TUCS

Seppo Virtanen

A Framework for Rapid Design and Evaluation of Protocol Processors

Turku Centre *for* Computer Science

TUCS Dissertations
No 55, September 2004

A Framework for Rapid Design and Evaluation of Protocol Processors

Seppo Virtanen

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in the Tauno Nurmela Auditorium at 12 noon on October 22, 2004.

Department of Information Technology
University of Turku
FIN-20014 Turku, Finland

2004

Supervisors

Professor Johan Lilius
Department of Computer Science
Åbo Akademi University
Lemminkäisenkatu 14A
FIN-20520 Turku, Finland

Professor Jouni Isoaho
Department of Information Technology
University of Turku
Lemminkäisenkatu 14A
FIN-20520 Turku, Finland

Reviewers

Professor Jarmo Takala
Institute of Digital and Computer Systems
Tampere University of Technology
Box 553, FIN-33101 Tampere, Finland

Professor Mats Brorsson
Department of Microelectronics and Information Technology
The Royal Institute of Technology - Kungliga Tekniska Högskolan
SE-100 44 Stockholm, Sweden

Opponent

Professor Dake Liu
Computer Engineering Division
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

ISBN 952-12-1410-4
ISSN 1239-1883

Painosalama Oy, 2004
Turku, Finland

To my children Salla, Veikko and Ella

Acknowledgements

It gives me much pleasure to conclude many years of work by expressing my gratitude to the individuals and institutions that have influenced the research presented in this doctoral thesis. First of all, I would like to thank my supervisors Johan Lilius and Jouni Isoaho for their expert guidance, support and insight during my doctoral studies. I wish to extend special thanks to professor Lilius for all his efforts in providing an inspiring and productive research environment.

I also wish to thank professors Jarmo Takala from Tampere University of Technology and Mats Brorsson from the Royal Institute of Technology (Stockholm) for performing detailed reviews of this thesis and for providing constructive comments and accurate suggestions for improvement. By following their recommendations I am now able to deliver a better and more complete presentation of my research work.

I gratefully acknowledge the financial support and working environment provided for the most part of my doctoral studies by the Turku Centre for Computer Science (TUCS). I also wish to thank the Department of Information Technology (University of Turku) for appointing me to a teaching position at the last stages of my studies, thus providing me with further financing as well as an opportunity to participate in educating future researchers. In addition, I wish to express my sincerest gratitude to the HPY research foundation and the Nokia foundation for their generous financial support directed to the research work presented in this thesis. Their grants have been both an enabling and a motivating factor throughout my doctoral studies.

Several of my colleagues in the TUCS community have in one way or another helped and supported me in my research work. I wish to thank PhD student Tero Nurmi for providing physical estimation results and developing the physical estimation model, PhD students Tomi Westerlund and Jani Paakkulainen for providing synthesis results and developing the synthesis model, M.Sc. Tomas Lundström for his implementation of the design tool, and M.Sc. Zhifeng Yang for his IXP1200 implementations of IPv6 routing functions. I also wish to thank PhD student Dragos Truscan as well as Tero Nurmi, Tomi Westerlund and Jani Paakkulainen for co-authoring papers with me. Tero Nurmi also thoroughly proofread this thesis, for which I am very grateful. I also wish to thank all my friends and colleagues in the participating departments of TUCS for creating a pleasant working environment. In this respect I wish to extend special thanks to everyone in the Embedded Systems laboratory as well as to everyone attending the informal Monday morning coffee meetings in the second floor of DataCity. Moreover, I would like to thank M.Sc. Robert Gyllenberg and M.Sc. Timo Virtanen for all the lively discussions we have had over coffee during the past five years, whether scientific or otherwise.

Finally, I wish to thank all my friends and my closest relatives for their friendship and for all the great times we have shared over the years. I especially wish to thank my parents Orvokki and Keijo Virtanen for all the care, support and guidance they have given me throughout my life. I also wish to thank my uncle Jouko Virtanen and my father-in-law Mauno Hyyppä for their genuine interest towards my research work throughout my doctoral studies.

In conclusion, and most of all, I thank my wife Elina as well as my children Salla, Veikko and Ella for all the wholehearted love and all the wonderful moments they have given me in all the different phases of the work that ultimately lead to this thesis.

Turku, September 20, 2004

Seppo Virtanen

Contents

Chapter 1

Introduction

Network hardware design is becoming increasingly challenging as more and more demands are put on network bandwidth and throughput requirements, and on the speed with which new devices must appear on the market. General purpose microprocessors are no longer an appealing alternative for networking hardware due to their lack of optimized execution units for network processing. Using general purpose processors all networking functionality must be implemented in software. This in turn leads to very high CPU clock frequency requirements. General purpose processors that operate in a suitable frequency range are often too expensive, consume too much power or occupy physically too much space in the target system with all their required external circuitry. Also, many general purpose processor features, like floating point arithmetic units (FPUs), can usually not be taken advantage of in networking applications. For these reasons among others, Application-Specific Integrated Circuits (ASICs) have been widely used for networking devices. ASICs can provide a higher processing speed and a lower power consumption at lower clock frequencies than general purpose processors. However, resorting to ASICs requires special hardware design expertise, which usually is not as readily available as general purpose programming skills. ASIC design is a demanding and expensive process, and ASIC time-to-market tends to be long. Also, ASICs are usually minimally programmable and thus need to be redesigned for updated or new network protocols, which makes them inflexible in dynamic market segments.

A potential solution to this problem is designing programmable processors with network-optimized hardware [67]. As suggested in e.g. [19] and [44], the challenge in designing such programmable network or protocol processors is to find an architecture that is as good a compromise as possible between a general purpose processor and a protocol-application-specific custom chip (i.e. an ASIC implementation). An ideal protocol processor would harness both the programmability of general purpose processors and the application-specific hardware optimization of ASICs. While programmable, most of the currently available commercial protocol and network processors are still merely high speed multiprocessors with several parallel general purpose processing elements, lacking true protocol processing hardware optimization. Such processors are e.g. the Intel IXP family [1], IBM PowerNP [5] and the Motorola (C-Port) C-5 [13]. The protocol and network processor development approaches in the academic community have shown some interest into the direction of optimized hardware solutions, and we will look at some of the academic solutions more closely in section 1.4 (Related Work) of this chapter. Still, looking at the current solutions, it can be observed that there is a continuing need to better understand the exact needs of protocol processing in terms of designing and implementing both hardware and software architectures. Thus, in addition to developing new architectures with optimized processing elements, attention must also be paid to application-domain-specific processor design methodologies for protocol processing. Such methodologies should support the designer in analyzing the application domain (and preferably the particular application in question) as well as in exploring and evaluating different hardware/software configurations for performing the target application.

1.1 Objectives

The contribution of this thesis and its author is **the TACO protocol processor design framework**, which aims to solve both of the problems described above by providing:

1. A hardware platform optimized for protocol processing, and

2. A design methodology and a toolset for rapidly specifying, simulating, evaluating and synthesizing protocol processors based on the above hardware platform.

The framework is a part of the TACO research project that was started in TUCS in 1999 by Seppo Virtanen and Johan Lilius (TACO stands for *Tools for Application-specific hardware/software CO-design*). Currently six persons are either directly or indirectly involved in active TACO research. The protocol processor design framework presented in this thesis provides a rapid top-to-bottom protocol processor design and evaluation flow, from application specification to gate-level synthesis, with an optimized (although not necessarily the most optimal) solution obtained as result. The research problems to be solved in order to reach such a design framework can be categorized into two main areas:

1. Issues related to the hardware platform and the simulation, estimation and synthesis models for it. The hardware platform needs to be specified. Different microprocessor architectures need to be analyzed to find a base architecture that is programmable and supports modular library based design. To enable modularity, well-defined interfaces between the execution units and the control structures in the platform are required. The simulation, estimation and synthesis models of the platform need to be parameterizable so that all models for a particular architecture can be rapidly constructed by only specifying the number and connections of the execution units needed. The techniques used for implementing the models depend on the chosen set of development languages and environments. Emphasis should be on obtaining reliable results rapidly from simulations and estimations, and on providing a precise and reliable synthesis model that correctly reflects the characteristics of the simulated model. The potential benefits of object oriented programming techniques in hardware specification and simulation need to be analyzed.

2. Issues related to the design methodology. The initial problem to be solved is how the protocol processing application should be analyzed to determine which tasks to implement in hardware. Performing this application analysis requires familiarity with the protocol processing application domain. Methods need to be specified for refining the initial application into a set of requirements for the hardware and software executing the application. With knowledge of the hardware requirements, the hardware design space needs to be explored to find suitable processor architecture candidates for implementation, and the software needs to be tuned for each candidate. The simulation and estimation models for the hardware platform, as mentioned above, should support the designer in evaluating the processor candidates at this phase. Once a suitable architecture has been found for the target application, the design flow must support generating a logic synthesis model for it.

The suggested solutions provided by the TACO framework to these key problem categories are briefly outlined below, and discussed in detail in later chapters of this thesis.

Hardware Platform The TACO hardware platform for protocol processor design is based on the concept of transport triggered architectures (TTAs) [20, 107]. In TTA processors operations are triggered by programmed data transports. This is contrary to the traditional approach, in which programmed operations cause data transports to occur. A TTA based processor is composed of functional units (FUs) that communicate via an interconnection network of buses. The FUs are connected to the buses through modules called sockets. In TACO processors, each FU performs a specific protocol processing task or operation.

The benefit of a TTA-based hardware platform is its modularity and scalability. Functional units can be added to a processor configuration or they can be refined and changed as long as they provide the same interface to the sockets connecting them to the interconnection network. The TACO hardware platform should be regarded as a base template for protocol processors, instantiated for a specific protocol or a specific family of protocols. The hardware platform facilitates extensive component and module reuse between design projects. The modularity of the platform also enables design automation. To our knowledge, the TACO hardware platform is the first and so far the only approach in which the TTA paradigm is applied to protocol processing. Similarly, to our knowledge the memory organization and access scheme of the TACO hardware platform is unique in comparison to existing TTA implementations.

Simulation, estimation and synthesis models TACO processor architectures can be described using three different models: a simulation model, a physical estimation model and a synthesis model. The simulation model provides signal- and cycle-accurate simulations of processors and their application code. The estimation model produces estimated values for resulting chip area and power consumption on a given technology. A key goal in designing these models has been to obtain accurate and reliable results fast using affordable computers such as standard PCs. This is achieved in part by modeling internal functionality of hardware modules at a higher level of abstraction than the inter-module communication and signaling. The simulation and estimation models are used iteratively to rapidly evaluate the effects of hardware and/or software modifications to performance and physical characteristics. Since the simulation and estimation models can be trusted to provide accurate results, the goal in the synthesis model implementation has been on achieving precise and reliable logic synthesis that correctly reflects the characteristics of the simulated architecture. As can be expected, synthesis takes considerably longer than simulation and estimation, and requires more expensive computer workstations.

The three models are implemented as a component library that contains implementations of functional units, sockets, interconnection buses, and the program dispatch logic. To construct a processor model, the designer chooses modules from the library and specifies their connections. Doing this manually by editing top-level files for each model is an error-prone process due to the large amount of signals and modules to be connected. To alleviate this problem, the TACO framework also includes a graphical processor design tool. The tool automatically generates all necessary files by means of supporting the application programming interfaces (APIs) of the three processor models mentioned above. All this is done without manually modifying the code for the three models, for which reason the tool considerably speeds up instantiation of processor models for a given architecture and eliminates coding errors.

Application analysis The starting point in any TACO protocol processor design project is a high level application description or specification. The development of the application software guides the processor design work: the processing requirements of the target application determine a design space for hardware architectures. Automatic methods for application analysis and automatic hardware requirement derivation based on the analysis are nowadays pursued in another line of TACO research [72]. In the scope of this thesis, the application analysis is performed by the designer manually, following directions and guidelines set by the TACO design methodology. The original application specification is refined and its granularity made finer until a very precise listing of required operations is obtained. The main goal of the application analysis is to find complex and frequently needed processing tasks that should be considered for hardware implementation. Implementing such tasks as hardware modules instead of software blocks considerably reduces the amount of clock cycles needed to execute the entire application. As the number of cycles is reduced, also a decrease in application-level power consumption can be expected.

Design Flow with Design Space Exploration Architectural candidates for implementation are obtained through design space exploration. Design space exploration is the task of evaluating the quality of several designs in terms of hardware architecture, application software, or a combination of both. In the TACO framework, this analysis is performed based on quality metrics specified by the target application; the metrics could typically be a combination of acceptable ranges of values for power consumption, chip size and allowable clock cycle duration. The TACO hardware design space is three-fold; design decisions are needed for the types of hardware blocks to be used, the number of such blocks, and the connections between them. Within this design space, first a set of hardware architectures that are able to perform the required tasks correctly needs to be found. Only a subset of these architectures will be able to function within the timing constraints set by the protocol processing application. Of these, again only a subset is feasible in terms of manufacturing costs, power consumption and circuit size.

Design space exploration in the TACO framework is not completely automated: the designer relies on his experience from previous projects in specifying typically five to ten architecture configurations for detailed evaluation.

