGS += -nostdinc -nostdlib -nodefaultlibs -g
LIBS   = -lh264-arm7 -L$(OSHOME)/lib/arm7                        # HAL library
LIBS  += -los-kernel -los-comm -lgcc -lc                        # OS & Comm libs
OBJSUF  = .o

SRC     = $(wildcard $(SRCDIR)/*.c)
OBJ     = $(SRC:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
OSOBJS  = $(OSHOME)/lib/arm7/libos-hds.o

TARGET  = sw.bin


all:      $(TARGET)

$(TARGET): $(OBJ)
          @echo
          @echo 'creating binary "$(TARGET)"'
          $(CC) -o $(TARGET) $(OSOBJS) $(OBJ) $(LIBS) $(FLAGS)
          $(OBJDUMP) -D $(TARGET) > sw.d
          @echo '... done'
          @echo
```

Fig. 6.9 Example of compilation makefile for ARM7 processor

6.3.1.1 Loading Software Image in Memory

An important aspect of the virtual prototype design consists of loading the binary image of the software into the memories of the chip. Usually, MPSoC architectures provide complex memory hierarchies composed of different memories, such as ROM, SRAM, DRAM, FLASH. The binary image obtained after the compilation and linking is divided into two sections: read-only (RO) which contains the code and data only for read operations, and read–write (RW) section which contains the data that can be both read and written. Usually the RO part is loaded into a ROM memory. The RW part is stored in the ROM before the execution, and then it is initialized from the ROM into a RAM memory.

The structure of a binary image is defined by the number of regions and output sections, the positions in the memory of these regions and sections when the image is loaded, and the positions in the memory of these regions and sections when the image is executed. Each output section contains one or more input sections. Input sections are the code and data information from the object files obtained after the compilation.

The image regions are placed in the system memory map at load time. Then, before execution of the image, some regions are moved to their execution addresses and some parts of memory are set to zero creating the ZI (zero initialize) sections. Thus, there are two different views of the memory map: load view and execution view. The load view defines the memory in terms of addresses where the sections are located at the load time, before execution of the image. The execution view describes the address of the sections while the image is executing. Figure 6.10 shows the load and execution view of the memories.

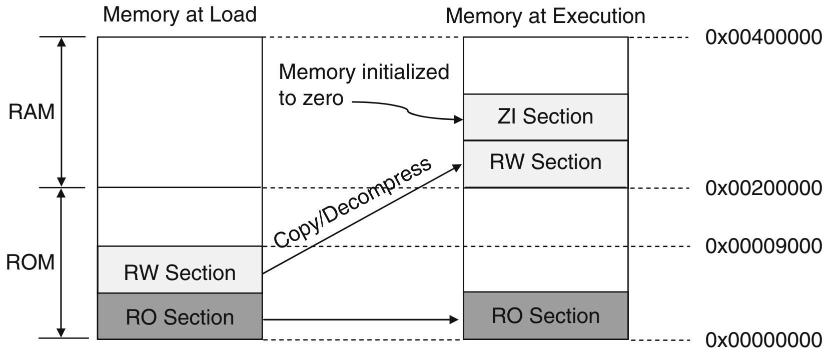


Fig. 6.10 Load and execution memory view

The image memory map is specified during the linking phase. The linking can be done using command line options for software images with few loading and execution sections or by using scatter-loading description file for more complex cases. The scatter-loading description file represents a text file that specifies the grouping information of sections into regions and the placement addresses of the regions to be located in the memory maps. The scatter-loading description file also allows to place the data at a precise address in the memory map to access memory-mapped I/Os and peripherals. Moreover, stack and heap addresses are defined using the same description file.

Figure 6.11 shows an example of scatter-loading description file for an ARM processor according to the memory mapping described in Fig. 6.10 [6].

```
ROM_LOAD 0x0                    ; Start address of load region
{
    ROM_EXEC 0x0 0x9000         ; Start address and maximum size of exec region
    {
        * (+RO)                 ; Place all code and RO data in this exec region
    }

    RAM 0x00200000 0x00200000 ; Start address and maximum size of exec region
    {
        * (+RW,+ZI)             ; Place all RW and ZI data into this exec region
    }
}
```

Fig. 6.11 Example of scatter-loading description file for the ARM processor

This scatter-loading descriptor example defines one load region (*ROM_LOAD*) and two execution regions (*ROM_EXEC* and *RAM*). The entire program, including code and data, is placed in ROM at *ROM_LOAD*. The RO code will execute from *ROM_EXEC*. Its execution address (0×0) is the same as its load address (0×0), so it does not need to be moved being a root region. The RW data will get relocated from *ROM_LOAD* to RAM at address 0×00200000. The ZI data will get created in RAM, above the RW data.

Before the execution of the binary image, the processor runs an initialization sequence code to set up and configure the system. Figure 6.12 presents an example of initialization code using HAL for the ARM processor [6].

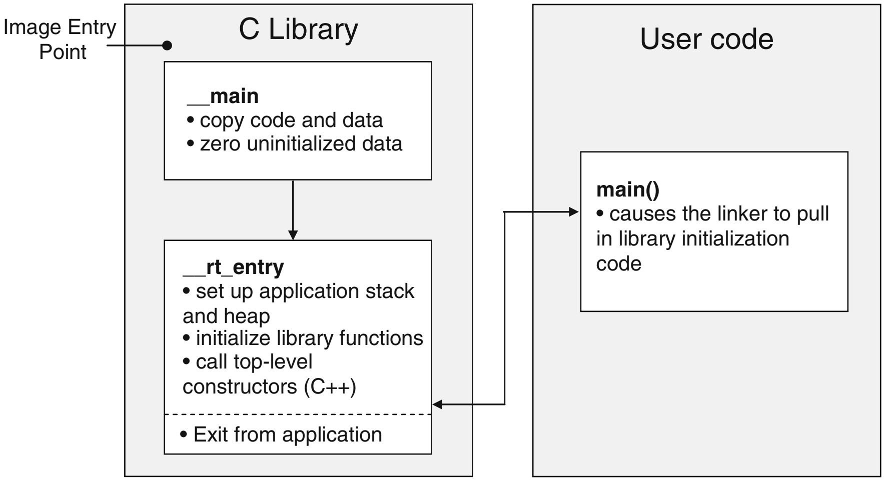


Fig. 6.12 Example of initialization sequence for the ARM processor

The initialization sequence has two principal functions: *__main* and *__rt_entry*. The *__main* function is responsible for setting the run-time image memory map. It also performs the copy of code and data and initializes the ZI section with zero. The *__rt_entry* (run-time entry) function is responsible to set up the application stack and heap memories, to initialize the library functions and static data, and it calls the constructors of global objects declared in C. Then, the *__rt_entry* function continues with the *main* user function, which represents the entry point of the software stack. For instance, the *main* function can be the initialization function of the OS that declares and initializes the tasks running on the processor.

6.3.2 Hardware at Virtual Prototype Level

The hardware platform is fully detailed with cycle-accurate TLM or RTL components for the hardware resources. The hardware at the virtual prototype level is comprised of the same components as at the transaction-accurate architecture level. In order to reach accurate performance estimation, the hardware modules are modeled at this level with cycle accuracy. Cycle accuracy can be achieved in two modeling methods:

– TLM modeling of the virtual prototype and use of execution delay annotation for cycle accuracy
– RTL (register transfer level) modeling of the virtual prototype

Both methods can make use of SystemC design language. The TLM modeling method has the advantage to ensure a fast simulation environment, while the RTL modeling may allow synthesizing the hardware architecture within a hardware–software MPSoC co-design flow.

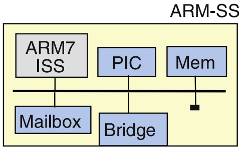
The virtual prototype contains ISS for the processors in the processor subsystem to execute the software stack.

Example 35. Hardware code for the token ring application at the virtual prototype level The virtual prototype in the case of token ring application running on the 1AX architectures is modeled using cycle-accurate TLM.

Figure 6.13 shows an example of processor subsystem for the ARM7-SS of the 1AX architecture running the token ring application. The ARM7-SS includes the processor core ArmCore SystemC module, which encapsulates an ISS of the software. The rest of the components of the ARM7-SS are those from the transaction-accurate architecture level (local bus, local memory bridge, interrupt controller, and mailbox).

6.3.3 Hardware–Software Interface at Virtual Prototype Level

At the virtual prototype level the communication consists of physical I/Os, e.g., *load/store*. The hardware–software interface is represented by the ISS for the processors. An instruction set simulator (ISS) is a simulation model, usually coded in a high-level language such as C language, which mimics the behavior of a microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers.