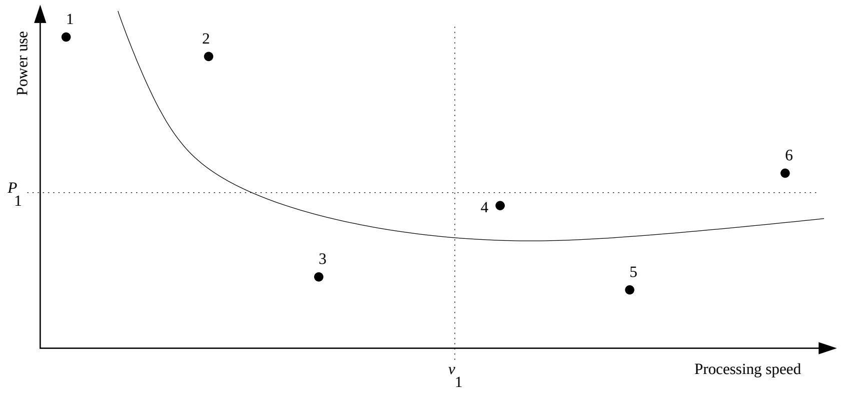


Figure 1.1: Hypothetical example of design space exploration. With shown maximum power (P_1) and minimum throughput (v_1) constraints, only candidates **4** and **5** are acceptable. Of these, candidate **5** has a higher processing speed with a smaller power consumption.

Each configuration is evaluated (simulated and estimated at the system-level) before the next one is specified. TACO simulation and estimation setup is a very well automated process, and the simulator and estimator run very fast (typically in a matter of seconds). This way, results and experiences from each configuration can be taken into account when specifying the next one. With a specific application domain (protocol processing) and a modular hardware platform optimized for it (TACO architecture), the designer is able to construct a set of very good configuration candidates in the iterative process outlined above. The goal of the TACO design space exploration is to find at least one protocol processor configuration that correctly performs the target application and fulfills all given design constraints. If more than one such configuration is found, the one with best results in the most critical constraint(s) is chosen for further evaluation (synthesis and post synthesis simulation).

As an example, Figure 1.1 presents partial results for a hypothetical design space exploration experiment. A set of six architecture candidates has been explored for throughput, area and power use (in addition to correct application functionality). The figure shows throughput and power consumption results for the six candidates; corresponding results should also be obtained for circuit size. In Figure 1.1, P_1 is the upper limit for tolerable power use, and v_1 is the lower limit for tolerable throughput (processing speed) in the target application (v_1 is a function of (1) the processing cycles needed by the candidate to perform the application and (2) the estimated achievable clock speed of the candidate). The curve shows the approximate relation between throughput and power consumption for the target application, derived from the results for the six candidates. As a result of this hypothetical example, two acceptable candidates have been found for the target application. Of these, candidate 5 seems to be most promising as it provides a higher throughput and a lower power consumption than candidate 4. Candidate 5 is not necessarily the most optimal solution for the target application. However, it is a good solution that satisfies all given design constraints.

The techniques and tools provided by and used within the TACO framework reduce the amount of work and time needed in both setting up and carrying out evaluation experiments of different protocol processor hardware configurations, especially in terms of estimating their functionality and performance characteristics at early stages of the design process. A design space exploration experiment like the hypothetical one discussed above could be completed in the course of one day with the TACO framework.

1.2 Overview of Thesis

Chapter 2 of this thesis discusses the protocol processing application domain. The emphasis in chapter 2 is in pointing out similarities in the processing requirements of different protocols. Realizing that such common requirements exist motivates further discussion on designing optimized hardware for the protocol processing application domain.

Chapter 3 discusses the TACO protocol processor hardware platform. The discussion includes an overview of computer architectures, an architectural description of the hardware platform as well as a detailed discussion on the protocol processing functional units available in the TACO module library. A VCI (Virtual Component Interface) [118] compliant interface for Network-on-Chip (NoC) integration is also discussed.

Having covered the hardware platform, Chapter 4 proceeds to discussing the TACO design methodology. The topics covered include the techniques used for application analysis and design space exploration. The simulation, estimation and synthesis models for the hardware platform are also discussed in this chapter. The emphasis in discussing the models is on the SystemC simulation model. The benefits of using SystemC to simulate hardware cycle-accurately at the system-level are discussed, as well as the detected limitations of SystemC in terms of object oriented hardware design. For the Matlab estimation model and the VHDL synthesis model we concentrate the discussion on using these models integrally in the TACO framework: these models are developed in other lines of research, and details of their internal implementations are beyond the scope of this thesis. Finally, a graphical design and evaluation tool for the TACO framework is discussed. The tool is able to generate all three models for a given TACO processor architecture, and also to aid the designer in evaluating simulation and estimation results.

Chapter 5 discusses three case studies on protocol processor design using the tools and methods provided by the TACO framework. The first case, ATM AIS [60, 64] cell processing, focuses on demonstrating design space exploration in the TACO framework. The second one, IPv6 [28] packet forwarding/routing, compares the processing performance of the TACO architecture to the Intel IXP 1200 processor architecture. The last one of the three case studies, IPv6 client operation, demonstrates the component reuse and Network-on-Chip (NoC) integration capabilities of the TACO framework.

Chapter 6 concludes and summarizes this thesis.

1.3 List of Publications

This thesis is based on and extended from the author's contributions to the publications listed below (in chronological order):

1. Seppo Virtanen: **On Communications Protocols and their Characteristics Relevant to Designing Protocol Processing Hardware.** TUCS Technical Report 305, Turku Centre for Computer Science, Finland, September 1999.

2. Seppo Virtanen, Johan Lilius and Tomi Westerlund: **A Processor Architecture for the TACO Protocol Processor Development Framework.** In Proceedings of the 18th IEEE NORCHIP Conference, Turku, Finland, November 2000.

3. Seppo Virtanen and Johan Lilius: **The TACO Protocol Processor Simulation Environment**. In Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES'01), Copenhagen, Denmark, April 2001.

4. Seppo Virtanen, Dragos Truscan and Johan Lilius: **SystemC based Object Oriented System Design**. In Proceedings of the 2001 Forum on Design Languages (FDL'01), Lyon, France, September 2001.

5. Seppo Virtanen, Tomas Lundström and Johan Lilius: **A Design Tool for the TACO Protocol Processor Development Framework.** In Proceedings of the 20th IEEE NORCHIP Conference, Copenhagen, Denmark, November 2002.

6. Seppo Virtanen, Dragos Truscan and Johan Lilius: **TACO IPv6 Router - A Case Study in Protocol Processor Design.** TUCS Technical Report 528, Turku Centre for Computer Science, Finland, April 2003.

7. Seppo Virtanen, Jani Paakkulainen, Tero Nurmi and Jouni Isoaho: **NoC Interface for a Protocol Processor.** In Proceedings of the 21st IEEE NORCHIP Conference, Riga, Latvia, November 2003.

8. Tapani Ahonen, Seppo Virtanen, Juha Kylliäinen, Dragos Truscan, Tuukka Kasanko, David Sigüenza-Tortosa, Tapio Ristimäki, Jani Paakkulainen, Tero Nurmi, Ilkka Saastamoinen, Hannu Isännäinen, Johan Lilius, Jari Nurmi and Jouni Isoaho: **A Brunch from the Coffee Table - Case Study in NoC Platform Design.** Chapter 16 in Jari Nurmi, Hannu Tenhunen, Jouni Isoaho and Axel Jantsch (eds.): Interconnect-Centric Design for Advanced SoC and NoC, Kluwer Academic Publishers, Boston, MA, U.S.A., April 2004.

9. Seppo Virtanen, Tero Nurmi, Jani Paakkulainen and Johan Lilius: **A System-Level Framework for Designing and Evaluating Protocol Processor Architectures**. In International Journal of Embedded Systems 1(1) (Special Issue on Hardware-Software Codesign for SoC), Inderscience publishers, Geneva, Switzerland, 2004 (in press).

1.4 Related Work

In this section we discuss scientific approaches and directions that can be deemed relevant for and related to the research presented in this thesis.

Related Design Methodologies

Behavioral HDL flows. Current ASIC design is often carried out using behavioral hardware description language (HDL) based modeling flows. The work is done in the functional module abstraction level[1] using register-transfer level (RTL) modeling. For example in [18] a methodology is presented for designing a complex communication chip using behavioral VHDL as a starting point. In [99] a methodology called PRCLIB (parameterized re-usable component library) is proposed for constructing SoC's by specifying different hardware configurations of RTL level synthesizable and parameterizable IP blocks from a library, and exploring the design space for such SoC's. These behavioral HDL based flows require that a specification of the target system has already been produced (e.g. as a result of a system specification flow). Also, the component libraries used in these approaches are normally not application domain specific.

Abstract specification based flows. With a continuous trend towards increasing system complexities, abstract specification based modeling flows are constantly gaining popularity. The idea is to start with a very abstract system description and then to incrementally refine this specification to contain more and more architectural and communicational details. The work is

[1]Design abstraction level names used here follow the naming convention defined in [91].

not *necessarily* tied to a single abstraction level or modeling style; the initial specification may e.g. be refined into a system level model using transaction-level models (TLMs) and later into the functional module level using RTL models. In [8] an OCAPI [115] description of system behavior is used as a starting point in a design methodology for an Ethernet packet decoder. In [68] a Y-chart based design methodology and its use for design space exploration is presented. In the Y-chart methodology the performance of a selected architecture is analyzed for a given set of applications. As a result of this analysis the designer receives performance data, based on which decisions and design choices can be made to the architecture. The process is repeated iteratively until a satisfactory architecture for the target set of applications is found. Most of these methodologies yield only the specification of a good hardware architecture candidate, and the actual hardware model of this candidate still needs to be captured in the traditional way using an HDL. A different approach is pursued in the ODETTE project [83], in which the goal is to develop a complete design flow, from specification to synthesis, entirely based on one language. The language to be used is SystemC [71, 85] accompanied with specific limiting rules (called extensions) for the use of object-oriented techniques. These rules, or extensions to base SystemC, are called SystemC-Plus [39]. By making exact rules for use of object orientation direct synthesis from tools supporting SystemC-Plus is expected; *free* use of object oriented techniques in hardware description is generally deemed as a synthesis-disabling factor.

ASIP methodologies. Application-Specific Instruction-set Processor (ASIP) design methodologies aim to find sequences of general purpose operations (like multiply-and-accumulate (MAC) sequences) in the target application, and group these sequences into a hardware implementation [7, 10, 40, 89]. The recurring command sequences that are chosen for hardware implementation are often quite short, usually less than 10 and often only 2-3 general purpose processor commands of length. In ASIP methodologies often a general purpose processing core is enhanced with hardware execution units for the detected command sequences. In [7] this kind of an approach provides a performance increase of no more than 30 % when compared to a general purpose processor designed with equal area and power constraints. If larger, yet frequently occurring operations could be implemented in hardware, greater increases in processing speeds could be expected. Unfortunately, complex application-domain specific operations (e.g. cyclic redundancy checking in the protocol processing application domain) are likely to not be well optimized in these approaches due to the shortness of the detected command sequences. ASIP methodologies are becoming increasingly popular in embedded processor design, especially in the DSP (digital signal processing) community. There are already several commercial tool suites available for ASIP design; Examples of such suites are the LISATek suite from CoWare and the Chess/Checkers suite from Target Compiler Technologies. The LISATek suite uses the LISA language for constructing machine descriptions from which software tools and a synthesis model can be generated [47]. The Chess/Checkers suite includes a retargetable compiler and an instruction-set-simulator generator that operate on an ASIP processor model called the instruction-set graph (ISG) [114]. An interesting new academic ASIP design flow has been discussed in [17]. In this flow a common machine description is used for both a compiler and a core generator. The core generator generates simulation and synthesis models for an architectural template called STA (synchronous transfer architecture). According to [17], STA is a simplification of TTA optimized for the predictable execution environment of DSP.

Also the ***MOVE* framework** [20, 21, 32] should be categorized as an ASIP design process, as stated in [20]. The framework consists of a set of tools for hardware and software synthesis. The tools operate on a parametric TTA architecture template that supports design space exploration. The design process starts with an application written in C or C++, from which combinations of hardware (architecture instances) and software (architecture-optimized code) are derived. The design space exploration tools of the *MOVE* framework are then used to evaluate hundreds of possible solutions (combinations of hardware and software) automatically. The exploration process initially starts with a very large machine configuration, the complexity and connectivity of which is gradually reduced in the exploration process. Upon completion, the exploration process produces a report on each solution candidate's execution time as a function of implementation costs. The parametric TTA template consists of funtional units (FUs) that implement combinations of general purpose operations like addition and multiplication. A designer can also define user modules called special function units (SFUs) for use in the template. However, every time an SFU is added, the backend tools of the *MOVE* framework need to be recompiled and reinstalled to make them aware of a new unit. Finally, the original application code needs to be modified to make use of the operation provided by the new SFU.

A further survey on ASIP methodologies can be found in e.g. [66].

Relation to the TACO framework. The TACO framework can be seen as a combination of the three types of design methodologies and flows outlined above. In the TACO framework, application-specific programmable processors are designed as is the case in **ASIP design**. However, two major differences with traditional ASIP can be identified. First, in the TACO framework the sequences of operations chosen for hardware implementation are considerably larger than in traditional ASIP; in TACO the emphasis is not on detecting recurring general purpose code sequences but in identifying frequently needed domain-specific functions in the target application. Second, TACO processors are not general purpose processors enhanced with special execution units for the detected functionality, but are constructed solely of special execution units and no general purpose processing elements.

The *MOVE* framework requires further discussion. In TACO all work is done in the protocol processing application domain and all execution units are highly optimized for it, whereas in this sense the *MOVE* framework resembles more the traditional ASIP approaches. In terms of complexity, the two frameworks approach the problem of finding a suitable architecture from different directions: in *MOVE*, the initial machine configuration contains all possible modules and execution units, and its complexity is reduced in the exploration process. In TACO, the initial application is analyzed and refined until the exact functions needed for processing the application are found. If the functions are not supported by the FUs in the TACO library, such FUs are created (and stored in the library). The result of this process is a minimal architecture called “virtual processor”, to which complexity is gradually added to meet original design constraints (e.g. to improve the processing performance). In the TACO framework only application-domain specific special function units (SFUs) are used, and they are determined in the application analysis and refinement process. Finally, the *MOVE* framework relies on automatic design space exploration of hundreds of possible solutions, whereas the TACO approach promotes a designer and application analysis driven approach in which (based on our experiences) typically less than ten potential solutions need to be explored. We will return to the *MOVE* framework in Chapter 3 with a discussion on the differences between the TACO platform and the *MOVE* TTA architecture.

There are similarities between **abstract specification based flows** and the TACO framework, as one might expect: in TACO the work is done in a specific problem domain, it being protocol processing. In the TACO flow the hardware architecture is also represented as a specification and simulation model written in a high level language, but only prior to synthesis. And, lastly, performance data such as clock cycle requirements and module utilization is obtained from the high level simulations (in addition to the verification of correct processor functionality). However, the TACO approach is very much library-based and allows extensive component re-use for both simulation and synthesis. Since all modules in our models are not developed from scratch every time, the actual hardware design times in the TACO framework are quite short. Only the TACO VHDL component development and VHDL model maintenance follow more the conventions of **behavioral HDL based flows**.

As a major difference to the mentioned abstract specification based flows, TACO processor models are developed in three different development environments at the same time: there is a model for system-level simulations written in SystemC, a model for estimating physical parameters (e.g. processor area and power consumption) at the system level written in Matlab, and a model for synthesizing architectures written in VHDL. The TACO SystemC simulation model takes freely advantage of standard C++ [105] object oriented techniques like polymorphism and inheritance in addition to standard SystemC data types and functions. This is a key difference to the ODETTE project, in which object orientation is used in SystemC through definition of custom structures (resulting in non-standard SystemC code and in very limited use of the powerful object oriented techniques available in C++). The TACO platform models are highly parameterizable. For this reason, processor models can be constructed by just writing top-level files, in which the number of TACO modules and the connections between them need to be specified. A top-level file is needed for each language for a particular architecture, but the actual processor model implementations do not need to be modified. The top-level files in all three languages can also be automatically generated using the TACO design tool.

Related Protocol Processor Architectures

As mentioned earlier, most commercial protocol processors available today are devices with several high performance general purpose computing elements, possibly accompanied with a general purpose microprocessor core. For example, the Intel IXP1200 processors [57] are built of a StrongArm microprocessor core and six programmable multithreaded "microengines", and the Motorola C-5 processors [13] of a RISC core, 16 programmable "channel processors" and embedded coprocessors for table lookup, buffer memory and queue management. Compared to these, the TACO approach has more in common with application specific processors (ASIP) in that we try to provide hardware implementations of frequently occurring operations. The differences between the TACO approach and ASIP were discussed earlier in this section.

A different commercial approach has been taken with the Xelerator X10q processors [119]. They are data flow processors that operate on protocol data as it flows through the processor. The architecture is organized around a linear array of 200 processing units that form a programmable pipeline. This kind of architecture does not require load-store functionality for protocol processing; the goal is to process the data at network speed.

Something similar to the Xelerator processors has been suggested already some time ago in the academic community; in [45] a programmable network interface accelerator with protocol specific function pages (FPs) is presented. The processor is targeted for accelerating the transfer of incoming data packets to a host processor and one processor per input interface is needed. In this approach the idea is that each layer of the protocol is processed in a separate stage. The data flow is thus organized according to layers in the protocol architecture in question. In our approach something similar could be achieved by dedicating one or more interconnection buses for each layer in the protocol stack. Another approach to direct manipulation of the incoming data flow as described above has been proposed in [44].

In [75] a programmable finite state machine (FSM) architecture optimized for Internet protocol (IP) processing is proposed. The emphasis in the optimization is on the handling of the state table; the processor contains a special unit to handle jumps efficiently. In this approach it was observed that the branch instruction penalties were considerably lower than in general purpose processors executing the same application. In contrast to this architecture, in TACO processors the programmer is responsible for scheduling jumps in the application.

Two recent licentiate theses from Linköping University [43, 80] give detailed surveys of current industrial and academic network and protocol processor architectures and research projects. Also, a recent book [23] presents seven currently abundantly used commercial network processor architectures in much detail. Thus, we will not repeat such surveys and presentations in this thesis, but direct the reader towards the given references.

Chapter 2

Characteristic Functionality in Protocol Processing

In this chapter we argue that different protocols and the functionality needed in their processing exhibit common characteristics that can be taken advantage of in protocol processing hardware design. To support this argument, we analyze six widely used protocols to determine the key functionality needed in their processing. We expect to show that this functionality is not unique to each protocol but is actually characteristic to several of the analyzed protocols. Establishing the existence of such common functionality in the processing of a variety of protocols motivates designing optimized hardware for protocol processing.

2.1 Layered Protocol Architectures

For human beings protocols mean rules of conduct, sets of actions and reactions to be taken in certain situations. Protocols have been specified for a multitude of human-to-human interactions, for example royal and presidential receptions as well as public defenses of academic dissertations. Although sometimes these protocols may seem to be making things more complicated than necessary, they ensure that people put into unfamiliar situations, often with people they do not know, are able to behave in a predictable and correct way. Basically, in the world of networks, protocols are needed for the same purpose: to ensure that networked devices know how to act and react in different communication situations. More formally, protocols specify the syntax and semantics of communication tasks.

In computer networks the communication tasks are usually too complex to be implemented using monolithic protocols. Instead, modular, or *layered*, protocol architectures are preferred. The protocols form a stack of layers, in

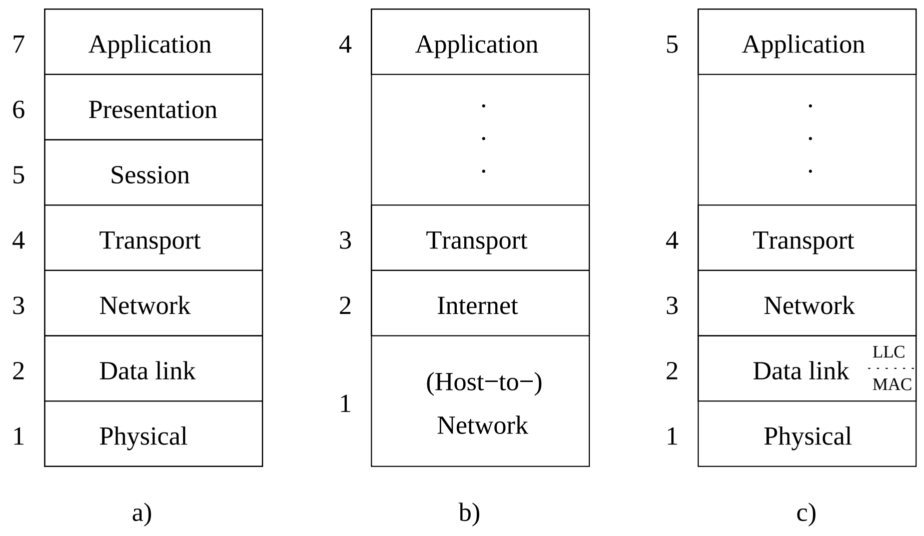


Figure 2.1: a) The ISO OSI reference model, b) the TCP/IP reference model, and c) a hybrid reference model.

which each layer communicates with the one above it and the one below it by passing information through predefined service access points (SAPs), using protocol-specific logical service primitives. As an example, in the ATM protocol the primitive *ATM-DATA.request* is issued by the upper layer through an SAP to request data transfer [64]. The advantage of a layered protocol architecture is that each layer abstracts away some technical functions from the layer above it. So, e.g. a programmer designing a new networking application (working in the topmost layer) does not have to worry about voltage levels and their corresponding logic states in the lowest layer.

Another important benefit of this construction is the possibility to use many different physical mediums and well designed standard protocols for each medium type to perform the same high level task. The International Organization for Standardization (ISO) has defined a protocol stack reference model (i.e. the layers and their duties). The model is called the Open Systems Interconnection (OSI) reference model [25]. The OSI reference model, as seen in Figure 2.1 *a)*, distinguishes the typical tasks needed in communication, but does not specify the actual services and protocols to be used in the different layers. The tasks become more primitive when moving down in the stack and more advanced when moving up. A protocol in one of the layers can be modified as long as the changes have no effect to the operation of the rest of the protocols in the stack. ISO has also defined protocols matching the definitions of the OSI model, but these protocols have not become very popular in practice. The OSI reference model is not the only attempt to standardize

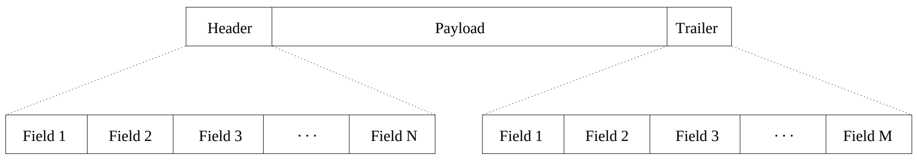


Figure 2.2: A generic Protocol Data Unit (PDU).

the protocol stack; other suggestions include e.g. the TCP/IP reference model [15, 70] and the ATM reference model [64]. The ATM reference model is in use in ATM networks, and is discussed later in this chapter. The TCP/IP reference model is shown in Figure 2.1 *b)*.

The OSI reference model and its layer tasks were defined before the actual OSI protocols were suggested. The TCP/IP reference model, on the other hand, was defined for the existing protocols in the TCP/IP protocol suite[1]. For this reason, the TCP/IP reference model does not accurately define the tasks to be performed in the lowest layer of the model. The OSI reference model (with layers 5 and 6 dropped) has become quite popular in describing networks, but the OSI protocols for the reference model have not been widely adopted. On the other hand, the protocols in layers 2-4 in the TCP/IP reference model are widely used, but the TCP/IP reference model is not usable for describing modern networks. In practice, the protocols used in computer networks nowadays can best be classified using a hybrid reference model such as the one seen in Figure 2.1 *c)*. This approach is suggested in e.g. [110]. In the hybrid model, OSI layers 5 and 6 have been dropped as suggested by the TCP/IP reference model, and on the other hand TCP/IP reference model layers 1 and 2 have been replaced by OSI layers 1-3. Since IEEE[2] local area network (LAN) protocols are very abundantly used in existing networks, and since these protocols are always classified as either LLC (Logical Link Control) or MAC (Medium Access Control) protocols, we extend the hybrid model definition of [110] to include an optional division of the data link layer into the LLC and MAC sublayers as seen in Figure 2.1 *c)*.

Protocols encapsulate the information to be exchanged into Protocol Data Units (PDUs). PDUs are protocol-specific, i.e. PDUs of a certain protocol are understood only by stations supporting the same protocol. The OSI reference model suggests the use of the names APDU for application layer, PPDU for presentation layer, SPDU for session layer, TPDU for transport layer, Packet for network layer, Frame for data link layer and bit for physical layer protocol data units. In practice however, PDUs are often generally referred to as *packets* regardless of the layer in which their exchange takes place. Also,

[1]The TCP/IP protocol suite consists of protocols in layers 2-4 of the TCP/IP ref. model.
[2]IEEE stands for "The Institute of Electrical and Electronics Engineers, Inc."

most protocol specifications define a name to be used for their PDUs. For example, ATM has cells, Ethernet has frames, IP has datagrams, TCP has segments, etc.

As seen in Figure 2.2, PDUs are constructed of a header, a payload and a trailer. The payload is the actual data being carried. The header and trailer contain protocol-specific control information organized into fields defined in the protocol specification. The fields may contain information such as protocol version, payload length, traffic class, receiving station address, error-checking checksum etc. Many protocols do not have a trailer part in their PDUs, but carry all the necessary control information in the header. For example, TCP, IP and ATM PDUs do not have trailers. A trailer is required e.g. in Ethernet frames.

2.2 Layer Characteristics

Application layer protocols are the ones closest to the computer users. These protocols allow applications to exchange data, and often require user input before the exchange. Typical applications include file transfer, E-mail and web browsing. For example, requesting a web page using a web browser causes an HTTP (Hyper-Text Transfer Protocol) [33] request message to be sent. The server responds to this message with another HTTP message containing the target web page[3]. As another example, when sending electronic mail the user actually manually fills in header fields of an SMTP (Simple Mail Transfer Protocol) [88] message by entering an E-mail address into the "To:" field and a title into the "Subject:" field. Once the mail is written and the user tells the E-mail application to send it, the application constructs an SMTP message from the user input and application settings (e.g. the "From:" field containing the user's own E-mail address is automatically inserted into the header), and sends the message as specified in the SMTP protocol.

The most important task for the transport layer is to provide a reliable data exchange connection between two networked devices. Transport layer protocols communicate in an end-to-end fashion, i.e. the sending and receiving protocols communicate directly with each other. The transport layer regards the network below it as a pipe, into which data is inserted in one end and consumed in the other end. Protocols of the lower layers need to concern themselves with intermediate stations like routers as part of their operating environment. Depending on the transport layer protocol in question, tasks like end-to-end flow control, data segmentation and connection multiplexing may be included in the protocol. An obvious example of transport layer protocols

[3]Actually, only the main text of the web page. Images etc. are retrieved in subsequent HTTP requests and responses.

is the TCP protocol (Transmission Control Protocol) [50]. It provides reliable connection-oriented communication between two remote devices. Of the tasks mentioned above, TCP does application data segmenting and uses flow control mechanisms to fight network congestion.

The network layer is the first chained layer in the protocol stack: communication is not anymore taken care of by just the sending and receiving hosts as in the above layers, but it is dependent of neighboring intermediate stations. The most important task of the network layer is to interconnect different kinds of underlying physical networks. The network layer takes care of issues like routing and traffic statistics. As an example, the Internet Protocol (IP) [49, 51] is used to form the Internet out of a huge selection of underlying subnetworks. IP is responsible for e.g. routing packets through multiple routers and subnetworks from the source station to the destination.