```
#include "ARM7_SS.h"
extern int debug_flag;

ARM7_SS::ARM7_SS(sc_module_name name, char *bin)          // ARM7-SS
:sc_module(name)
{
    sys_bus = new TlmBus("sys_bus");                       // local bus

    core = new ArmCore("ARM7_ISS",bin,debug_flag);         // ARM7 ISS
    core->rw_port(*sys_bus);

    mem = new Sram("mem0",0x300000);                       // local memory
    mem->port(*sys_bus);
    mem->port.set_map(0x0,0x2FFFFF);

    bridge = new AhbIf("bridge");                          // bridge
    bridge->master(*sys_bus);
    bridge->slave(*sys_bus);
    bridge->slave.set_map(0x400000,0x7fffffff);

    pic = new Pic<1>("pic",0x20);                          // PIC
    pic->port(*sys_bus);
    pic->port.set_map(0x300000,0x30001f);

    sync = new Sync("sync",0x400);                         // mailbox
    sync->port(*sys_bus);
    sync->port.set_map(0x300800,0x300bff);

    TlmIntrSig *sig_sync = new TlmIntrSig("sig_sync");   //interrupt signals
    sync->intr(*sig_sync);
    pic->in_irq[0](*sig_sync);
    s1 = new TlmIntrSig("sig_intr1");
    s2 = new TlmIntrSig("sig_intr2");
    pic->out_fiq(*s1);
    pic->out_irq(*s2);
    core->nIrqPort(*s2);
}
```

Fig. 6.13 SystemC CODE of the ARM7-SS module

6.4 Execution Model of the Virtual Prototype

The integration of instruction set simulators for the software execution on specific processors with hardware simulators of the architecture behavior is largely used in MPSoC domain. By using ISS, this approach allows simulating a detailed hardware–software interaction. The timing information can be measured instead of estimated as at the previous abstraction levels and design steps.

The execution model of the virtual prototype resides on a co-simulation between the software stack simulator and the hardware simulator [110]. Two types of simulators are combined: one for simulating the programmable components running the software and one for the dedicated hardware part [47]. The software stack is executed using processor-specific ISS. Instruction-level or cycle-accurate ISS simulators are commonly used. The hardware simulation is performed using hardware RTL descriptions realized in VHDL, Verilog or SystemC, or cycle-accurate TLM description realized in SystemC. In the following examples, we use SystemC for the hardware simulation.

The hardware–software simulation is driven by SystemC. The SystemC initializes the processor SystemC modules that encapsulate the ISS. During the simulation, the ISS features a simulation loop which fetches, decodes, and executes instructions one after another. The ISS is developed as sequential software running on a single processor. The simulation performed at this level is cycle accurate. The simulation of the virtual prototype allows validating the memory mapping of the target architecture and the software binary.

Example 36. Execution model for the token ring application at the virtual prototype level Figure 6.14 shows the execution model of the 1AX architecture running the token ring application. The model contains two ISS to execute the binary codes, corresponding to the ARM7, respectively, XTENSA processors. The rest of the architecture components are cycle-accurate SystemC components modeled at TLM with execution timing information.

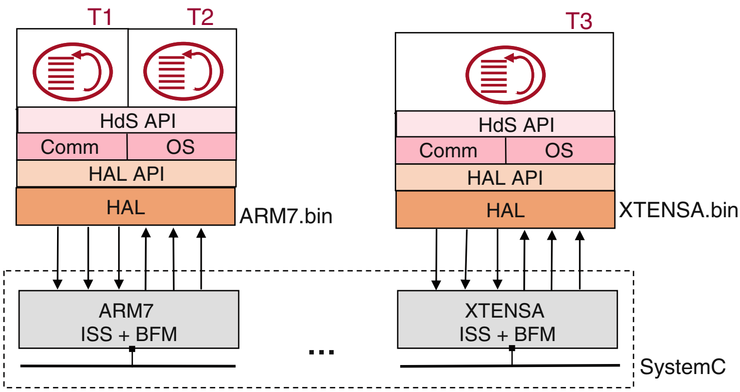


Fig. 6.14 Execution model of the virtual prototype

6.5 Design Space Exploration of Virtual Prototype

6.5.1 Goal of Performance Evaluation

The goal of the performance evaluation at the virtual prototype level is to validate the final software stack and the overall performance of the system. The performance evaluation is related to both computation and communication aspects.

Based on the results obtained by executing the final software on the virtual prototype model, the designer may need to improve some parts of the design or revise design options due to unsatisfied design constraints, for example, if real-time requirements are not met, such as number of frames processed per second in multimedia applications, usually defined as 25 frames/s. Software optimization aims to decrease program and data size, usually achieved through application algorithm optimization or communication overhead reduction.

6.5.2 Architecture/Application Parameters

The virtual prototype validates the adaptation of the final software code to a specific processor.

The designer may choose different types of processor cores from the same processor family or different processor families. The different kinds of processor cores of the same family have a common architecture, but are differentiated by some specific features, such as size of data and instruction cache memories, bus interfaces, the availability of tightly coupled memory, power consumption, area, clock frequency (MHz), or DSP extensions. Table 6.1 shows a subset of different characteristics of processors belonging to the ARM7 family compared to the ARM926EJ-S processor of ARM9 family [6]:

Table 6.1 ARM7 and ARM9 processors family

	Cache size (Inst/data)	Tightly coupled mem	Mem Mgmt	Bus interface	Thumb	DSP	Jazelle
ARM720T	8 k unified	–	MMU	AHB	Yes	No	No
ARM7EJ-S	–	–	–	Yes	Yes	Yes	Yes
ARM7TDMI	–	–	–	Yes	Yes	No	No
ARM7TDMI-S	–	–	–	Yes	Yes	No	No
ARM926EJ-S	16 k/16 k	Yes	MMU	AHB	Yes	Yes	Yes

The designer may change these parameters and may set up different configuration schemes, including target compilation optimizations, to increase the overall performance.

6.5.3 Performance Measurements

The simulation of the virtual prototype provides precise performance information such as software execution time, computation load for the processors, the number of cycles spent on communication, the number of cycles spent by processors in idle state.

Other important metrics that can be measured at this level are program and data memory size requirements of the final software stack, number of cycles spent by the processor on certain application functions, or number of instructions executed per clock cycle. This kind of data can be gathered thanks to the precise profiling capabilities of most of the instruction set simulators. Usually, the virtual prototype is a cycle-accurate model, thus it implies long simulation time. Therefore, the simulation time represents another key feature to be measured at the virtual prototype level.

Example 37. Performance measurements for the token ring application at the virtual prototype level In the case of the token ring application, the execution of the three tasks on a single ARM7TDMI processor without operating system requires 484,775 clock cycles running at 60 MHz. The application compiled for a single ARM7 processor produces a code size of 1,112 bytes and 108 data bytes. The computation of the FFT on the ARM7 processor involves 33,329 clock cycles.

6.5.4 Design Space Exploration

At the virtual prototype level, the design space exploration consists of processor core configuration and exploration. The different types of processor cores or differently configured processors have different performances in terms of speed, power consumption, and cost.

Example 38. Design space exploration for the token ring application at the virtual prototype level For instance, the XTENSA processor is a configurable processor provided by Tensilica [152]. The SoC designers may customize functional blocks to exactly match the required application. Because these processors are fully programmable, changes can be made in firmware even after the silicon production.

Generally, the configurable processors have two essential features:

– Configurability, which allows the designers to pick and configure the features they need.
– Extensibility, which allows designers to add multi-cycle execution units, registers, or register files. For instance, the Tensilica Instruction Extension (TIE) of the XTENSA processors is a methodology that allows designers to specify and verify the functional behavior of the new data path and the RTL is automatically generated [152].

Another space that can be explored at the virtual prototype level represents the memory mapping. Thus, the different data structures can be mapped on different memories at different addresses accessible through *load/store* instructions.

6.6 Application Examples at the Virtual Prototype Level

The following paragraph presents the virtual prototype model for the two case studies: the Motion JPEG decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 encoder application running on the Diopsis R2DT architecture with Hermes NoC.

6.6.1 Motion JPEG Application on Diopsis RDT

At the virtual prototype level, the software stacks of the Motion JPEG decoder application running on the two processors contain all the components. A processor-specific HAL layer is linked with the application tasks, operating system, and communication libraries. Usually, the HAL layer is provided by the processor vendors. Thus, a specific ARM7 HAL is implemented in the final software running on the ARM7. Similarly, the HAL of the DSP is integrated in the software stack. The two software stacks produce two different binary images that will be interpreted and executed by the ISS corresponding to each of these processors.

The hardware platform contains cycle-accurate detailed components using TLM modeling with timing annotation or RTL modeling. Figure 6.15 illustrates a global view of the virtual prototype platform with the use of ISS as processor execution model.

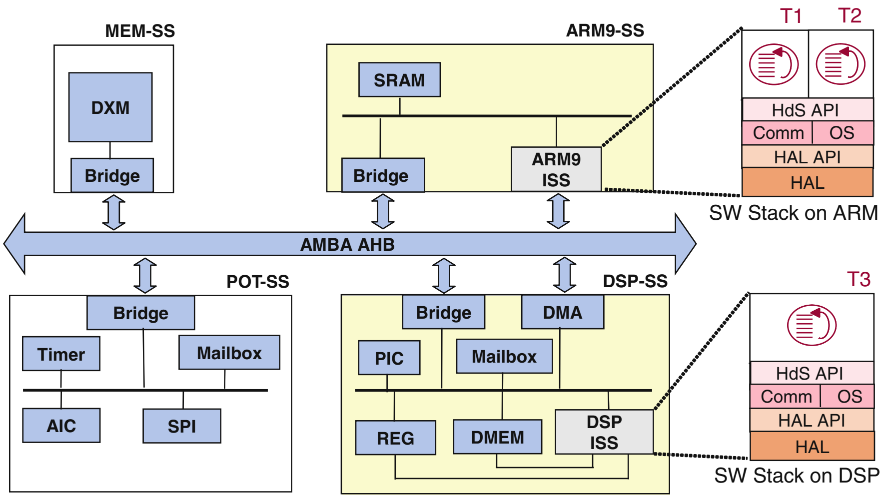


Fig. 6.15 Global view of the virtual prototype for Diopsis RDT with AMBA bus running motion JPEG decoder application

Figure 6.16 illustrates a piece of HAL code for DSP processor, implementing the *_set_context()*, *vector_enable()*, and *vector_disable()* HAL APIs. The first HAL code sets the context of the application task running on the DSP, while the others