The data link layer presents the underlying physical network (i.e. cable or wireless transmission channel) as an error-detecting reliable transmission line to the network layer. The data link layer protocols often utilize error detection and error correcting mechanisms, and perform retransmissions when necessary in error situations. As mentioned earlier, in practice the data link layer is often seen consisting of two sublayers: the Logical Link Control (LLC) and Medium Access Control (MAC) sublayers. Of these, the MAC sublayer arbitrates access rights to the physical network, and the LLC sublayer manages the logical aspects of the connection (e.g. flow control, error correction). Commonly used data link layer protocols include the IEEE 802.2 LLC protocol [52] and the IEEE 802.3 MAC protocol [54] used in Ethernets.

The physical layer deals with the physical (mechanical, electrical, optical) characteristics of the transmission medium and is mostly concerned with correct interpretations of signals to ones and zeros. For example, which voltage represents a one and which a zero, and what is the duration of one bit in the incoming signal. Examples of physical layer protocols are the Synchronous Digital Hierarchy (SDH) [61, 62, 63] and Synchronous Optical NETwork (SONET) [6] used in telephone trunk networks.

2.3 Analysis of Protocols and Protocol Processing Tasks

In order to develop devices for the protocol processing application domain, careful studies on protocols and protocol processing applications are needed. Emphasis should be on finding functionality that varies very little or not at all from one protocol to another. As a starting point, Jantsch et al. have identified three typical characteristics of protocol processing in [67]: (1) pattern matching and replacement in bitstrings (especially in frame or cell header analysis), (2) control dominated operation (large finite state machines and nested if-then-else and case structures) and (3) the need for irregular memory accesses (managing tables and buffers of various sizes).

We agree with these findings and proceed to search for additional characteristic functionality that could be taken advantage of in protocol processing hardware design. The focus in the following discussion is on analyzing protocols that can be regarded as layer 1-3 protocols in the OSI reference model (Figure 2.1). The protocols in these layers are not end-to-end protocols but require intermediate stations (e.g. repeaters, bridges, switches, routers etc.) between the source and destination devices. Thus, these protocols present a clear need for application-specific hardware systems in addition to application software, whereas the end-to-end protocols in layers 4 and above are often completely implemented as software running on networked workstations.

The layer 1 protocol reviewed in this chapter is the Synchronous Digital Hierarchy (SDH) [61, 62, 63]. The IEEE 802.3 and IEEE 802.11 MAC layer specifications [53, 54] as well as the Asynchronous Transfer Mode (ATM) [59] are covered as layer 2 protocols[4]. From layer 3, the Internet Protocol (IP) [49, 51] and the Internet Protocol version 6 (IPv6) [28] are reviewed. The decision to review these protocols has been made based on the fact that five of these protocols are currently in widespread use in networks around the world, and also the remaining one (IPv6) is expected to be so in the future.

2.3.1 Synchronous Digital Hierarchy

The number of long distance telephone operators increased heavily in the 1980's. Most operators utilized their own optical TDM (Time-Division Multiplexed) systems to connect customers from local telephone companies to long distance networks. At the end of that decade, the CCITT (nowadays known as International Telecommunication Union, or ITU) released recommendations G.707 [62], G.708 [61] and G.709 [63] describing a high capacity digital transmission network called Synchronous Digital Hierarchy (SDH). These recommendations have been constantly updated since: the most recent update is from the year 2003.

A parallel ANSI (American National Standards Institute) standard was defined earlier in North America, describing the Synchronous Optical NETwork (SONET)[6]. The main goal for designing SDH and SONET was to define a common standard for making the networks of different long distance carriers compatible. In addition to a multitude of national carriers, the European, American and Japanese digital telephone systems needed to be able to transfer calls between each other. All these systems utilized 64 kbps

[4]ATM does not directly map onto the OSI reference model; actually it exhibits characteristics both from OSI layer 2 and OSI layer 3.

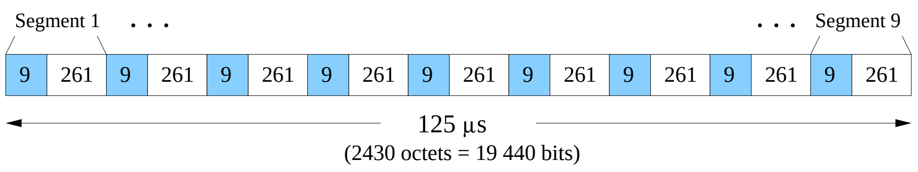


Figure 2.3: Overview of the SDH STM-1 frame. The grayed boxes represent the SDH overhead (SDH header) and white boxes represent the SDH payload. Numbers indicate byte (octet) count.

PCM (Pulse Code Modulation) channels but the multiplexing methods used in these systems were incompatible. The multiplexing in SDH and SONET was decided to be TDM-based so that the entire bandwidth is devoted to one channel that contains time slots for various subchannels.

The standards also specified support for operations, administration and maintenance (OAM) functionality. OAM was not well enough supported in existing systems in the 1980's. The SONET specification also includes support for electrical transmission lines in addition to optical ones.

The differences between SDH and SONET are minor, although because of them only a subset of SDH is compatible with SONET and the other way round. So with certain options, communication and information transfer between SONET and SDH networks is possible. Specifically, OAM functionality is not supported between SONET and SDH networks, and thus a SONET network can not be administered directly from an SDH network and vice versa.

The transmission speeds for SDH and SONET are the same except for the lowest speeds of SONET: the basic transmission unit of SDH, the STM-1 (synchronous transport module level one) frame, is sent at 155.52 Mbps. The basic transmission units of SONET, the STS-1 frame (synchronous transport signal level one, electrical) and the OC-1 frame (Optical Carrier level one, optical), are sent at 51.84 Mbps. Actually, the STM-1 frame structure is very similar to the structure obtained by multiplexing three SONET OC-1 or STS-1 frames. Thus the SONET OC-3 and STS-3 are similar (in content) to the SDH STM-1 frame, and the transmission speeds for STM-1 and OC-3/STS-3 are the same. The standards define transmission speeds up to 2.4 Gbps.

In addition to carrying telephone calls, SDH can also be used to carry for example ATM cells: the STM-1 (STS-3/OC-3 in SONET) frames are sent at the speed of 155.52 Mbps, and the STM-4 (STS-12/OC-12 in SONET) units at 622.08 Mbps. These are also standardized ATM transmission speeds, so SDH and SONET trunks can be used to transport ATM. SDH is a synchronous protocol; the sender and receiver share a clock, and frames are exchanged every 125 μs (8000 times per second) even if there is no data to be sent. In digital form a single telephone call requires 64 kbps of bandwidth (PCM sampling,

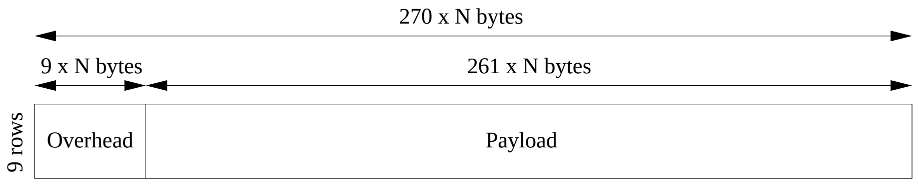


Figure 2.4: General structure of SDH STM-N frames. An STM-N frame is a container for time-division multiplexed STM-1 frames. For this reason, the overhead and the payload are N times longer than in an STM-1 frame.

8 bits per sample at 8000 samples per second). The available bandwidth for telephone calls using SDH STM-1 frames is 150.336 Mbps (the rest of the bandwidth is used for section overheads). Thus, an STM-1 frame is able to transport a maximum of 2349 telephone calls at a time.

SDH frame format

SDH data is structured into units called frames. The basic STM-1 frame consists of 2430 octets[5] and is divided into nine equal length segments, as shown in Figure 2.3. Each segment consists of a 9-octet header part (called the segment overhead) and a 261-octet payload part. In SDH there is no single frame header, but the information is distributed evenly throughout the frame in segment overheads.

The higher order SDH frames (e.g. STM-3, ..., STM-N) are formed by synchronously multiplexing lower order frames. Higher order frames are often represented as a 9-row stack of segments as shown in Figure 2.4. In this representation, it still must be realized that the frame is sent one row at a time, from left to right similarly as the one-row STM-1 frame show in Figure 2.3. As with STM-1 frames, the STM-N frames are also formed of 9 consecutive overhead and payload parts. The lengths of these parts (and hence the required transmission speed) are multiplied with N when compared to the STM-1 frame; thus, an STM-N frame is still sent every 125 μs, but it contains N times more data than an STM-1 frame. An example of multiplexing lower-order frames to higher-order frames is given in Figure 2.5. In the example, STM-1 frames travel from sender to receiver through two faster networks. On the way, the frames are first multiplexed into STM-3 frames and later into an STM-12 frame, and de-multiplexed accordingly.

[5]An octet is an eight-bit byte.

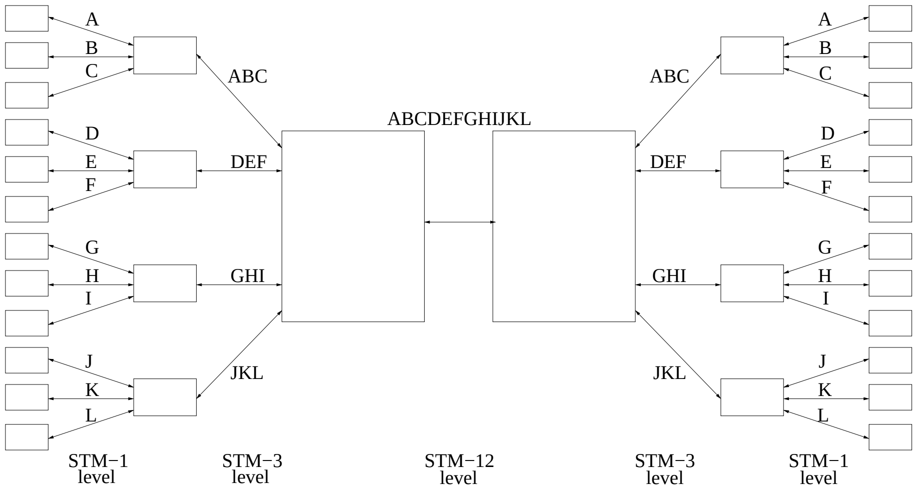


Figure 2.5: An example SDH network. Single letters indicate STM-1 frames being transmitted. Three-letter combinations indicate STM-1 frames multiplexed into STM-3 frames. ABCDEFGHIJKL indicates four STM-3 frames multiplexed into an STM-12 frame. STM-1 frames travel at 155.52 Mbps, STM-3 frames at 466.56 Mbps, and the STM-12 frame travels at 1866.24 Mbps. Boxes indicate network nodes capable of multiplexing and demultiplexing STM-N frames.

SDH processing characteristics

In terms of required processing in SONET and SDH systems, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **Processing speed.** As described previously, SONET and SDH are very high speed wide area network systems. Their transmission speeds currently vary from 52 Mbps to 2.4 Gbps. Because of these high speeds the hardware used for operating these networks must be able to handle incoming information at speeds of up to 2.4 Gbps. This speed requirement is not variable in an SDH system; since the incoming data rate is always constant, there are no temporary peak-rate data bursts that could be dealt with by buffering. The execution units, interfaces and internal transport buses of an SDH-processing device must be able to provide adequate performance, which sets clear timing constraints on both software and hardware design.

- **Segment overhead analysis.** The control and administration information in SDH systems is conveyed as overhead bytes. The overheads are not transmitted sequentially, but are distributed e.g. in the basic

STM-1 frame in 9-octet series that appear every 270 octets in a 2430-octet frame. Because of this distributed nature, a system processing SDH frames must be able to find the distributed overhead bytes in the incoming bit stream, and process them. Processing the overheads requires data analysis and manipulation at the bit level. A need for **Boolean evaluation** can also be seen in overhead analysis. Finally, since the SDH overheads may only contain values greater than or equal to zero, it becomes obvious that **unsigned arithmetic** is adequate for SDH overhead processing.

- **Exact timing.** As it is a synchronous and time-division multiplexed transmission system, SDH requires very exact timing (and hence timing hardware units) in sending, receiving, assembling and disassembling frames. Also, clock synchronization between the sender and the receiver is required.

- **Framing and deframing.** In SDH, the protocol header (overhead) information is distributed across the frame to be sent. Data framing in SDH is thus more complicated than in protocols that place their control information entirely in the beginning and optionally in the end of the frame. When outputting a frame, the overhead information to be sent must be temporarily stored in a **buffer** or **memory block**, and parts of it outputted at exactly specified times into the outgoing bitstream. **Counter functions** are also needed to calculate the correct amount of payload bytes to be written before writing overhead bytes again, and vice versa.

2.3.2 IEEE 802.3: Medium Access in the Ethernet

Most modern cabled local area networks are based on the IEEE 802.3 Medium Access Control (MAC) standard [54]. Often these networks are referred to as "Ethernets". The original Ethernet was suggested by Metcalfe et al. in 1976 [78]. The term "ether" was used to express that the data transmission space between the networked stations is filled with a shared transmission medium (i.e. a single cable that runs from one station to another)[6]. The first IEEE standard describing Ethernets (i.e. the aforementioned 802.3 standard) was published in 1983, and at the time of writing, the most recent revision for the base standard is from 2002. The standard and its amendments currently

[6]Until the end of the 19th century it was commonly accepted that all apparently empty parts of space were filled with an invisible material substance called the "ether". Among other things, the "ether" was believed to make different kinds of waves able to travel through space. For example James Clerk Maxwell (1831-1879), who is generally regarded as one of the greatest physicists ever, strongly relied on the concept of "ether" in his research.

define access rules for networks based on coaxial cable, twisted pair cable and optical fiber, and network speeds ranging from 1 Mbps to 10 Gbps.

The original shared medium access scheme of the Ethernet is still the basic mode of operation and a required capability for all 802.3 compliant devices. All stations expect to be networked with a single cable running through all the stations in the same network segment, and thus they are prepared to address situations in which two stations attempt to send data at the same time (i.e. data collisions occur). The 802.3 standard calls this medium access method CSMA/CD (Carrier Sense Multiple Access with Collision Detection). In the coaxial cable implementations, stations are organized in this kind of bus topology both physically and logically (the cable is the "ether"). This is different from the twisted pair (TP) implementations, in which the stations physically form a star topology: the cables are drawn from each station to a centralized device called *hub*. In such a case, the devices still form a logical bus topology with the circuit board of the hub acting as the "ether".

In the coaxial cable and hub based TP implementations, the entire bandwidth of the network is shared by all stations in the same network segment. For example, let us assume a 100 Mbps hub based TP network with 10 stations. If five of the stations have a need to send data at the same time, the available bandwidth for each of the stations is only 20 Mbps. To alleviate this, many TP networks nowadays use intelligent switching hubs instead of basic hubs. Switching hubs are able to buffer data and transport it directly between two stations without letting other stations in the same network segment see the data. This approach has obvious benefits both in terms of security and in terms of performance; stations are not able to eavesdrop on data not addressed to them, and all stations in the segment have the maximum bandwidth available at all times. Also, no collisions occur in the network and thus time is not wasted in retransmission attempts. In this approach, physically the network implements a star topology, but logically it implements a direct connection (per transfer) topology. The stations still consider it a logical bus topology and are prepared for data collisions and retransmissions.

The 802.3 CSMA/CD Access Method

The key responsibility of the 802.3 MAC protocol is to define medium access rules for stations in the same network segment. 802.3 does not utilize a static multiplexing scheme, but instead the entire bandwidth is dynamically allocated to any station requiring network access. This decision is warranted due to the bursty nature of data transfers in computer networks; stations do not produce constant-speed bitstreams but short bursts of data with silence between the bursts. As mentioned earlier, the medium is accessed using the CSMA/CD method: if a station is ready to send a frame, it first senses

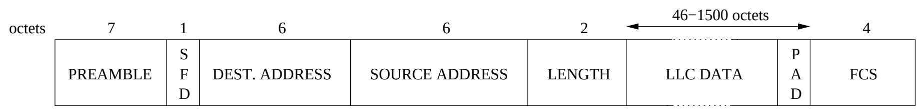


Figure 2.6: Structure of the IEEE 802.3 MAC layer frame. SFD = Start of Frame Delimiter, LLC = Logical Link Control, FCS = Frame Check Sequence.

the medium (channel) to determine whether it is available. If the medium is available, the station may immediately proceed to sending the outgoing frame. If the medium is busy (i.e. some other station is currently sending a frame), the station waits for the medium to become free before sensing the medium again.

It is evident that using such a medium access scheme frame collisions will still occur: due to propagation delays[7] it is possible that a station on a far edge of the network segment may sense the medium to be free while another station is already sending a frame. The frame collision is detected by one of the sending stations and a special 48-bit noise burst is generated onto the network to inform other stations of a collision. The senders of the collided frames stop transmitting their frames immediately, and start executing the so called **binary exponential backoff algorithm**. The binary exponential backoff algorithm determines a random period of time the station sending a colliding frame has to wait before another sending attempt. With consecutive collisions, the maximum limit for randomization increases exponentially. After sixteen consecutive collisions for the same frame, the MAC layer reports an error message to the upper layers in the protocol stack and refrains from sending the colliding frame.

802.3 MAC Frame Format

The 802.3 MAC frame is formed of eight fields, as seen in Figure 2.6. The frame begins with a 7 octet **preamble** used for synchronization; the octets of the preamble form a square waveform with alternating ones and zeros. The preamble is followed by the one-octet **start of frame delimiter (SFD)**. The SFD continues the synchronization waveform, but ends with two consecutive ones. As the name suggests, SFD (in accordance with the preamble) is used to determine the starting point of the frame in the incoming bitstream.

802.3 MAC addresses are device specific (fixed) global addresses. Network device manufacturers obtain the addresses for their devices directly from IEEE. Thus, no two devices can have the same address and hence the MAC address can be used in e.g. automatic network configuration. The **destina-**

[7]Propagation delay is the time needed by signals to travel, or *propagate,* through the network from the sender to the receiver.

tion address field of the header is analyzed by all stations in the segment (assuming the segment is not switched). If a station finds its own MAC address in the destination address field, it stores the frame for further processing. Otherwise, the frame is discarded. The **source address** is used to identify the sending station.

The minimum length of a 802.3 MAC frame is 72 octets, or 576 bits. This length and the bit rate of the network determine the maximum segment length of 802.3 based local area networks; transmitting the frame must take longer than twice the propagation delay between two stations on opposite edges of the network. This way it is impossible for a station to finish sending a frame prior to detecting a collision (i.e. sending is still in progress when the first bits of the colliding frame arrive). If the frames were shorter, collisions might not be detected. For example, the maximum allowable segment length in a 10 Mbps 802.3 based LAN is 2.5 km using four repeaters. This corresponds to a value of 51.2 μs for two times the propagation delay. This is exactly 512 bits in a 10 Mbps network, which is equal to the minimum length of the 802.3 MAC frame without the preamble and the SFD. Thus, if there is not enough data in the **LLC data** field, the frame must be padded to reach the minimum frame length of 72 octets using the **pad** field[8]. The **length** field determines the length of the actual upper layer data (i.e. the length of the LLC data field) so that the data can be separated from the padded content.

The 4-octet, or 32-bit, **frame check sequence (FCS)** field contains a checksum calculated for the entire 802.3 MAC frame. The checksum is used to detect any errors in transmission. The calculation is carried out using cyclic redundancy checking (CRC). The generator polynomial used for calculating the 32-bit CRC checksum is defined in the 802.3 standard[9]. The same generator is used for all IEEE 802 family protocols that require a 32-bit checksum.

The **LLC payload** contains the upper layer data being transmitted. The maximum length of the upper layer data is 1500 octets, and thus the maximum length of an 802.3 MAC frame is 1526 octets including the **preamble** and the **SFD**.

802.3 MAC Processing Characteristics

In terms of required processing in local area networks running IEEE 802.3 MAC, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **High speed operation.** The emerging 1 and 10 Gigabit Ethernets require very high processing speeds, which naturally places great de-

[8]The **pad** field content is not specified; it can be any combination of ones and zeros.

[9]$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$

mands on hardware intended for processing these protocols. High bit rates in the data link layer also provide high speed transmission capability for processes operating in higher layers of the protocol stack; for example, a 10 Gigabit Ethernet is theoretically capable of transporting IP datagrams at about the same rate.

- **Random number generation.** Random numbers are needed by the binary exponential backoff algorithm to determine the lengths of wait periods after frame collisions.

- **Timer functions.** Timer functions are also needed by the binary exponential backoff algorithm. After the length of the wait period has been calculated, time needs to be measured to actually implement the wait.

- **Counter functions.** The binary exponential backoff algorithm also requires the ability to count the number of consecutive collisions. The upper limit for randomization is dependent on the number of consecutive collisions occurring for the same frame.

- **Frame header analysis** In basic 802.3 operation, all stations see all the frames in the network segment. After synchronizing to the bitstream, each station needs to determine whether an incoming frame belongs to it or not. This is done by comparing the destination address field in the header to the stations's hardware address. If the frame belongs to a station, the header needs to be analyzed further to e.g. determine the payload length. These tasks require operations like **bitwise manipulation, Boolean comparisons** and **counting functions.** Also, it can be oserved that all 802.3 MAC header fields may only contain values greater than or equal to zero. Moreover, implementing the CSMA/CD access method does not require the use of negative values. Thus, 802.3 MAC processing can be carried out resorting only to **unsigned arithmetic**.

- **Checksum calculation.** Each 802.3 frame contains a 32-bit CRC checksum. The checksum needs to be calculated both in sending and in receiving stations.

- **Data buffering.** In basic 802.3 operation the most recently sent frame needs to be buffered by the sending station to make consequent retransmission attempts after collisions possible. In intelligent Ethernet switches more extensive buffering is needed; each port in the switch needs to be able to buffer several frames to ensure maximal throughput without frame collisions.

2.3.3 IEEE 802.11: Medium Access in Wireless LANs

As portable and hand held computing devices have become increasingly popular, the need for connecting them easily to different types of networks has risen. From the user's point of view, the easiest way of connecting such a device to a network is to use a self-configuring wireless connection: the user would not have to connect any cables to the portable or hand held device, and the device would automatically sense the presence of an available wireless network. The user would be able to use the mobile unit anywhere within the transmission range of the wireless network. However, it is evident that wireless networking does bring forth new problems in data communication, such as increased noise and interference, overlapping wireless network segments and an increased need for securing the connection.

The IEEE 802.11 MAC standard [53] for wireless local area networks was first published in 1997 for 1 and 2 Mbps transmission speeds. The standard originally specified operation for infrared wavelengths in the range of 850..950 nm, and for both direct-sequence and frequency-hopping spread spectrum radio in the unlicensed industrial, scientific and medical (ISM) frequency band at 2.4 GHz (which is also used by e.g. microwave ovens and Bluetooth[10] devices). The most recent revision of the base standard was published in 1999. The base standard has received several supplements and amendments since the initial publication. The key supplements and amendments, as described in the list below, specify operation in higher transmission speeds as well as in additional radio frequency bands:

- **802.11a.** This amendment to the 802.11 standard was published in 1999. It defines a high speed (up to 54 Mbps) physical layer for the unlicensed 5 GHz frequency band. 802.11a is not directly compatible with 802.11 systems operating in the 2.4 GHz ISM band. On the other hand, there is less interfering radio traffic in the 5 GHz band, and more transmission channels available for users.

- **802.11b.** This supplement to the 802.11 standard, also known as *Wi-Fi*, was also published in 1999 (and updated in 2001). As IEEE expresses it, this supplement defines a "higher speed" physical layer extension to the 2.4 GHz ISM frequency band. Devices supporting 802.11b have a maximum transmission speed of 11 Mbps, which is 5-10 times more than originally specified in the 802.11 standard for the ISM frequency range. *Wi-Fi* has gained a lot of popularity over 802.11a (in part due to more cost-efficient RF implementations in the 2.4 GHz band), and currently most new devices with WLAN support are *Wi-Fi* compatible.

[10]A personal-area network (PAN) specification for robust short range communication between (portable) devices. Named after Danish viking ruler Harald Blåtand (910-986).

- **802.11g.** This amendment to the 802.11 standard, also known as *Wireless-G*, has only recently been ratified by IEEE [16]. It is actually an extension to the 802.11b supplement, and it defines a high speed (up to 54 Mbps) physical layer in the 2.4 GHz ISM band while maintaining downwards compatibility with 802.11b (and base 802.11 ISM part). Although this amendment was officially published only quite recently, details of it were mostly settled already by the end of 2002. For this reason, device manufacturers have already for some time been able to manufacture 802.11g compatible, or *Wireless-G*, devices and make them available on the market.

Wireless devices can operate in either ad hoc[11] mode or in infrastructure mode. By definition, an ad hoc network is a spontaneously formed temporary and small network and the term is usually used in the context of local or personal area networks. An ad hoc WLAN network is formed by devices using compatible WLAN interfaces, and it is disassembled when devices sign out of it. Ad hoc networks are useful in e.g. exchanging data between wireless devices located in the same room without cables.

The infrastructure mode requires the use of an access point (base station). The access point is connected (by cable) to an existing infrastructure network (e.g. a campus area LAN or a corporate LAN). Wireless devices have access to the infrastructure network through the access point. In infrastructure mode, the access point acts as a hub for connecting wireless devices to the wired network. Naturally, more than one access point can be connected to the same infrastructure network, and a wireless device can be moved from the range of one access point to the range of another one and still remain connected to the same infrastructure network.

The 802.11 CSMA/CA Access Method

The medium access method of 802.11 wireless LANs is called CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance). It differs from the CSMA/CD method used in IEEE 802.3 (Ethernet) LANs in the sense that it focuses on avoiding frame collisions (CA, collision avoidance) instead of detecting them (CD, collision detection). In the basic CSMA/CA operation in 802.11 a host senses the transmission medium for activity. If the medium is idle, the host waits a short period of time called interframe space (IFS) plus a random number of backoff periods called "slots". Then, the station senses the medium again[12]. If the medium is still idle, the host may start transmitting data. If the medium is busy, the host must back off for a random time

[11] *Lat.* Ad hoc = for this, for this purpose.

[12] There are actually three different IFS times, and the one chosen for use depends on the mode of operation.

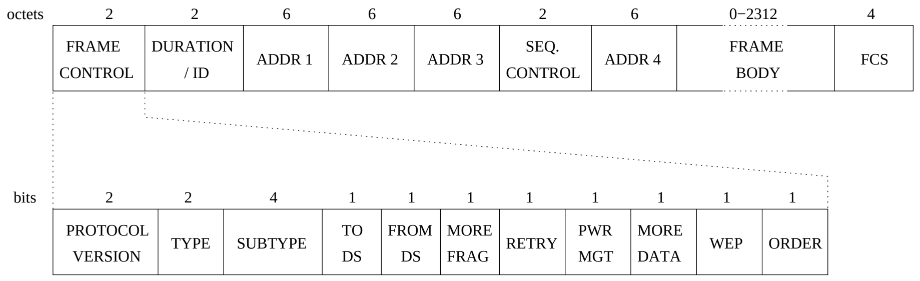


Figure 2.7: Structure of the IEEE 802.11 MAC layer frame. SFD = Start of Frame Delimiter, LLC = Logical Link Control, FCS = Frame Check Sequence.

before attempting to initialize transmission again. This mode of operation, where stations contend of medium access, is called **distributed coordination function (DCF)** in 802.11 networks. If the network is equipped with an access point (a base station), also contention-free access may be provided. This optional contention-free mode is called **point coordination function (PCF)**. As the name suggests, the access point grants individual stations medium access during PCF operation.