```
void_set_context(thread_context_tbuf, fcall_tfunction, void *stack)
{
           _DBIOS_Init(function,buf,stack);
}

voidvector_enable(intirq_type){
          AT91F_DSP_INTERRUPT_Enable(irq_type);
}

void vector_disable(intirq_type){
          AT91F_DSP_INTERRUPT_Disable (irq_type);
}
```

Fig. 6.16 Global view of the virtual prototype for Diopsis RDT with AMBA bus running motion JPEG decoder application

implement the interrupt vector enabling and disabling services. The implementation of these HAL APIs relies on the other APIs that are provided by the DSP vendor, Atmel.

Figure 6.17 summarizes the total execution cycles measured when executing the whole Motion JPEG application on a single-processor single-task configuration. The experimentation was done using three types of processor cores. The first processor core represents the ARM7TDMI-S processor of the ARM7 family. This processor

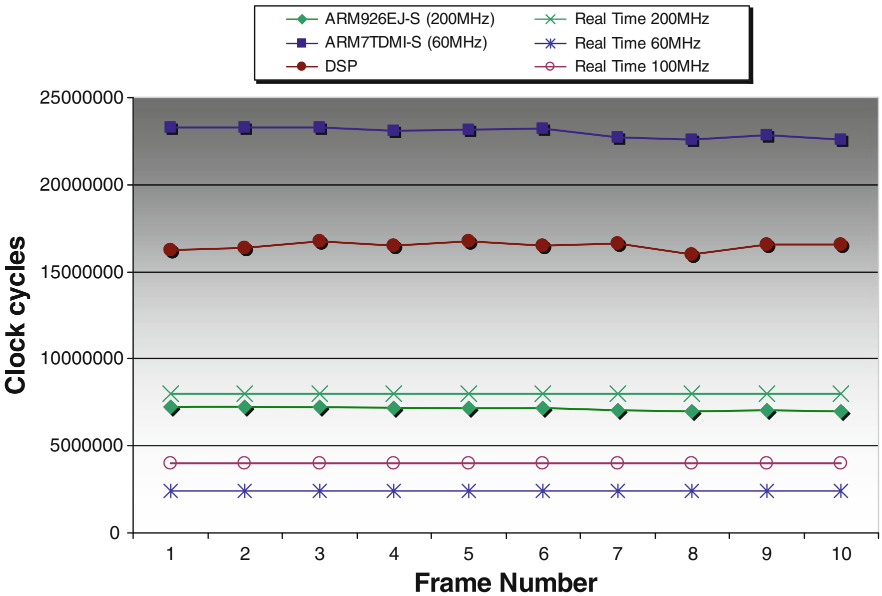


Fig. 6.17 Execution clock cycles of motion JPEG decoder QVGA

works at 60 MHz frequency and has no data cache or instruction cache memories. The second core belongs to the ARM9 processor family and represents the ARM926EJ-S type of core. This runs at 200 MHz frequency and is equipped with 16 kB data cache and 16 kB instruction cache memories. The third processor represents the magicV VLIW DSP processor, running at 100 MHz. In all the cases, the real-time execution requirement defines an image rate equal to 25 images/s to be decoded.

As shown in Fig. 6.17, the number of execution cycles required to decode an image is approximately 7 mega cycles on the ARM9 processor, 16 mega cycles on the DSP processor, and 23 mega cycles on the ARM7 processor.

The performance difference between the two ARM processors is explained by the availability of the additional cache memories and improvement in the number of cycles required for load/store operations characterizing the ARM9 core family compared with the ARM7 core. The real-time requirement implies 8 mega cycles on a CPU running at 200 MHz, 4 mega cycles on a CPU running at 100 MHz, and 2.4 mega cycles on a CPU running at 60 MHz. Thus, the MJPEG decoder can be executed in real time by using the ARM9 processor, while the execution on a single ARM7 processor requires application code optimization. The execution on the DSP can be improved by using DSP-specific optimization features.

After the compilation of the MJPEG decoder application, the memory requirements are as follows: 7,592 bytes of code size for the program memory and 1,402 bytes data memory. These values were obtained in case of targeting both ARM7 and ARM9 processor cores using the CodeWarrior development tool [100]. In case of the DSP processor, the MJPEG decoder requires 614 bytes data memory and 2,806 bytes program memory.

In order to compare the virtual prototype simulation with the actual physical implementation, the Motion JPEG application can be executed on an FPGA-based platform. The application binary is loaded and executed on the ARM 9 processor of the FPGA emulation platform of the Diopsis RDT architecture. Thus, in order to validate the correctness and efficiency of the multi-level software design flow, the generated software stack of the MJPEG application maybe executed on a Diopsis emulation platform with Xilinx Virtex-II XC2V8000 FPGA provided by Atmel Inc. In this example, all the four tasks of the Motion JPEG application are mapped onto the ARM9 processor. Then, the software stack is designed and verified incrementally. For the tasks management, a tiny OS can be used to implement basic OS services, such as tasks scheduling and software FIFO channels for the communication between the tasks. This OS must be enriched to support specific context switch for the ARM9 processor. A multi-ice GDB debugger server can be used in order to load the final software binary image on the local SDRAM memory. The FPGA platform-based emulation ensures the reliability of the software code's functionality.

Compared with virtual prototype simulation, the FPGA execution is faster than simulation. Often, the application can run at full speed. Also, unlike simulation, the FPGA does not necessarily slow down as more hardware components are integrated in the design. So, it becomes possible to test the entire design rather than individual components and to run tests with large real-life data sets instead of surgically crafted test cases.

Because of the speed and the fact that the FPGA has the actual I/O cells that the design requires, someone can also test the design in-system: either in an FPGA development board that the tester has lashed into the target system or in the target PCB (printed-circuit board) if it is ready to go. Such testing eliminates the lingering uncertainty about whether the test cases really reflect the operating environment of the design. Also, testing the application in its actual board can uncover I/O-related issues – electrical problems, signal-integrity issues, or incompatibilities in high-speed serial protocols, for example – that would be virtually undetectable in any other way.

But the main disadvantage of the FPGA-based execution is the signals visibility. In the simulation environment, all the signals can be observed and accessed easily. This is not the case for execution on a FPGA.

6.6.2 H.264 Application on Diopsis R2DT

The H.264 encoder running on the Diopsis R2DT architecture at the virtual prototype level is illustrated in Fig. 6.18. In the same way as in the case of the Motion JPEG decoder, there are three final software stacks running on the architecture, one per each processor. The HAL libraries were included in the software stack for each particular CPU.

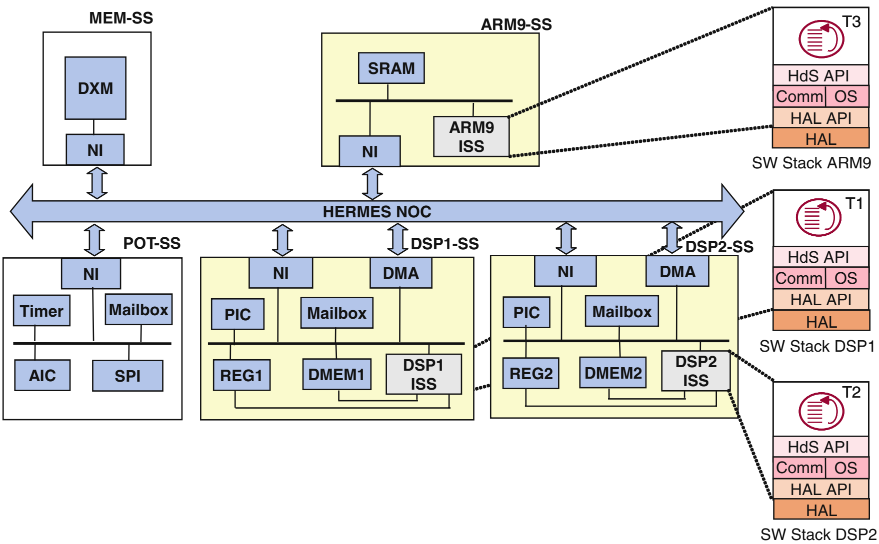


Fig. 6.18 Global view of the virtual prototype for Diopsis R2DT with Hermes NoC running H.264 encoder application

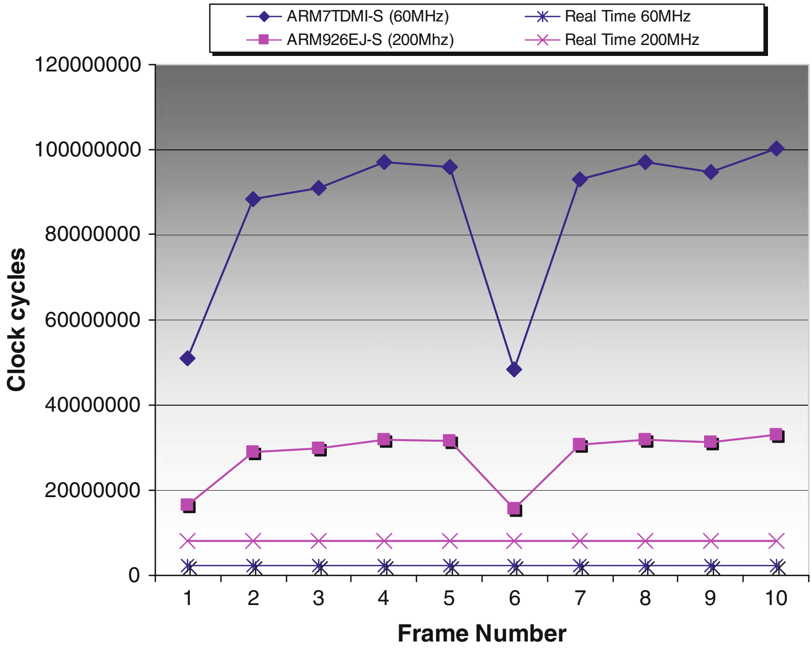


Fig. 6.19 Execution clock cycles of H.264 encoder, main profile, QCIF video format

The hardware platform includes ISS to execute the final software. The ISS allows determining the execution cycles spent on each task. The virtual prototype of the Diopsis R2DT running the H.264 encoder application is illustrated in Fig. 6.18.

Figure 6.19 captures the results of executing the H.264 encoder application, main profile, QCIF video resolution on the ARMTDMI-S and ARM926EJ-S processors. In this single task fashion, the H.264 encoder requires around 30 mega cycles to encode a P frame and 16 mega cycle for encoding an I video frame on the ARM9 CPU running at 200 MHz. If the target processor is the ARM7 core, the encoder requires approximately 50 mega cycles for a frame type I and 90 mega cycles for a frame type P.

As shown in Fig. 6.19 both results do not respect the real-time encoding requirement established at 25 frames encoded per second. Running on a single processor, the achieved frame rate is 9 frames/s for a P frame and 12 frames/s for the I video frame. The H.264 results represented in Fig. 6.19 consider a key frame of five frames, which mean that between two I frames there are five video frames that will be encoded as P frame.

Figure 6.20 shows the program and code size of the H.264 application compiled with the CodeWarrior tool targeting the ARM7 and ARM9 processors. The data size

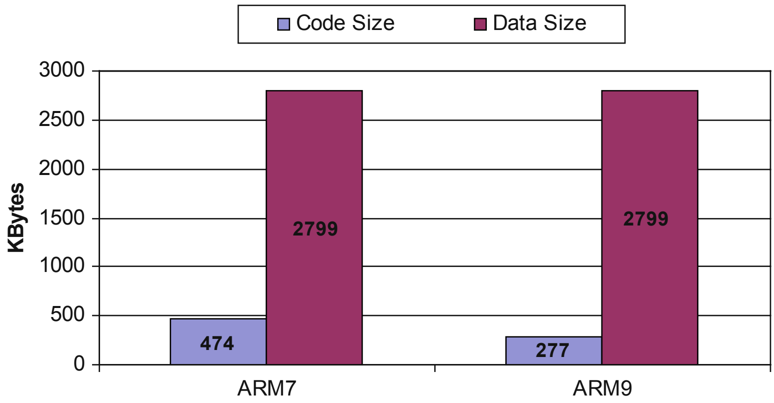


Fig. 6.20 Program and memory size

has the same value for both types of processor cores, being equal to 2,799 kB, while the program size is 474 kB compiled for the ARM7 processor and 277 kB for the ARM9 core.

6.7 State of the Art and Research Perspectives

6.7.1 State of the Art

Currently, virtual prototype environments for modeling and simulation based on SystemC, such as Maxsim [7], Coware ConvergenSC [41], and Synopsys System Studio [147], provide a rich set of components such as processors, memories, and peripherals that can be extended by user-defined modules.

The concept of virtual platform appears in [70] with the purpose to allow software development and code optimization before the real board is available. Reference [119] uses virtual prototype simulation to perform software profiling, such as total execution cycles and software performance analysis.

Execution of software using an ISS still suffers from low simulation speed compared to real hardware. Therefore, many researchers focus on developing new techniques to attain high simulation speed. In this context [136] mixes the interpreted ISS simulation with the compiled ISS simulation in order to allow a multi-processing simulation approach to increase simulation speed. Reference [144] describes an ultra-fast ARM and multi-core DSP instruction set simulation environment based on just-in-time (JIT) translation technology, which refers to the dynamic translation of the target instructions (ARM, DSP) to the host instructions ($\times 86$) during the execution.

Reference [80] presents a fast and hybrid simulation framework which allows switching between native code execution and ISS-based simulation. In this approach, the platform-independent parts of the software stack are executed directly on the host machine, while the platform-dependent code executes upon an ISS. Thus, the framework allows debugging a complex application through executing it natively until the point where the bug is expected, and then executing on the ISS to examine the detailed software behavior.

Other research groups focus on integrating ISS within existing design flows. For instance, [91] presents a framework of ISS integration within the Ptolemy design environment that leverages the approach of time-approximate co-simulation based on source code estimation of execution time and refines its precision by using an ISS.

Reference [120] presents the automatic generation of virtual execution platforms for the hardware architecture to analyze the run-time behavior of the application running on a real-time operating system and to estimate accurately performance data.

Reference [76] allows generation of synthesizable communication from high-level TLM communication models. The scope of this work is to reduce the gap between TLM and RTL designs for automating MPSoC synthesis.