As mentioned, the DCF mode is the basic mode of operation in an 802.11 network. If PCF is not used, the network segment is always in DCF mode. If both DCF and PCF are in use, the DCF (contention) mode and the PCF (contention-free) mode co-exist in the network so that one mode is always followed by another in short cycles (the DCF + PCF cycle is called a superframe). The time allocated to each mode changes dynamically based on the hosts' need for each type of traffic in the LAN. The cycle length is specified so that all hosts have a guaranteed chance of accessing the network during the superframe. The interframe spaces are by definition shorter for hosts that operate in contention-free (PCF) mode; This way hosts in DCF mode are not able to interrupt the contention-free period. The PCF mode is useful when there is a need to guarantee a certain constant amount of bandwidth for one or more hosts in the network segment (e.g. in multimedia applications).

802.11 MAC Frame Format

The 802.11 frame is formed of nine fields as seen in Figure 2.7. The first seven fields form the frame header. The header fields contain frame control, duration, address and sequencing information. The length of the header is 30 octets (240 bits).

The **Frame control** field is further divided into 11 subfields. It contains subfields for protocol version, frame type and subtype, fragmentation control, wireless encryption and power management. The **protocol version** is intended to be changed from 0 to another value in case of a major change in the protocol specification. The **type** and **subtype** subfields indicate the frame's function, i.e. whether the frame is a data, control or management frame and what its key function is (e.g. association/disassociation request or response, beacon, powersave etc.). The **to DS** and **from DS** subfields determine how the address fields in the header are interpreted (i.e. what addresses are given).

In DCF (contention) operation the **Duration/ID** field holds a frame type dependent frame duration value (1..32767). If the system is in PCF operation, this field is fixed to the value 32768. Other values are either reserved for future use or used for carrying association information in the frames of the type in question.

Depending on the type of frame in question, the 48-bit **Address** fields can carry any of the following: a BSS identifier (the 48-bit address identifying the basic service set (the *cell*) in which the host operates), a destination address, a source address, a transmitter address and a receiver address. The types of addresses contained in the address fields is defined by the **to DS** and **from DS** subfields in the frame control field. The 48-bit addresses are formatted the same way as in 802.3 (Ethernet) MAC.

The **Sequence Control** field has two subfields, the fragment number subfield (4 bits) and the sequence number subfield (12 bits). If service or management data needs to be fragmented, the sequence number subfield contains a specific identification value for the service or management unit in question. Fragments are numbered from zero onwards and the first fragment has the value zero in its fragment number subfield.

The header is followed by a variable-length **frame body** with the actual payload to be carried. The frame body length is specified to be between 0 and 2312 octets. The frame ends with a one-field trailer used for error detection. The only trailer field is **FCS (frame check sequence)**, which contains a CRC-32 checksum of the entire frame. The FCS is used and generated the same way as in 802.3 (Ethernet) MAC.

802.11 MAC Processing Characteristics

In terms of required processing in local area networks running IEEE 802.11 MAC, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **Random number generation.** Random numbers are needed in the 802.11 MAC protocol to determine te number of backoff slots to wait before sending a frame is allowed.

- **Timer functions.** Time needs to be measured to implement the waits required by interframe spaces and random backoffs.

- **Frame header analysis.** As in 802.3, also in 802.11 the frame header analysis requires a multitude of operations. First, the combination of addresses included in the header needs to be determined by analyzing the DS bits. After this, the addresses need to be validated (to find out, if the receiving station is required to process the frame further). These operations require **bitwise manipulation** and **Boolean comparisons**. **Counting functions** are needed for processing the duration/ID value, fragmentation information and sequencing control. Finally, as seen in the previous discussion, all 802.11 MAC header fields may only contain values greater than or equal to zero. Moreover, implementing the CSMA/CA access method does not require the use of negative values. Thus, 802.11 MAC processing can be carried out using **unsigned arithmetic**.

- **Checksum calculation.** Each 802.11 frame contains a 32-bit CRC checksum. The checksum needs to be calculated both in sending and in receiving stations.

- **Data buffering.** Buffering is needed in processing fragmented frames; incoming fragments need to be stored until all fragments of the original data have arrived.

2.3.4 Asynchronous Transfer Mode

In the late 1980's a solution called Broadband Integrated Services Digital Network (B-ISDN) was suggested as the basis for future telephone networks. The goal was to make transportation of anything from telephone calls to television programs and video on demand (VoD) possible across a single fast network. The underlying transport technology in B-ISDN was called Asynchronous Transfer Mode (ATM) [59, 64]. B-ISDN did not become popular, but its transport technology ATM is widely used. It is used e.g. to connect remote LANs to each other. The standardized transmission speeds for ATM are 155.52 Mbps and 622.08 Mbps, which are also standardized SONET and SDH speeds. Until recently, ATM has been a valid choice for network technology when transmission speeds of several hundred megabits per second have been needed. However, the recent emergence of Gigabit Ethernets (as standardized in IEEE 802.3 and its amendments) may soon change this situation.

In PSTNs (Public Switched Telephone Networks) a method called circuit switching is used to send information from the sender to the receiver. Basically circuit switching means that a (physical) wired circuit is formed between the calling parties by means of using switches that connect wires to one another. ATM is a packet switching technology that logically forms virtual circuits (VCs). This means that instead of using switches to form a direct wire connection with the calling parties, data PDUs are sent from the source through a series of nodes called ATM switches to the destination. The route, or VC, that the ATM PDUs (called ATM cells) travel on has been decided on during connection setup and is not altered during the connection; thus all cells of a single connection take the same route from sender to receiver. When the parties wish to terminate the connection, the virtual circuit is torn down (i.e. information of the particular VC is removed from the connecting ATM switches). This method of transmission has many advantages: first, the ATM cells moving on the same virtual circuit always arrive in the same order they were sent, and thus no cell reordering is required in the receiving end. Second, no routing decisions need to be made on individual cells in the switches, since their routes have been decided on already during connection setup. On the other hand, in packet-switched virtual circuit networks such as ATM, a malfunctioning network device causes all virtual circuits (i.e. connections) traveling through the device to be cut off.

In ATM networks the connection hierarchy has two levels: virtual circuits and virtual paths. Several virtual circuits form a virtual path similarly as several insulated copper wires would form a cable. The idea of this two level hierarchy is that if due to a network problem rerouting is necessary, then rerouting a single virtual path causes automatically the rerouting of the virtual circuits inside the virtual path. Thus the virtual circuits (single connections) do not need to be rerouted individually, but can be rerouted as a batch.

The ATM specification itself does not standardize a format for cell transmission. Cells can be sent independently or they may be embedded into a carrier system, e.g. SONET or SDH. There are separate standards specifying how ATM cells are packed into carrier frames provided by such systems. At any rate, this indicates that ATM cells can be sent either on electric wires or optical fibres, depending on the underlying carrier system.

ATM cells

Each ATM cell contains 53 octets. Of this, the ATM header consists of the first 5 octets and the cell payload consists of the last 48 octets as shown in Figure 2.8.

Figure 2.8 displays one of the two possible ATM header types, the network-to-network interface (NNI) header. There is also a header called the user-to-network interface (UNI) header. The UNI header differs from the NNI header with only a slight difference in the first field: the 12-bit VPI field is split into a four-bit flow control field called GFC, and an 8-bit VPI field. The GFC field exists only between the sending host and the first ATM switch, it is overwritten by the first ATM switch, and it has no significance to the

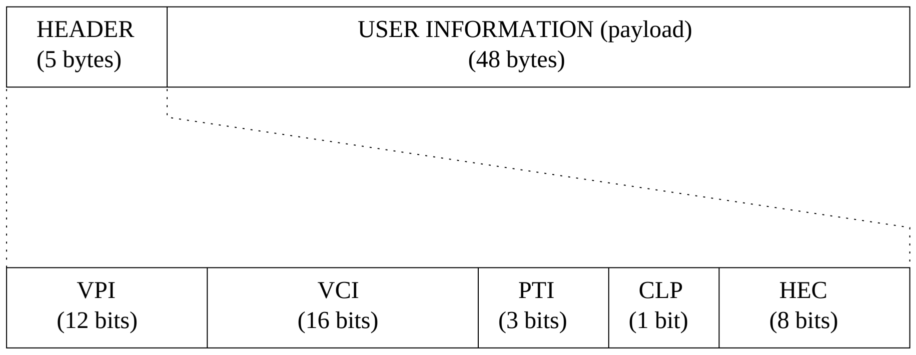


Figure 2.8: The ATM cell, network-to-network interface (NNI). VPI = virtual path identifier, VCI = virtual circuit identifier, PTI = payload type identifier, CLP = cell loss priority, HEC = header error check.

receiving station. Thus, in the following discussion we focus on the NNI header type.

The predefined route for the cell is represented by the **virtual circuit (VCI)** and **virtual path (VPI)** identifiers in the ATM header. Since the path information is a 12-bit integer and the circuit information is a 16-bit integer, it is easily determined that an ATM switch could have up to 4 096 incoming and outgoing virtual paths, each containing 65 536 virtual circuits. Actually the number of virtual circuits available is smaller since some VCI values are reserved for network control operations like setting up connections.

The **payload type identifier (PTI)** indicates whether the cell is a user data cell or a maintenance/resource management cell (OAM, operations and maintenance). Also, problems in cell transmission (congestion etc.) are indicated in the payload type identifier.

If there is congestion in an ATM switch and cells need to be discarded to regain normal operation, then cells with **cell loss priority (CLP)** set to 1 are discarded before cells with CLP=0.

The **header error check (HEC)** field of the header contains a checksum of the ATM cell header. Only the cell header is checked for errors (i.e. included in the checksum calculation). This is done to ensure that correct virtual circuit and path identifiers are received. The ATM HEC checksum is calculated using CRC-8[13].

ATM protocol stack and data transfer

As seen in Figure 2.9, the ATM protocol stack is somewhat different from the OSI and hybrid protocol stacks shown in Figure 2.1. Actually an ATM

[13] $G(x) = x^8 + x^2 + x + 1$

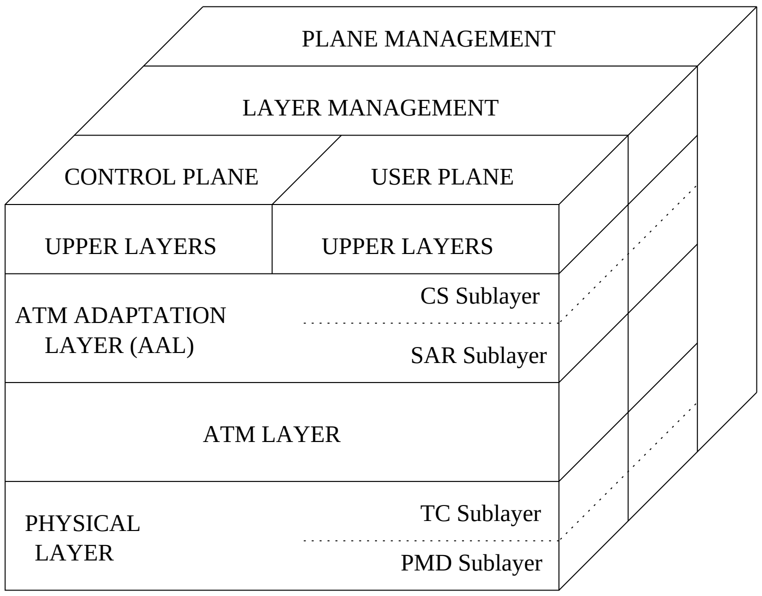


Figure 2.9: The B-ISDN ATM reference model as defined in [64].

network can not be conclusively subdivided into parts fitting the OSI or hybrid protocol models, since single layers of the ATM stack perform tasks across the layers of OSI and hybrid stacks. For example, the ATM layer is usually regarded as a data link layer protocol. However, a data link layer protocol is often defined as a single hop protocol used by machines at the opposite ends of a wire, but the ATM layer has actually the characteristics of a network layer protocol: end-to-end virtual circuits, switching and routing [110].

The most important task to be performed by the ATM adaptation layer (AAL, see Figure 2.9) is to segment the messages from upper layers in the protocol stack to small pieces that can be used as payload in ATM cells. Recall that the ATM cell is quite short (53 octets); thus, segmentation is needed for most upper layer data. Naturally, the AAL must also reassemble the messages in the receiving device. The convergence sublayer (CS) in the AAL provides the standard interface (i.e. the service access points) towards the upper layers, and the segmentation and reassembly of upper layer data is managed by the SAR (segmentation and reassembly) sublayer. The third dimension of the ATM protocol stack (i.e. control and user planes, layer and plane management) defines mechanisms for, among other things, flow control, resource management and interlayer coordination.

Before the upper layer data can be sent across an ATM network, the ATM layer needs to set up a virtual circuit from the sender to the receiver. The ATM layer sends a request cell to establish a transfer route. The switches along the transmission path return virtual route information to the unit that requested the route. The route information is then included in the cell headers (VPI and VCI) of all outbound data units. The transport medium for ATM is usually optical fiber, but for transmissions of less than 100 meters coaxial cable or category 5 twisted pair can be utilized [26, 84, 110].

ATM cells can be sent through a network directly or via a carrier. The ATM layer provides an outbound sequence of cells, and the transmission convergence (TC) sublayer of the physical layer (see 2.9) provides a uniform interface between the physical medium dependent (PMD) sublayer and the ATM layer. The TC sublayer converts the ATM cells to a bitstream and provides the bitstream to the PMD sublayer. The PMD sublayer is concerned with putting bits in the correct format at the correct time onto the medium.

An important issue in ATM is synchronization to an incoming bitstream. The header error check (HEC) field of the ATM cells is used to deal with this issue: the TC sublayer has a 40-bit shift register in connection with the incoming bitstream. The 40 bits in the register are examined to determine if they could form a correct ATM header (i.e. whether the last 8 bits form a correct CRC-8 checksum of the first 32 bits). If CRC-8 does not compute correctly, the register is shifted left one bit, and the next bit in the incoming stream is inserted into the rightmost position in the shift register. This is done until a valid HEC is found.

Once the register contains a valid ATM cell header (which still could be a random bitstring and not actually an ATM header), the next 424 bits (48 octets, possible cell payload) are discarded and the 40 bits after them (potential next header) are read into the register. The process is repeated until δ consecutive valid headers are found. This way the possibility of a random bitstring being interpreted as a valid row of ATM cells is $P = 2^{-8\delta}$ [108].

After finding δ consecutive valid headers, normal operation (i.e. processing incoming cells as valid data) starts. In normal operation, if α consecutive invalid headers are encountered, the system goes back to searching for one valid header and then to find δ consecutive valid ones.

ATM Processing Characteristics

In terms of required processing in networks running ATM, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **High speed operation.** ATM systems need to be able to handle ATM traffic at speeds of up to 622 Mbps. Such a high speed places clear demands on hardware performance.

- **Checksum calculation**. In ATM, the CRC-8 checksum is calculated very often, although only for 32 bits at a time. In normal operation it is done every 424 bits, which is about 1.5 million times per second in a 622 Mbps ATM network.

- **Counter functions.** Counter functions are needed frequently in ATM, e.g. to maintain the α and δ values needed for synchronization and error detection.

- **Cell header analysis.** Processing the ATM header fields requires at least **bitwise manipulation** (e.g. for processing virtual path and virtual circuit identifiers, and in synchronization) and **Boolean comparisons** (e.g. for comparing the checksum calculation result to zero). Also, as seen in this section, ATM header fields may only contain values greater than or equal to zero. In addition, ATM processing does not require the use of negative values. Thus, ATM implementations can be done using only **unsigned arithmetic**.

- **Data buffering.** Buffering is needed in AAL for storing cell payloads until all parts of an upper layer message have arrived. Buffering is also necessary in ATM switches.

2.3.5 The Internet Protocol

"The Internet began in early 1969". This statement given in [102], as blunt as it sounds, is in fact quite accurate. In the 1960's the U.S. department of defense saw a need for a robust command and control network that would remain functional even if several connecting nodes in the network fell offline (e.g. were destroyed by a bomb). The idea was that network traffic could be rerouted around the non-functional nodes through other nodes. In December 1969 the design work resulted in a functioning network of four nodes called the ARPANET[14]. The protocols used in the original ARPANET were not very well suited for internetworking, and in 1974 Cerf and Kahn proposed a new set of internetworking protocols called TCP/IP [15]. The original ARPANET no longer exists, but newer networks and nodes built in its place and in addition to it today form the Internet as we know it.

The Internet protocol, or IP, is the network layer protocol used in the Internet. The current version of the protocol (IP version 4, IPv4) is by no means a novelty; after Cerf and Kahn's work in the 1970's, a Request for Comments (RFC) specification[15] of the IP protocol was given in 1980 (RFC 760) [49]. It

[14]ARPA = Advanced Research Project Agency (of the U.S. department of defense).

[15]RFCs are in general more or less considered to be Internet standards.

was updated in 1981, and the resulting specification was published as RFC 791 [51]. This is the most recent specification revision of the IP protocol.

The Internet is formed of a mixture of different kinds of networks running different kinds of protocols and operated by different kinds of devices. The IP protocol is used to construct a single internetwork out of this mixture. As stated above, IP was designed with internetworking as the key design goal. This means that the IP protocol is required to support a global addressing scheme and to be able to route datagrams (IP PDUs) from the sending machine to the receiving machine through a number of intermediate nodes. IP addresses these requirements by resorting to connectionless service: hosts send datagrams onto the network and expect them to eventually reach the receiving host. It is up to the protocols in layers above the IP protocol to manage retransmission when necessary, and it is up to the IP routers in the network to decide how individual datagrams travel across the mixture of networks from sender to receiver. It is important to realize that each datagram travels independently through the network, and thus datagrams that carry parts of a single application message may take different routes and arrive out of order. Also, it is possible that some routers need to further fragment the datagrams on the way (if e.g. a datagram is too large to travel through one of the networks on the route). For this reason, the IP protocol in the receiving station must be able to reorder incoming datagrams, reorder incoming fragments, and rebuild the original datagrams out of the fragments.

IP Datagram Format

An IP datagram, as seen in Figure 2.10, consists of two parts: a header part and a text part. The text part is the user payload of a datagram. Both of these parts can have a variable size, and the total maximum size of a datagram is 64 kB. Usually the total datagram size is less than 1500 octets; this makes it possible to fit a datagram into an Ethernet frame. The way this should be done was specified already in 1985 in RFC 894 [48].

The **version** field in the datagram header indicates to which version of the IP protocol the datagram belongs to. For IPv4, the value of this field is always 4. The **IHL (IP header length)** field indicates how many 32-bit data words form the variable-length header of the datagram. The minimum, and typical, value for this field is 5. The **type of service** field indicates the speed and reliability requirements for the datagram's transmission. This field is usually ignored by routers. The **total length of datagram** field indicates the size of the entire datagram, both the header and the text parts. The value is given in octets, and the maximum value for it is 65535 octets.

If the datagram is broken into fragments along the transmission path, then the **identification** field of each fragment indicates to which original datagram

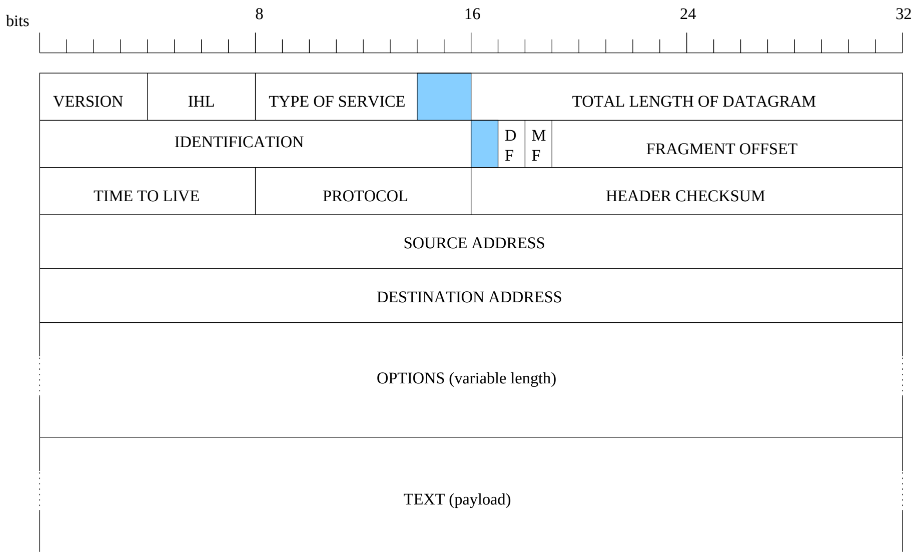


Figure 2.10: The IP datagram. Gray bits are unused (or no longer used). IHL = IP Header Length, DF = Don't fragment, MF = More fragments.

the fragment belongs to. The two control bits, **DF** and **MF**, are used for controlling fragmentation. If set, the **DF** bit indicates that the datagram may not be fragmented along the transmission path but must be delivered in one piece. The **MF** bit indicates that there are still more fragments of the datagram to come. The last fragment does not have this bit set, but all other fragments do. The **fragment offset** field indicates the position in the original datagram where the fragment belongs to.

The **protocol** field is used to indicate which transport layer protocol takes the datagram after it has been reassembled at the receiving host's network (IP) layer. Possibilities are e.g. TCP (indicated by value 6) and UDP (indicated by value 17).

The **time to live (TTL)** field indicates the time left in seconds before the datagram must be discarded in a router. Usually this value is decreased at each network hop instead of being decreased every second. This field ensures that a datagram can not wander around the Internet forever.

The **header checksum** field is used to verify the correctness of a datagram's header. Because at least one field in the header changes at each network hop (TTL is decreased), this value must be recalculated at each hop. The checksum is calculated using a custom algorithm defined in RFC 791 [51]:

> "The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero."

The **options** fields can contain values that concern security and routing issues, as well as any user defined options. One purpose of these fields is to allow experimenting without the need for allocating fixed non-used fields to each datagram.

The **source address** field contains the 32-bit IP address of the device that sent the datagram, and the **destination address** field contains the 32-bit IP address of the receiving device. IP addresses are formed of two main parts, the network identifier and the host identifier. If subnets are used, the host identifier is further divided into a subnet part and a host part. Subnet masks are used to separate the subnet and the host parts in subnet routers[16]. The lengths of the network and host identifiers vary depending on the class of network in use; A class addresses have an eight-bit network identifier and a 24-bit host identifier, B class addresses have a 16-bit network identifier and a 16-bit host identifier, and C class addresses have a 24-bit network identifier and an 8-bit host identifier. IP addresses are usually presented as a dot-separated list of eight-bit decimal numbers, e.g. the IP address 11000000 10101000 11111111 00010100 is written in the form 192.168.255.20.

The addresses were originally allocated by granting an institution or corporation a network identifier from one of the classes. The institutions and corporations then allocated the individual addresses to their hosts. Since the C class only supports 256 addresses (of which a few are special, i.e can not be allocated to hosts), many institutions and corporations requested B class addresses with up to 65 536 possible addresses. This kind of address hierarchy and allocation policy leads to inefficient IPv4 address space use; according to [110], more than half of class B networks have less than 50 hosts (of the more than 65 000 possible ones) connected to them.

For this reason among others, IPv4 is running out of addresses. The long term solution for the problem is expected to be provided by the next version of the protocol, IPv6. It introduces 128-bit addresses (which make it possible to have about $7 \cdot 10^{23}$ IPv6 addresses on each square meter of the earth's surface [110]), and a more efficient allocation policy. IPv6 is discussed further in the next section. The short term solutions that have been been taken into use, while waiting for IPv6 to start dominating the Internet, are **classless inter-domain routing (CIDR)** [34] and **network address translation (NAT)** [103]. In CIDR, the rest of the available IP addresses are allocated more efficiently by forgetting the concept of address classes. If for example an institution or corporation needs 4000 addresses, it is granted an address space of 4096 addresses (the nearest two's power). However, this approach requires more processing in routers due to a more complex analysis of the destination address.

[16] An AND operation is performed between the destination address and the subnet mask to find out to which subnet a datagram should be transported.

NAT takes a totally different approach to the problem: the basic idea is to use a certain block of (hundreds of) IP addresses internally in an institution or corporation, and to use just one or a few globally known addresses for outside connections. The internal addresses are not visible to the outside network, but only to hosts within the institutions or corporations own network. A special device called a NAT gateway is used to translate the internal IP addresses into one of the available global addresses. A session ID[17] is also included in the newly formed datagrams. With the use of session IDs, addresses of incoming datagrams can again be converted (or mapped) back to internally used addresses. Certain ranges of IP addresses have been defined for use in such internal networks, and if an address in one of these ranges is encountered in an inter-domain router, the corresponding datagram is discarded.

IP Processing Characteristics

In terms of required processing in networks running Internet Protocol version 4, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **Support for high speed operation.** The speed at which datagrams arrive in a host running the IP protocol depends on the underlying data link and physical protocols in use. IP devices may need to be able to process datagrams at speeds of up to 10 Gbps. More typically, the network speed range for IP networks nowadays is from 1 Mbps to 1 Gbps.

- **Checksum calculation**. Each host and each router needs to be able to calculate the custom IP checksums. Sending hosts need to calculate the checksum for the headers of outgoing datagrams. Routers need to first verify the checksum of an incoming datagram and then recalculate it after the routing decision has been made. Receiving hosts need to verify the checksums in incoming datagrams.

- **Datagram header analysis.** Processing the IP header fields requires at least **Counter functions** for e.g. accessing the header fields (IHL determines header length), **bitwise manipulation** for e.g. analyzing the addresses and processing subnet information, **Boolean comparisons** for e.g. determining correct protocol version. Since the IP header fields may only contain values greater than or equal to zero, and IP processing can be done without using negative values, **unsigned arithmetic** is adequate for IP implementations.

[17]More precisely, actually the distinct set of source and destination TCP/UDP ports and source and destination IP addresses of a transmission is used as its session ID.

- **Data buffering.** Buffering is needed to reorder datagram fragments before providing the payload to the upper layer. Routers also need to buffer datagrams that have entered the router but have not been processed yet.

2.3.6 The Internet Protocol version 6

To answer to the existing problems in IPv4 (first and foremost, the problem of running out of IP addresses), in RFC 1550 [12] the IETF[18] called for proposals describing a new version of IP. IETF used the name "IPng", or IP: next generation, for the new protocol. As a result of this call, a combination of two of the proposals was published in 1994 with the name "Simple Internet Protocol Plus" (SIPP) [46]. A year later, the first specification describing Internet Protocol version 6 [27] was published based on the SIPP proposal. The most important difference with the SIPP proposal and the IPv6 specification was that the 64-bit addresses suggested for SIPP were replaced by 128-bit addresses in IPv6. The most recent version of the IPv6 specification was published in 1998 as RFC 2460 [28].