Reference [139] proposes automatic software synthesis from TLM platforms. They support automatic generation of HdS, including code generation, communication software synthesis, multi-task synthesis, and generation of the configuration and makefiles to control the cross-compilation and linking of the generated code for a particular processor.

6.7.2 Research Perspectives

The virtual platform has to be available earlier than the real hardware in order to allow concurrent software and hardware designs. Therefore, one research perspective regarding the virtual prototype design relies on the automation of the generation process by using the architecture parameters annotating the system architecture model and the simulation results of the higher abstraction levels. The automatic generation of the virtual prototype model shortens the design time and permits to reduce human coding errors.

Virtual prototype uses instruction set simulators for the software execution. This implies high-accuracy and low-simulation speed. Moreover, the simulation time increases exponentially with the number of processor cores integrated on the same chip. Thus, finding new methodologies that speed up simulation but still maintain accurate performance evaluation represents another important issue for future research perspectives.

Another important research perspective, related to the considered case studies, represents the simulation of the MJPEG and H.264 applications running on the multi-processor architecture. As the target architectures include commercial off-the-shelf DSP processors and their compiler and instruction set simulator were provided as stand-alone applications, the integration of the DSP instruction set simulators into a hardware simulation environment, such as the one previously described in

SystemC, represents an essential future perspective. Generally, the integration of ISS into an existing platform imposes development of a software simulation wrapper that interacts with the hardware model and solves the synchronization problem for the hardware and software interaction.

6.8 Conclusions

This chapter detailed the virtual prototype design. The virtual prototype design consisted of integrating the HAL component into the software stack, cross-compiling it for the target processor, and fixing the final memory mapping.

The verification of the software binary was performed by using instruction set simulators (ISS). Thus, the token ring application was executed on the 1AX architecture, Motion JPEG on the Diopsis RDT architecture, and H.264 encoder on the Diopsis R2DT architecture.

The Motion JPEG application was also executed using ISS on a single processor (ARM7, ARM9, and DSP) and the H.264 encoder was simulated using ISS running on both multi-processor architecture with three ARM7 processors and single processor (ARM7 and ARM9). The simulation of the virtual prototype model allows to verify the final software binary and the memory mapping.

Chapter 7
Conclusions and Future Perspectives

7.1 Conclusions

The current design practice for system-on-chip is RTL (register transfer level) design with a late integration of hardware and software. Its shortcomings are inefficient design, long design cycles, over-design of the hardware, etc. Most recently designers have started to adopt virtual prototypes in practice. These help with earlier integration leading to a better software and earlier availability and improved design. However, this phase adds many man months of effort to a typical project and has the following disadvantages: still requires long simulation time; the architecture details are not necessary for the design of all types of software components; it is difficult to identify the different sources of the software bugs.

To overcome these challenges, we propose and show in our examples how the transaction-accurate architecture design can help with these challenges. The additional effort of only few man months to create these models is more than offset by the following value it provides: more than 1,000 times faster simulation speed, early performance estimation with transaction level accuracy, early verification of the tasks scheduling by the OS. Concrete examples from the case studies presented in the book help to better understand this design step and to create new models in a relative short time. Still, at this phase designers are facing the following shortcomings: it is difficult to map application functions onto processors running an OS without a prior validation of the tasks code; the scheduling of the communication primitives, that serve to exchange data between the tasks, needs to be verified before the execution upon an OS, in order to find possible deadlocks (a condition that occurs when two tasks are each waiting for the other to complete before proceeding; the result is that both processes hang) or livelocks (a condition that occurs when two or more tasks continually change their state in response to changes in the other tasks; the result is that none of the processes will complete).

These shortcomings can be addressed in a phase called virtual architecture design, which adds the value of allowing validation of the application partitioning and mapping on the processing units. But the virtual architecture design does not address the application algorithm design, optimization, and verification, which in exchange can be addressed by a system architecture design phase.

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8_7,

Thus, this book presented a software design and verification flow able to efficiently use the resources of the architecture and allowing easy experimentation of several mappings of the application onto the platform resources. The book presented two case studies: Simulink environment to capture both application and architecture initial representations and SystemC for the low-level software design. The software generation and verification was performed gradually from this initial model corresponding to different software abstraction levels. Specific software development platforms (abstract models of the architecture) in SystemC were used to allow debugging the different software components with explicit hardware–software interaction.

The presented software design flow decreases the complexity of the design process by structuring it into several layers. The different components of the software stack were generated and validated incrementally: the simulation at system architecture level validated the application's functionality; the virtual architecture level simulation allowed debugging the final application tasks code; the execution at the transaction accurate architecture level validated the integration of the tasks code with the OS and communication library, while the virtual prototype enables the validation of the binary image.

Besides the software debug, the platforms also allow to accurately estimate the use of the hardware resources by counting the total number of transactions exchanged during the simulation. The software design flow also made it possible to optimize the communication performance by using the architecture capabilities. The communication optimization relied on easy experimentation of different mappings of the communication onto the platform resources, using simple annotations of the initial Simulink model and generating the corresponding platforms.

The flow is able to facilitate programming existing hardware platforms that contain heterogeneous multiprocessor architectures with specific I/O components. The design flow allows mapping sophisticated software organized into several stacks made of different layers on these platforms. The new software design flow masters the complexity of the software design process. This is achieved thanks to the incremental software layers generation and corresponding software development platforms generation. These platforms are able to abstract multimedia architectures at different abstraction levels and enable separate debug of the software components.

Apart from the case studies presented in this book, the programming environment case studies have been applied successfully for the following multimedia applications running on the corresponding MPSoC architectures [124]:

- √ Token ring application targeting the 2A1X (two ARM processors and one XTENSA processor interconnected through the AMBA bus) and Diopsis RDT architectures
- √ MP3 audio decoder running on the Diopsis RDT architecture with AMBA bus
- √ Vocoder audio encoder executed on the Diopsis RDT architecture with AMBA bus

√ Motion JPEG image decoder running on the following architectures: 1AX, 2A1X, Diopsis RDT with NoC interconnect, Diopsis RDT with AMBA bus in normal mode without burst data transfers

√ H.264 video encoder, main profile, running on the following architectures: Diopsis RDT with NoC, Diopsis RDT with AMBA bus with and without burst transfer and Diopsis R2DT with AMBA bus

√ H.264 video decoder application, base profile, running on 1AX and 2A1X architectures

7.2 Future Perspectives

Future research perspectives tackle the following described items:

(i) *Automation of the software design and validation flow*

The automation of the software design flow concerns two aspects:

– automatic tools for the software stack construction
– automatic tools for the software simulation models generation

The automatic generation of the different MPSoC abstraction levels could be made possible by applying a service-based modeling of the hardware–software interface as described in [56]. The composition of the services allows the automatic generation tools to build easily the different software and hardware simulation models.

The automatic generation of the hardware and software architectures at the different abstraction levels shortens the design time and permits to reduce human coding errors.

(ii) *Automatic generation of the RTL hardware architectures*: The automatic generation of RTL to allow synthesis of the target hardware architectures represents future work. This would make possible hardware design in parallel with the software design for a specific application, allowing hardware implementation of some functions for a target application.

(iii) *Formalization of the hardware–software partitioning process*: The presented software design and validation flow uses system architecture model, which represents the partitioned model of the application onto the target architecture. Thus, formalizing the partitioning process represents another future perspective to allow early design space exploration. Design space exploration represents an essential issue to analyze the impact on performances by using different application partitioning, mapping, and communication schemes. Design space exploration allows finding the best combination of the application/architecture configurations to achieve the required communication and computation constraints. Future works focus on better parallelization of the applications and exploration of the different partitioning and mapping combinations.

Automatic tools for application partitioning, mapping, and evaluation metrics such as performance, power, and cost are necessary to fully explore the design space and help designer's choices. Therefore, future work should address estimation tools such as power estimation to meet the tight power constraints on MPSoCs. For instance, power estimation can be implemented by embedding cycle-accurate power model into each hardware component of the development platform.

Another aspect of future perspective for design space exploration constitutes the annotation of the intermediate abstraction platforms with execution delays to provide more accurate performance estimation at the design steps earlier than the virtual prototype design.

(iv) *Support of multiple applications:* The support of the multiple applications running on the same MPSoC architecture, i.e., an audio encoder combined with a video encoder application is also envisioned for the future. During the parallel execution of the multiple applications, the main difficulties that need to be overcome are related to the global scheduling and hardware resource sharing of the different applications.

(v) *Hardware–software co-design flow*: A completely automated hardware–software co-design flow represents another future research perspective. The flow involves a seamless refinement at the four abstraction levels (system architecture, virtual architecture, transaction-accurate architecture, virtual prototype). It requires automatic code generators for the software design, platform-based generators for the hardware design, and automatic hardware–software interfaces refinement.

Glossary

A

AHB advanced high-performance bus. AHB is an AMBA high speed bus.

ALU arithmetic/logic unit. The arithmetic/logic unit is a fundamental building block of the CPU's internal architecture. It performs logical and arithmetic operations.

AMBA The AMBA bus is a an open bus standard promulgated by ARM.

APB advanced peripheral bus. APB is an AMBA bus. The APB bus is devoted to low-speed peripherals and it is optimized to minimize power consumption and to reduce interface complexity.

API application programming interface. An application programming interface is a set of declarations of the functions that an operating system, library, or service provides to support requests made by computer programs.

ASB advanced system bus. ASB is an AMBA bus which is mainly deprecated and it has been substituted by AHB.

ASIC application-specific integrated circuit. An application-specific integrated circuit is an integrated circuit customized for a particular use, rather than intended for general-purpose use.

ASIP application-specific instruction set processor. An application-specific instruction set processor is a stored-memory CPU whose architecture is tailored for a particular set of applications.

AVC advanced video codec. Advanced video codec (AVC) is a standard for video compression. It is also known as H.264.

B

BDF Boolean dataflow. BDF is a model of computation based on dataflow. It is a generalization of SDF that sometimes yields to deadlock or boundedness analysis.

K. Popovici et al., *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Embedded Systems, DOI 10.1007/978-1-4419-5567-8,

BFM bus functional model. A bus functional model is used to simulate operating systems on the host machine. It transforms the functional memory access into hardware memory access.

BSB board support package. A board support package includes a set of minimal services necessary to load an operating system in an embedded system and the device drivers for all the devices on the board.

C

CABAC context adaptive binary arithmetic coding. CABAC is an entropy encoding algorithm, part of the H.264 video compression standard.

CAVLC context adaptive variable length coder. CAVLC is an entropy encoding algorithm, part of the H.264 video compression standard.

CORBA common object request broker architecture. The common object request broker architecture allows to execute different application objects that reside either in the same shared address space or remote address space. The application objects are described using interface definition languages (IDL).

Cycle accurate. cycle-accurate level is the most detailed abstraction level in our multiple abstraction levels design space exploration flow.

CPU central processing unit. A central processing unit is a description of a class of logic machines that can execute computer programs.

D

DCT discrete cosine transform. The DCT is a frequency transform used in video and image processing, whose coefficients describe the spatial frequency content of an image or video frame. The DCT operates on 2D set of pixels, in contrast to the Fourier transform which operates on a 1D signal.

DDF dynamic dataflow. DDF is a model of computation based on dataflow, which uses only runtime analysis to detect deadlock and boundedness.

DFT discrete Fourier transform. The DFT, occasionally called the finite Fourier transform, is a transform for Fourier analysis of finite-domain discrete-time signals. It expresses an input function in terms of a sum of sinusoidal components by determining the amplitude and phase of each component.

DMA direct memory access. The DMA is a component of the hardware architecture which allows accessing the memory independently of the processor. Usually it is used to initiate a data transfer between a local and global memory.

DSP digital signal processor. A digital signal processor is a specialized microprocessor designed specifically for digital signal processing, generally in real-time computing.

F

FIFO first in–first out. First in–first out is an abstraction in ways of organizing and manipulation of data relative to time and prioritization.

FPGA field programmable gate array. An FPGA is an integrated circuit which contains programmable logic components, which can be configured to perform complex combinational functions or logic operations. It can contain also memory elements.

FSM finite state machine. The FSM is a model of computation comprised of a finite number of states, transitions between those states and actions.

G

GPP general-purpose processor. A general-purpose processor is a processor that is not tied to, or integrated with, a particular language or piece of software.

H

HAL hardware abstraction layer. A hardware abstraction layer is an abstraction layer, implemented in software, between the physical hardware of a computer and the software that runs on that computer. Its role is to hide differences in hardware from most of the operating system kernel, so that most of the kernel-mode code does not need to be changed to run on systems with different hardware.

HdS hardware-dependent software. Hardware-dependent software is the part of an operating system which varies across microprocessor boards and is comprised notably of device drivers and boot code which performs hardware initialization.

I

IDCT inverse discrete cosine transform. A inverse discrete cosine transform expresses the opposite process of transforming a sequence of finitely many data points in terms of a sum of cosine functions oscillating at different frequencies.

ILP instruction-level parallelism. Instruction-level parallelism is a measure of how many of the operations in a computer program can be performed simultaneously.

IQ inverse quantization. The IQ is used in multimedia decoding applications and represents the opposite of the quantization process. It consists of the multiplication of each of the 64 DCT coefficients by its corresponding quanta step size. The quanta steps are stored in the quantification tables.

ISR interrupt service routine. An interrupt service routine is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt.

ISS instruction set simulator. An instruction set simulator is a simulation model which mimics the behavior of a mainframe or microprocessor by “reading” instructions and maintaining internal variables which represent the processor’s registers.

ITRS international technology roadmap for semiconductors. The international technology roadmap for semiconductors, known throughout the world as the ITRS, is the 15-year assessment of the semiconductor industry's future technology requirements. The future needs drive present-day strategies for world-wide research and development among manufacturers' research facilities, universities, and national labs.

J

JPEG Joint Photographic Experts Group. JPEG is a commonly used method and standard of compression for photographic images and is created by the joint photographic experts group.

M

MCU microcontroller. The microcontroller is a type of CPU specialized mostly on control functions and combined with support functions such as timers, watchdog, serial and analog I/Os.

MIMD multiple instructions multiple data. Multiple instructions multiple data are a technique employed to achieve thread-level parallelism.

MIPS millions of instructions per second. MIPS stands for "millions of instructions per second" and is a rough measure of the performance of a CPU.

MJPEG motion JPEG. In multimedia, motion JPEG is an informal name for multimedia formats where each video frame or interlaced field of a digital video sequence is separately compressed as a JPEG image.

MPEG Moving Picture Experts Group. MPEG is a family of standards for video encoding and decoding.

MPI message-passing interface. MPI is a standard library for message passing that combines portability with high performance.

MPSoC multi-processor system-on-chip. The multi-processor system-on-chip is a system-on-chip which uses multiple processors, usually targeted for embedded applications.

N

NI network interface. The network interface is a component of a network-on-chip. It is responsible for providing *send/receive* operations for communicating subsystems, encapsulating these requests in packets, capturing and interpreting packets arriving from the NoC, and delivering them to the subsystems.

NoC network-on-chip. The network-on-chip is an interconnection component often used in MPSoC architectures to replace buses. It has many advantages compared to buses, which include high bandwidth, scalability, and power efficiency.

O

OS operating system. An operating system is the software component of a computer system that is responsible for the management and coordination of activities and sharing of the resources of the computer.

P

PIC programmable interrupt controller. The interrupt controller is a hardware component which handles external interrupts by according priorities to cope with external events. Its registers can be programmed by the designer to assign different priorities to different interrupt sources.

R

RAM random access memory. RAM represents a type of memory which can be accessed for read and write operations in any order.

RDL register description languages. The register description language allows specifying and implementing software-accessible hardware registers and memories.

RISC reduced instruction set computer. Reduced instruction set computer represents a CPU design strategy emphasizing the insight that simplified instructions which can be executed very quickly to provide higher performance.

ROM read-only memory. ROM represents a type of memory which can be only accessed for read operations. The data store in a ROM memory cannot be modified.

RTL register transfer level. In an integrated circuit design, register transfer level description is a way of describing the operation of a synchronous digital circuit. In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between hardware registers and the logical operations performed on those signals.

Q

QoS quality of service. Quality of service is a measure of reliability used for data transmission over a shared network.

S

SA system architecture. The system architecture level is the highest abstraction level used in the software design for an MPSoC architecture. It captures both application and mapping specification.

SAD sum of absolute difference. The sum of absolute difference is a widely used, extremely simple video quality metric used for macroblock matching in motion estimation for video compression. It works by taking the absolute value of the difference between each pixel in the original macroblock and the corresponding pixel in the macroblock of the reference frame, which is used for comparison. These differences are summed to create a simple metric of macroblock similarity.

SDF synchronous dataflow. SDF is a model of computation based on dataflow, which detects deadlock and boundedness.

SDG service dependency graph. The service dependency graph represents a unified model to capture the hardware/software interfaces for MPSoC architectures.

SIMD single instruction multiple data. The single instruction multiple data is a technique employed to achieve data-level parallelism, as in a vector processor.

SMP symmetric multi-processing. Symmetric multi-processing involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory.

SoC system-on-chip. System-on-chip refers to integrating all components of a computer or other electronic system into a single integrated circuit (chip).

T

TA transaction accurate. The transaction-accurate level is an abstraction level used to simulate an application running on an operating system.

TLM transaction-level modeling. The transaction-level modeling is a high-level approach of modeling digital systems, where details of the communication among the components are separated from the details of the implementation of functional units or of the communication architecture.

TLP thread-level parallelism. The thread-level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environments. It focuses on distributing the execution processes (threads) across different parallel computing nodes.

V

VA virtual architecture. The virtual architecture level is a more detailed abstraction level compared with the system architecture level. It serves to execute and debug a multi-threaded application.

VCI virtual component interface. Virtual component interface is a standard defined by the Virtual Socket Interface Alliance (VSIA). The overall objective is to obtain a general interface, such that intellectual property (IP), in the shape of virtual components (VCs) of any origin, can be connected to systems on chips of any chip integrator.

VLIW very long instruction word. VLIW is a style of computer architecture that issues multiple instructions or operations per clock cycle, but relies on static scheduling to determine the set of operations that can be performed concurrently.

VP virtual prototype. The virtual prototype level is a cycle-accurate abstraction level used to model hardware platforms. It serves to execute and debug the binary image of a multi-threaded application.

References

1. A386 library, http://a386.nocrew.org/
2. Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. Compilers: Principles, Techniques, and Tools (Second Edition). Addison Wesley, New York, August 2006.
3. HAL, http://www.altera.com/literature/hb/nios2/n2sw_nii5v2_02.pdf
4. AM2000 Processor Array Family, http://www.ambric.com
5. Anderson, A. J. Foundations of Computing Technology. CRC Press, Boca Raton, 1994, ISBN 0412598108
6. Technical documentation of ARM7 and ARM9 processors, AMBA Bus Technical Specification, RealView Compilation Tools Linker and Utilities Guide, Embedded Software Development with ADS v1.2, http://www.arm.com
7. Technical documentation of ARM MaxSim http://www.arm.com
8. Ascia, G., Catania, V. and Palesi, M. Mapping cores on network-on-chip. *International Journal of Computation Intelligence Research*, 1(2), 2005; 109–126.
9. mAgicV VLIW DSP and Diopsis http://www.atmelroma.it
10. Babcock, B., Babu, S., Datar, M., Motwani, P. and Widom, J. Models and issues in data stream systems. *Proceeding of 21st Symposium on Principles of Distributed Computing,* Monterey, California, USA, 2002, 1–16.
11. Bacivarov, I., Bouchhima, A., Yoo, S. and Jerraya, A. A. ChronoSym: A new approach for fast and accurate SoC cosimulation. *International Journal on Embedded Systems (IJES)*, 1(1), 2005; 103–111.
12. Bacivarov, I. Evaluation des performances pour les systèmes embarques heterogenes, multiprocesseurs monopuces. *Thèse de Doctorat INPG*, TIMA Laboratory, 2006 (in French)
13. Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J. and Neema, S. Developing applications using model-driven design environments. *IEEE Computer Society*, 39(2), 2006; 33–40.
14. Barr, M. Memory types. *Embedded Systems Programming*, May 2001; 103–104.
15. Beltrame, G., Sciuto, D., Silvano, C., Paulin, P. and Bensoudane, E. An application mapping methodology and case study for multi-processor on-chip architectures. *Proceeding of VLSI-SoC 2006*, 16–18 October 2006, Nice, France, 146–151.
16. Benini, L. and De Micheli, G. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1), 2002; 70–78.
17. Benini, L. and De Micheli, G. Networks on Chips: Technology and Tools. Morgan Kaufmann, San Francisco, 2006, ISBN-10:0-12-370521-5.
18. Berthet, C. Going mobile: The next horizon for multi-million gate designs in the semi-conductor industry. *Proceeding of DAC*, 10–14 June 2002, New Orleans, USA, 375–378.
19. Berens, F. Algorithm to system-on-chip design flow that leverages system studio and systemC 2.0.1. *The Synopsys Verification Avenue Technical Bulletin* 4, 2, May 2004.
20. Bleuler, S., Laumanns, M., Thiele, L. and Zitzler, E. PISA- A platform and programming language independent interface for search algorithms. *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Volume 2632/2003 of LNCS, Springer, 494–508.

21. Bonaciu, M. Plateforme flexible pour l'exploitation d'algorithmes et d'architectures en vue de réalisation d'application vidéo haute définition sur des architectures multiprocesseurs mono puces. *Thèse de Doctorat INPG*, TIMA Laboratory, 2006 (in French)
22. Bonaciu, M., Bouchhima, A., Youssef, W., Chen, X., Cesario, W. and Jerraya, A. High level architecture exploration for MPEG4 encoder with custom parameters. *Proceeding of ASP-DAC 2006*, January 2006, Yokohoma, Japan.
23. Bouchhima, A., Yoo, S. and Jerraya, A. A. Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model. *Proceeding of ASP-DAC 2004*, January 2004, Yokohama, Japan, 469–474.
24. Bouchhima, A., Chen, X., Petrot, F., Cesario, W. O. and Jerraya, A. A. A Unified HW/SW interface model to remove discontinuities between HW and SW design. *Proceeding of EMSOFT'05*, 18–22 September 2005, New Jersey, USA, 159–163.
25. Bouchhima, A., Bacivarov, I., Youssef, W., Bonaciu, M. and Jerraya, A. A. Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. *Proceeding of ASP-DAC 2005*, 18–21 January 2005, Shanghai, China, 969–972.
26. Buck, J., Ha, S., Lee, E. and Messerschmitt, D. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4, 1994; 155–182.
27. Busonera, G., Carta, S., Marongiu, A. and Raffo, L. Automatic Application Partitioning on FPGA/CPU Systems Based on Detailed Low-Level Information. *Proceeding of the 9th EUROMICRO Conference on Digital System Design*, 30 August – 1 September 2006, Croatia, 265–268.
28. Butenhof, D. R. Programming with POSIX Threads. Addison Wesley, New York, 1997, May, ISBN 0201633922.
29. Incisive Formal Verifier, http://www.cadence.com
30. Open Systems Glossary of Software Engineering Institute, Carnegie Mellon, http://www.sei.cmu.edu/opensystems/glossary.html
31. http://www.agilityds.com/news_and_events/press-release/dec3-2007.htm
32. Cesario, W. O., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Gauthier, L., Diaz-Nava, M. and Jerraya, A. A. Multiprocessor SoC platforms: A component-based design approach. *IEEE Design & Test of Computers*, 19(6), November-December 2002; 52–63.
33. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D. and McDonald, J. Parallel Programming in OpenMP. *Morgan Kaufmann*, San Francisco, 2000; ISBN 9781558606715.October.
34. Chakraborty, S., Kunzli, S., Thiele, L., Herkersdorf, A. and Sagmeister, P. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 41(5), April 2003; 641–665.
35. Cheong, E., Liebman, J., Liu, J. and Zhao, F. TinyGALS: A programming model for event-driven embedded systems. *Proceeding of 2003 ACM Symposium on Applied Computing*, Melbourne, Florida, USA, March 2003, 698–704.
36. Chen, K., Sztipanovits, J. and Neema, S. Toward a semantic anchoring infrastructure for domain-specific modeling languages. *Proceeding of EMSOFT 2005*, 19–22 September 2005, New Jersey, USA, 35–43.
37. Chen, J. W., Kao, C. Y. and Lin, Y. L. Introduction to H.264 advanced video coding. *Proceeding of ASP-DAC 2006*, 24–27 January 2006, Yokohama, Japan, 736–741.
38. Cho, Y., Yoo, S., Choi, K., Zergainoh, N. E. and Jerraya, A. A. Scheduler implementation in MPSoC design. *Proceeding of ASP-DAC 2005*, 18–21 January 2005, Shanghai, China, 151–156.
39. Cooley, J., Lewis, P. and Welch, P. The finite Fourier transform. *IEEE Transaction on Audio Electroacoustics*, 17(2), 1969; 77–85.
40. Coppola, M., Locatelli, R., Maruccia, G., Pieralisi, L. and Scandurra, A. Spidergon: A novel on-chip communication network. *Proceeding of International Symposium on System-on-Chip*, 16–18 November 2004.
41. ConvergenSC. Coware Processor Designer, http://www.coware.com

42. Culler, D., Singh, J. P. and Gupta, A. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, San Francisco, 1998; August.ISBN 1558603433.
43. Desoli, G. et al. A new facility for dynamic control of program execution: DELI. *Proceeding of EMSOFT 2002*, Grenoble, France
44. Diopsis D940, http://www.atmel.com
45. Socrates chip integration platform, http://www.duolog.com
46. eCos, http://www.ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html
47. Erbes, C., Pimentel, A. D., Thompson, M. and Polstra, S. A framework for system-level modeling and simulation of embedded systems architecture. *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 82123, June 2007.
48. Fei, Y. and Ha, N. K. Functional partitioning for low power distributed systems of systems-on-a-chip. *Proceeding of ASP-DAC 2002*, 7–11 January 2002, Bangalore, India
49. Flake, P. and Schirrmeister, F. MPSoC demands system level design automation. *EDA Tech Forum*, 4(1), March 2007; 10–11.
50. Flynn, M. Some computer organization and their effectiveness. *IEEE Transactions on Computers*, C-21(9), 1972; 948–960.
51. FreeRTOS, http://www.freertos.org
52. Freescale DSP, http://www.freescale.com
53. Furtber, S. B. and Bainbridge, J. Future trends in SoC interconnect. *Proceedings of 2005 International Symposium on System-on-Chip*, November 2005, 183–186.
54. Gajski, D. D. SpecC: Specification Language and Design Methodology. Kluwer, Dordrecht, 2000.
55. Gauthier, L., Yoo, S. and Jerraya, A. A. Automatic generation and targeting of application specific operating systems and embedded systems software. *IEEE TCAD Journal*, 20, Nr. 11, November 2001.
56. Gerin, P., Shen, H., Chureau, A., Bouchhima, A. and Jerraya, A. A. Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC. *Proceeding of ASP-DAC 2007*, 390–395.
57. Gerstlauer, A., Shin, D., Domer, R. and Gajski, D. D. System-level communication modelling for network-on-chip synthesis. *Proceeding of ASP-DAC 2005*, 18–21 January 2005, Shanghai, China, 45–48.
58. Gilliers, F., Kordon, F. and Regep, D. A model based development approach for distributed embedded systems. *Proceeding of RISSEF 2002*, 137–151, 2002.
59. Gipper, J. and Dingee, D. Navigating general purpose processor roadmap. *Embedded Computing Design*, 2007.
60. Glass, C. and Ni, L. The turn model for adaptive routing. *Journal of ACM*, 41(5), 1994; 278–287.
61. GNU tools and documentation, http://www.gnu.org
62. Grotker, T., Liao, S., Martin, G. and Swan, S. System Design with SystemC. Kluwer, Dordrecht, 2002, ISBN 1402070721
63. Guerin, X., Popovici, K., Youssef, W., Rousseau, F. and Jerraya, A. Flexible application software generation for heterogeneous multi-processor system-on-chip. *Proceeding of COMPSAC 2007*, 23–27 July 2007, Beijing, China
64. Ha, S., Lee, C., Yi, Y., Kwon, S. and Joo, Y. P. Hardware-software codesign of multimedia embedded systems: The PeaCE. *Proceeding of 12th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, 2006, 207–214.
65. Ha, S. Model-based programming environment of embedded software for MPSoC. *Proceeding of ASP-DAC'07*, 23–26 January 2007, Yokohama, Japan, 330–335.
66. Han, S. I. et al. Buffer memory optimization for video codec application modeled in Simulink. *Proceeding of DAC 2006*, San Francisco, USA, 689–694.
67. Hassan, M. A., Sakanushi, K., Takeuchi, Y. and Imai, M. RTK-spec TRON: A simulation model of an ITRON Based RTOS Kernel in SystemC. *Proceeding of DATE 2005*, 7–11 March 2005, Munich, Germany, 554–559.

68. Hassan, M. A. and Imai, M. A system level modeling methodology for RTOS centric embedded systems. *Proceeding of 14th IFIP VLSI-SoC*, PhD Forum Digest of Papers, 16–18 October 2006, Nice, France, 62–67.
69. Hennessy, J. L. and Patterson, D. A. Computer Architecture: A Quantitative Approach, Third Edition. Printed by Elsevier Science Pte Ltd, Amsterdam, 2003, ISBN 1558605967.
70. Hong, S., Yoo, S., Lee, S., Nam, H. J., Yoo, B. S., Hwang, J., Song, D., Kim, J., Kim, J., Jin, H. S., Choi, K. M., Kong, J. T. and Eo, S. Creation and utilization of a virtual platform for embedded software optimization: An industrial case study. *Proceeding of CODES+ISSS 2006*, Seoul, Korea, 235–240.
71. Hwang, H., Oh, T., Jung, H. and Ha, S. Conversion of reference C code to dataflow model: H.264 Encoder Case Study. *Proceeding of ASP-DAC 2006*, 24–27 January 2006, Yokohama, Japan, 152–157.
72. Kahn, G. The semantics of a simple language for parallel programming. *Information Processing*, 1974; 471–475.
73. Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hannikainen, M., Hamalainen, T. D., Riihimaki, J. and Kuusilinna, K. UML-based multiprocessor SoC design framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2), 2006, 281–320.
74. Kempf, T., Doerper, M., Leupers, R., Ascheid, G., Meyr, H., Kogel, T. and Vanthournout, B. A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. *Proceeding of DATE 2005*, 7–11 March 2005, Munich, Germany.
75. Kienhuis, B., Deprettere, Ed. F., van der Wolf, P. and Vissers, K. A. A methodology to design programmable embedded systems- the Y-Chart approach. *Lectures Notes in Computer Science, Volume 2268, Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation–SAMOS 2002*, Springer, 18–37.
76. Klingauf, W., Gadke, H. and Gunzel, R. TRAIN: A virtual transaction layer architecture for TLM-based HW/SW codesign of synthesizable MPSoC. *Proceeding of DATE 2006*, 6–10 March 2006, Munich, Germany, 1318–1323.
77. Klingauf, W., Gunzel, R. and Schroder, C. Embedded software development on top of transaction-level models. *Proceeding of CODES+ISSS 2007*, 30 September–3 October 2007, Salzburg, Austria, 27–32.
78. de Kock, E. A. et al. Yapi: Application modeling for signal processing systems. *Proceeding of DAC 2000*, USA, 402–405.
79. Kogel, T., Wieferink, A., Meyr, H. and Kroll, A. SystemC based architecture exploration of a 3D graphic processor. *Proceeding of IEEE Workshop on Signal Processing Systems*, 26–28 September 2001, Antwerp, Belgium, 169–176.
80. Kramer, S., Gao, L., Weinstock, J., Leupers, R., Ascheid, G. and Meyr, H. HySim: A Fast Simulation Framework for Embedded Software Development. *Proceeding of CODES+ISSS 2007*, 30 September–5 October 2007, Salzburg, Austria
81. OpenCL standard, http://www.khronos.org/opencl/
82. Kunzli, S., Poletti, F., Benini, L. and Thiele, L. Combining simulation and formal methods for system-level performance analysis. *Proceeding of DATE 2006*, 6–10 March 2006, Munich, Germany, 236–241.
83. Kwon, S., Jung, H. and Ha, S. H.264 decoder algorithm specification and simulation in Simulink and PeaCE. *Proceeding of ISOCC 2004*, Seoul, Korea
84. NicheStack TCP/IP protocol stack, http://www.iniche.com/nichestack.php
85. Intel processors, http://www.intel.com
86. Jerraya, A. and Wolf, W. Hardware-software interface codesign for embedded systems. *Computer*, 38(2), February 2005; 63–69.
87. Jerraya, A., Bouchhima, A. and Petrot, F. Programming models and HW-SW Interfaces abstraction for Multi-Processor SoC. *Proceeding of DAC 2006*, San Francisco, USA, 280–285.
88. Lavenier, D. and Daumas, M. Architectures des ordinateurs. *Technique et Science Informatique*, 25(6), 2006.

89. Lee, E. A. Nesting and unnesting models of computation. *Presentation at the Seventh Biennial Ptolemy Miniconference*, 13 February 2007, Berkeley, CA, USA
90. Lieverse, P., Stefanov, T., van der Wolf, P. and Deprettere, E. System level design with SPADE: an M-JPEG case study. *Proceeding of ICCAD 2001*, 4–8 November 2001, San Jose, USA, 31–38.
91. Liu, J., Lajolo, M. and Sangiovanni-Vincentelli, A. Software timing analysis using HW/SW cosimulation and instruction set simulator. *Proceeding of the 6th International Workshop on Hardware/Software Co-design CODES/CASHE'98*, 15–18 March 1998, Seattle, Washington, 65–69.
92. LLVM Compiler Infrastructure, http://llvm.org/
93. LynxOS, http://www.lynuxworks.com/rtos/
94. Matlab and Simulink, The MathWorks Inc., http://www.mathworks.com
95. Magee, D. P. Matlab extensions for the development, testing and verification of real-time DSP software. *Proceeding of DAC 2005*, Anaheim, USA
96. Martin, G. Overview of the MPSoC design challenge. *Proceeding of DAC 2006*, 24–28 July 2006, San Francisco, USA, 274–279.
97. Mattioli, J., Museux, N., Jourdan, J., Saveant, P. and de Givry, S. A constraint optimization framework for mapping a digital signal processing application onto a parallel architecture. *Proceeding of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001, 701–715.
98. http://www.multicore-association.org/press/080401.htm
99. Meijer, S., Walters, J., Snuijf, D. and Kienhuis, B. Automatic partitioning and mapping of stream-based applications onto the Intel IXP Network Processor. *Proceeding of Workshop on Software & Compilers for Embedded Systems (SCOPES'07)*, Nice, 20 April 2007.
100. CodeWarrior Development tools, http://www.metrowerks.com
101. Meyr, H. Application Specific Processors (ASIP): On design and implementation Efficiency. *Proceeding of SASIM 2006*, Nagoya, Japan
102. de Micheli, G., Ernst, R. and Wolf, W. Readings in Hardware/Software Co-design. Morgan Kaufmann, 2002, ISBN 1558607021.
103. Microchip Technologies Inc., http://www.microchip.com
104. Windows CE, http://www.microsoft.com/windows/embedded
105. Model driven architecture http://www.omg.org/mda/
106. Mohapatra, P. et al. Wormhole routing techniques for directly connected multicomputer systems. *ACM Computing Survey*, 30(3), 1998, 374–410
107. Moraes, F. et al. HERMES: An infrastructure for low area overhead packet-switching networks-on-chip integration. *VLSI Journal*, 38(1), 2004; 69–93.
108. MPI 2.2. Standard, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
109. MPICH, http://www.mcs.anl.gov/research/projects/mpi/mpich2/
110. Nicolescu, G. Specification et validation des systemes heterogenes embarques. *PhD Thesis*, TIMA Laboratory, 2002 (in French)
111. Nikolov, H., Stefanov, T. and Deprettere, E. Multi-processor system design with ESPAM. *Proceeding of CODES+ISSS'06*, 22–25 October 2006, Seoul, Korea, 211–216.
112. newlib, http://sourceware.org/newlib
113. Nexperia http://www.nxp.com
114. Nomadik, http://www.st.com
115. CUDA, http://www.nvidia.com/cuda
116. Object Management Group, CORBA Basics, 2006, http://www.omg.org/gettingstarted/corbafaq.htm
117. OpenMP specification for parallel programming, http://openmp.org/wp/
118. Open SystemC Initiative (OSCI) http://www.systemc.org
119. Oyamada, M., Wagner, F. R., Bonaciu, M., Cesario, W. and Jerraya, A. Software performance estimation in MPSoC design. *Proceeding of ASP-DAC'07*, 23–26 January 2007, Yokohama, Japan, 38–43.

120. Park, S., Olds, W., Shin, K. G. and Wang, S. Integrating virtual execution platform for accurate analysis in distributed real-time control system development. *Proceeding of RTSS 2007*, 3–6 December 2007, Tucson, Arizona, USA
121. Paolucci, P. S., Jerraya, A. A., Leupers, R., Thiele, L. and Vicini, P. SHAPES: a tiled scalable software hardware architecture platform for embedded systems. *Proceeding of CODES+ISSS 2006*, Seoul, Korea, 167–172.
122. Paulin, P., Pilkington, C., Langevin, M., Bensoudane, E., Lyonnard, D., Benny, O., Laviguer, B., Lo, D., Beltrame, G., Gagne, V. and Nicolescu, G. Parallel programming models for a multi-processor SoC platform applied to networking and multimedia. *IEEE Transactions on VLSI Journal*, 14(7), 2006; 667–680.
123. Pazos, N., Maxiaguine, A., Ienne, P. and Leblebici, Y. Parallel modelling paradigm in multimedia applications: Mapping and scheduling onto a multi-processor system-on-chip platform. *Proceedings of the International Global Signal Processing Conference*, Santa Clara, California, USA, September 2004.
124. Popovici, K. and Jerraya, A. A. Simulink based hardware-software codesign flow for heterogeneous MPSoC. *Proceeding of Summer Computer Simulation Conference (SCSC'07)*, 15–18 July 2007, San Diego, USA, 497–504.
125. Popovici, K. and Jerraya, A. Programming models for MPSoC. *Chapter 4 in Model Based Design of Heterogeneous Embedded Systems*. Ed. CRC Press, 2008.
126. Popovici, K., Guerin, X., Rousseau, F., Paolucci, P. S. and Jerraya, A. Platform based software design flow for heterogeneous MPSoC. *ACM Journal: Transactions on Embedded Computing Systems (TECS)*, 7(4), July 2008.
127. Pospiech, F. Hardware dependent Software (HdS). Multiprocessor SoC Aspects. An Introduction. *MPSoC 2003*, 7–11 July 2003, Chamonix, France
128. Pullini, A., Angiolini, F., Meloni, P., Atienza, D., Murali, S., Raffo, L., De Michelli, G. and Benini, L. NoC Design and Implementation in 65 nm Technology. *Proceeding of the 1st Internal Symposium on Networks-on-Chip*, 7–9 May 2007, Princeton, New Jersey, USA, 273–282.
129. Reyneri, L. M., Cucinotta, F., Serra, A. and Lavagno, L. A hardware-software codesign flow and IP library based on Simulink. *Proceeding of the 38th conference on Design Automation*, Las Vegas, United States, 2001, 593–598.
130. Richardson, I. and Sullivan, G. J. H264 and MPEG-4 Video Compression"
131. Rijpkema, E., Goossens, K. and Wielage, P. A router architecture for networks on silicon. *Proceeding of PROGRESS'01*, November 2001, 181–188.
132. Rijpkema, E., Goossens, K. and Radulescu, A. Tradeoffs in the design of a router with both guaranteed and best-effort services for networks on chip. *Proceeding of DATE'03*, March 2003, 350–355.
133. Roane, J. Electronic system level design for embedded systems. *EDA Tech Forum*, 4(1), March 2007; 14–16.
134. Rowson, J. A. Hardware/Software cosimulation. *Proceeding of DAC 1994,* San Diego, USA, 439–440.
135. RTLinux, http://www.fsmlabs.com
136. Qin, W., D'Errico, J. and Zhu, X. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. *Proceeding of CODES+ISSS 2006*, 22–25 October 2006, Seoul, Korea, 193–198.
137. Sarmento, A., Kriaa, L., Grasset, A., Youssef, W., Bouchhima, A., Rousseau, F., Cesario, W. and Jerraya, A. A. Service dependency graph, an efficient model for hardware/software interface modeling and generation for SoC design. *Proceeding of CODES-ISSS 2005*, New York, USA, 18–21 September 2005.
138. Schirner, G., Gertslauer, A. and Domer, R. Multifaced modeling of embedded processors for system level design (Abstract). *Proceeding of ASP-DAC 2007*, 23–26 January 2007, Yokohama, Japan, 384–389.

139. Schirner, G., Gertslauer, A. and Domer, R. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. *Proceeding of ASP-DAC 2008*, 21–24 January 2008, Seoul, Korea
140. Semeria, L. and Ghosh, A. Methodology for hardware/software co-verification in C/C++. *Proceeding of ASP-DAC 2000,* Yokohama, Japan, 405–408.
141. Shapes (Scalable Software Hardware Architecture Platform for Embedded Systems) European Project, http://shapes-p.org
142. Shin, D., Abdi, S. and Gajski, D. D. Automatic generation of bus functional models from transaction level models. *Proceeding of ASP-DAC 2004*, 27–30 January 2004, Yokohama, Japan, 756–758.
143. Shin, D., Gertslauer, A., Peng, J., Domer, R. and Gajski, D. D. Automatic generation of transaction-level models for rapid design space exploration. *Proceeding of CODES+ISSS 2006*, Seoul, Korea, 64–69.
144. Singhai, S., Ko, M. Y., Jinturkar, S., Moudgill, M. and Glossner, J. An Integrated ARM and Multi-core DSP Simulator. *Proceeding of CASES'07*, 30 September–3 October 2007, Salzburg, Austria, 33–37.
145. Skillicorn, D. and Talia, D. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), 1998; 123–169.
146. Spirit IP-XACT, http://www.spiritconsortium.com
147. Synopsys System Studio http://www.synopsys.com
148. Tanenbaum, A. S. Distributed Operating Systems. Prentice-Hall, Englewood Cliffs, 1995, ISBN 0132199084.
149. Tanenbaum, A. S. and Woodhull, A. S. Operating Systems: Design and Implementation. 1997, Prentice-Hall, Englewood Cliffs, ISBN 0136386776
150. Tanenbaum, A. S. Structured Computer Organization. 1999, Prentice-Hall, Englewood Cliffs, ISBN 013219901
151. Target Compiler Technologies, http://www.retarget.com
152. Xtensa processor architecture, XPRES Compiler, http://www.tensilica.com
153. Thies, W., Karczmarek, M. and Amarasinghe, S. StreamIt: a language for streaming applications. *Proceeding of International Conference on Compiler Construction*, April 2002, Grenoble, France
154. Thiele, L., Bacivarov, I., Haid, W. and Huang, K. Mapping applications to tiled multiprocessor systems. *Proceeding of Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, 10–13 July 2007, Bratislava, Slovak Republic, 29–40.
155. Thompson, M., Nikolov, H., Stefanov, T., Pimentel, A. D., Erbas, C., Polstra, S. and Deprettere, E. F. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. *Proceeding of CODES+ISSS 2007*, 30 September–3 October 2007, Salzburg, Austria, 9–14.
156. TI OMAP platform, DSPs, http://www.omap.com
157. Tile64 Processor Family, http://www.tilera.com
158. μITRON4.0 Specification, http://www.tron.org
159. Turley, J. Survey says: Software tools more important than chips. *Embedded Systems Design Journal*, 4-11-2005.
160. Van der Wolf, P. et al. Design and programming of embedded multiprocessors: An interface-centric approach. *Proceeding of CODES+ISSS 2004*, Stockholm, Sweden, 206–217.
161. Vanderperren, Y. and W. Dehaene. From UML/SysML to Matlab/Simulink: Current State and Future Perspectives. *Proceeding of Design Automation and Test in Europe, DATE 2006*, 6–10 March, Munich, Germany, 93–93.
162. Ventroux, N., Blanc, F. and Lavenier, D. A low complex scheduling algorithm for multiprocessor system-on-chip. *Proceeding of Parallel and Distributed Computing and Networks*, 15–17 February 2005, Innsbruck, Austria
163. Verdoolaege, S., Nikolov, H. and Stefanov, T. PN: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 75947.

164. Vergnaud, T., Pautet, L. and Kordon, F. Using the AADL to describe distributed applications from middleware to software components. *Proceeding of Ada-Europe 2005*, York, UK, 20–24 June 2005, 67–78.
165. Kronos model checking tool, http://www-verimag.imag.fr/TEMPORISE/kronos/
166. Sangiovanni-Vincetelli, A. and Martin, G. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test*, 18(6), 2001; 23–33.
167. Sangiovanni-Vincetelli, A. et al. Benefits and challenges for platform-based design. *Proceeding of DAC 2004*, USA
168. VSI Alliance. http://www.vsia.com/
169. Wallace, G. K. The JPEG still picture compression standard. *Communications of the ACM, Special Issue on Digital Multimedia Systems*, 34(4), April 1991, 30–44
170. VxWorks, http://windriver.com/vxworks
171. Wolf, W. High-Performance Embedded Computing. Morgan Kaufmann, San Francisco, 2006
172. h264 open source code, http://www.videolan.org/developers/x264.html
173. Xue, L., Ozturk, O., Li, F., Kandemir, M. and Kolcu, I. Dynamic partitioning of processing and memory resources in embedded MPSoC architectures. *Proceeding of DATE 2006*, 6–10 March 2006, Munich, Germany, 690–695.
174. Yoo, S. and Jerrara, A. A. Introduction to hardware abstraction layers for SoC. *Proceeding of DATE 2003*, 3–7 March 2003, Munich, Germany, 336–337.
175. Youssef, M. W., Yoo, S., Sasongko, A., Paviot, Y. and Jerraya, A. Debugging HW/SW interface for MPSoC: Video Encode System Design Case Study. *Proceeding of DAC 2004*, 7–11 June 2004, San Diego, USA, 908–913.
176. Zhao, Y., Liu, J. and Lee, E. A. A programming model for time-synchronized distributed real-time systems. *Proceeding of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, Bellevue, WA, USA, 2007.

Index