In addition to the size of the address space and the ways of allocating the addresses, IPv6 provides support for authentication and payload encryption, which were not integral parts of IPv4. IPv6 also specifies type-of-service features (i.e. traffic classes). The IPv6 main datagram header has a fixed size with 7 fields (IPv4 had a variable size with 13 fields), which should make its processing in routers faster. Dropping the header checksum also speeds up the processing of regular datagrams in both hosts and routers. However, checksum calculations are still needed in e.g. inter-router control messaging. For regular datagrams, IPv6 relies on the upper and lower layers in the protocol stack to manage error detection and error correction. Finally, inter-router fragmentation is not allowed; managing datagram fragmentation is up to the sending and receiving host only. Thus, if a datagram is too big for a subnet, the subnet router will discard the datagram and send an error message to the sending host. The sending host will then fragment the datagram into pieces small enough to travel through the subnet. This feature also improves network performance by simplifying the fragmentation and relieving routers from it.

IPv6 Datagram Format

Like IPv4, an IPv6 datagram consists of two parts, the header and the payload part. As a difference, the IPv6 header consists of a fixed-length part with seven fields, and an optional set of extension headers. The extension headers

[18]IETF stands for the Internet Engineering Task Force.

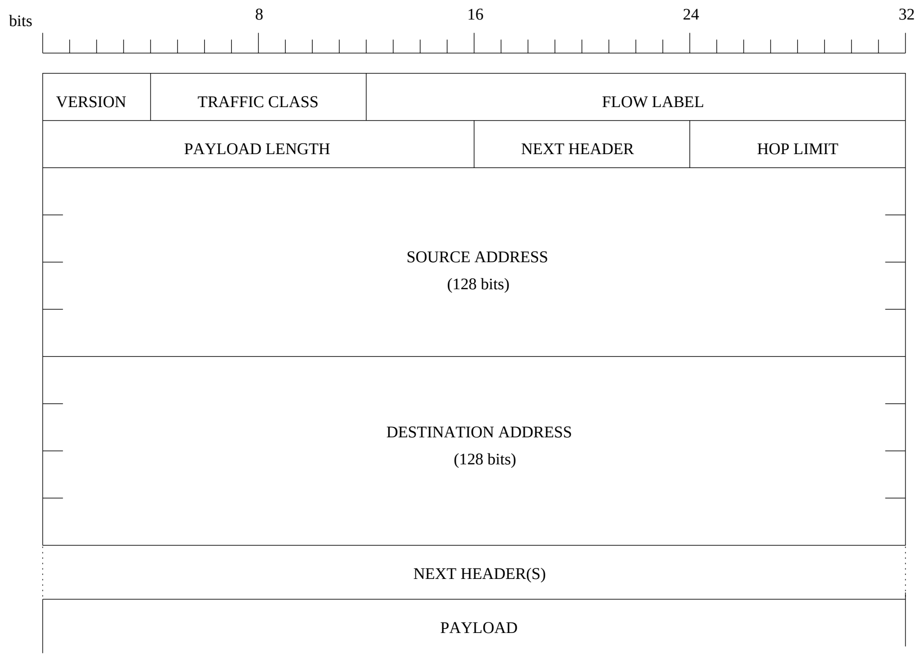


Figure 2.11: The IPv6 datagram. The first seven header fields are followed by zero or more extension headers, the upper layer header ("NEXT HEADER(S)" in the Figure) and the upper layer payload. The NEXT HEADER field points to the upper layer header, if no IPv6 extension headers are present.

specify additional information for the receiver or for the routers along the way. The extension headers can carry (a combination of) additional information about routing, end-to-end fragmentation, authentication and encryption. The fixed part of the header is 320 bits, or 40 octets, of length. Different kinds of extension headers are of different sizes. Some of them have a fixed length and some can have a variable length. Each extension header has a "next header" field pointing to the next header or the upper layer header.

The **version** field of the IPv6 header must always contain the value 6 to indicate the protocol version. We recall that in the currently used version of the IP protocol this field is always set to 4.

The **traffic class** field is used to indicate the type of service the data requires. Values from 0 to 7 support flow control mechanisms, and values from 8 to 15 are intended for real-time services such as audio and video connections. A higher value within one of these ranges indicates a higher priority and/or bandwidth requirement. The lower the value within one of the ranges, the sooner the packet is discarded if a router becomes congested.

Flow labels can be used to setup connections with special properties between two hosts. In other words, the flow label is a connection identifier for the communicating hosts. The flow label reserves bandwidth for a connection between two hosts: if a datagram with a non-zero flow label is encountered by a router, the router needs to find out from its flow tables the type of special service needed by the datagram.

The **payload length** field indicates the number of octets in the datagram in addition to the fixed-length header. The maximum value for this field is 65 535 8-bit bytes, or 64 kB. However, the IPv6 datagram size is not limited to any value; if the payload is larger than 64 kB, the payload length field is set to all zeros and the actual payload length is given in an extension header. Datagrams with payloads larger than 64 kB are called *jumbograms*. Jumbograms are mostly useful in supercomputing applications in which massive amounts of data need to be transferred.

The **next header** field indicates the type of extension header (if any) following the fixed length header. If there is no extension header following the main header, the field indicates the receiving upper level protocol (e.g. TCP).

The **hop limit** field is used to make sure that a datagram is removed from the network at some point and not left to travel in the network forever. The maximum value for the hop limit is 255, and it is decreased by one at each network hop (router).

The **source** and **destination address** fields contain the 128 bit IPv6 addresses of the sender and the receiver of the datagram. IPv6 addresses are written as eight semicolon-separated groups of four digit hexadecimal numbers, e.g. ABCD:1234:7890:0000:0000:000A:0D33:9000. It has been decided that in addition to this normal (called *preferred*) notation, an alternate (*abbreviated*) notation may be used; in this notation, all leading zeros are left out and sets of one or more groups of four hexadecimal zeros can be replaced by two semicolons. With these rules in mind, the previous IPv6 address could then be written as ABCD:1234:7890::A:D33:9000.

IPv6 Processing Characteristics

The processing requirements of IPv6 are quite similar to those of IPv4. However, the long addresses in IPv6 require more processing and bitwise manipulation than in IPv4. Also, the unlimited datagram size places greater demands on buffering and memory. In practice however, in most communication between IPv6 hosts, datagrams need to travel through networks that run the IEEE 802.3 (Ethernet) MAC protocol. For IPv6 over 802.3, the maximum datagram size has been specified to 1500 octets in [22]. Taking into account that routers do not fragment datagrams in IPv6, we can state that most IPv6 traffic is likely to be carried out with datagrams of 1500 or less octets in length.

On the other hand, the processing of IPv6 datagram headers is faster than in IPv4 because (1) there are fewer fields to analyze, (2) the main header is fixed in length, and (3) there is no checksum to be calculated by hosts. Routers need to be able to calculate checksums to send and receive control messages, but not for each incoming and outgoing datagram (which is the case in IPv4).

In terms of required processing in networks running Internet Protocol version 6, the following distinct characteristics set requirements and demands for designing protocol processing devices:

- **Support for high speed operation.** The speed at which datagrams arrive in a host running the IPv6 protocol depends on the underlying data link and physical protocols in use. IPv6 devices may need to be able to process datagrams at speeds of up to 10 Gbps. More typically, the network speed range for current networks is from 1 Mbps to 1 Gbps.
- **Checksum calculation**. IPv6 routers need to be able to calculate Internet checksums for control messages.
- **Datagram header analysis.** Processing the IPv6 header fields requires at least **counter functions** for e.g. reducing hop limits in routers, **bitwise manipulation** for e.g. analyzing the addresses and **Boolean comparisons** for e.g. determining correct protocol version. As seen in the previous discussion, the IPv6 header fields may only contain values greater than or equal to zero. Also, IPv6 processing can be done without using negative values. Thus, **unsigned arithmetic** is adequate for IPv6 implementations.
- **Data buffering.** Buffering is needed in the sending and the receiving host to reorder datagrams before providing their payloads to the upper layer. Routers also need to buffer datagrams that have entered the router but have not been processed yet.

2.4 Chapter Summary

This chapter gave an overview of the protocol processing application domain. The presented analysis of six important and widely used protocols and the additional findings in [67] make it clear that protocols and protocol processing applications have similar requirements for the functionality needed to process them, and this can be taken advantage of in hardware design. Table 2.1 summarizes the key processing operations and characteristic functionality of protocol processing found in this chapter. As can be seen in this table, most of the detected characteristic functionality for a particular protocol was found to actually be characteristic to several other protocols as well. A processor

Processing Characteristic	SDH	IEEE 802.3	IEEE 802.11	ATM	IP	IPv6
High bitrate	Yes	Yes	No	Yes	Yes*	Yes*
Boolean evaluations	Yes	Yes	Yes	Yes	Yes	Yes
Bitwise manipulation	Yes	Yes	Yes	Yes	Yes	Yes
Counter functions	Yes	Yes	Yes	Yes	Yes	Yes
Timer functions	Yes	Yes	Yes	No	No	No
Checksum calculation	No**	Yes	Yes	Yes	Yes	No***
Random numbers	No	Yes	Yes	No	No	No
Buffering	Yes	Yes	Yes	Yes	Yes	Yes
Unsigned arithmetic	Yes	Yes	Yes	Yes	Yes	Yes

Table 2.1: Summary of typical and essential functions found in the protocols analyzed in this chapter. Bitwise manipulation means bit-pattern matching and/or masking inside data words, and/or n-bit shifting. High bitrate means network speeds higher than 500 Mbps.

*The required processing speed in the network layer depends on lower layer protocols used.
**SDH uses parity bits, the calculation of which does not classify as a checksum.
*** Checksums are needed in some cases, e.g. in routers when processing control messages.

with optimized hardware support for the found common functionality should be easily and clearly able to outperform a similar processor with general purpose processing units. Also, the power consumption and area use of such an optimized processor can be expected to be less than those of a general purpose implementation. This is on the other hand due to a reduced clock speed requirement (the optimized processor is likely to provide equal processing performance at a lower clock speed), and on the other hand due to the fact that the optimized processor needs to implement only the required subset of operations (general purpose execution units may also implement extra functionality that is not needed for the desired protocol processing functionality). The last row in Table 2.1 displays a very important finding in terms of processor design: protocol processing can be implemented using only unsigned arithmetic (i.e. there is no need for managing negative values), which makes hardware implementations considerably simpler.

Clearly, there are foreseeable advantages to be gained by designing optimized processors for protocol processing applications. In Chapter 3 we develop a protocol processor hardware platform that has been designed taking into account the protocol analysis results reached in this chapter.

Chapter 3

The TACO Hardware Platform

Chapter 2 ended with the conclusion that it is beneficial to design processors with optimized hardware for protocol processing. This chapter proceeds from that conclusion to specifying a modular and scalable transport triggered (TTA) [20, 107] hardware platform for the TACO protocol processor design framework. In this hardware platform, which we will call the *TACO architecture*, each execution unit is optimized for a particular protocol processing task. Some of the tasks have been chosen for hardware implementation based on the findings of the previous chapter, and others based on application analyses carried out in later case studies such as the ones discussed in Chapter 5.

We start the discussion in this chapter with an introduction to computer architectures and key architectural approaches. The discussion outlines a path of improvements made to computer architectures through time, ultimately leading to high performance solutions like VLIW, superscalar and transport triggered architectures. VLIW has been seen to exhibit several characteristics that could be beneficial for automated protocol processor design, such as easy instruction decoding, adaptivity for specific applications and high performance (in part due to increased parallelism) [20]. However, as pointed out in [20], TTA based architectures can be expected to provide at least the same beneficial characteristics as VLIW, but at a reduced complexity. Thus, we argue that specifying a TTA based architectural solution for use in the TACO protocol processor design framework provides many advantages, among which is good support for design automation.

Based on the results of the protocol analysis presented in the previous chapter we have decided to implement support only for unsigned arithmetic into the TACO architecture to reduce hardware complexity. Also, among the TACO execution units there are no general purpose processing elements like multipliers or arithmetic-logic units (ALUs). We argue that this kind of an architecture is very efficient for protocol processing: on the other hand typical protocol processing operations can be performed with optimized processing units in hardware, and on the other hand the architecture remains relatively small and simple due to the missing general purpose processing elements. Also, resorting to TTA reduces the amount of control logic (and the processing overhead produced by it) needed in a processor, since many traditional hardware tasks are taken care of already at the time of designing the application software; for this reason, for example dynamic code scheduling in hardware is not necessary in the TACO architecture.

We also argue that the TACO architecture is very modular and scalable: creating new protocol processing execution units for use in the TACO platform is quite straight-forward since the communication interface differs very little from one unit type to another. Also, execution units created for an earlier TACO architecture instance can be conveniently reused in later TACO design projects. To validate these arguments, in this chapter we give a detailed presentation of the TACO architecture, its building blocks and their interfacing, and conclude the chapter by summarizing the key differences between the TACO architecture and the *MOVE* TTA template [20].

3.1 Computer Architectures

The classification of computer generations as well as the choice of a device to be credited as the first-ever computer are topics that many scientists and researchers disagree upon; in [104] Stallings suggests that there have been five generations of computers. The first three generations of these (vacuum tube computers 1946-1957, transistor circuit computers 1958-1964, integrated circuit (IC) computers 1965-1971) are agreed upon by most authors. According to [104], the division between the fourth and fifth generation is not clear, although the generation is suggested to have changed with the shift from large scale integration (LSI) based circuit technology (1972-1977) to very large scale integration (VLSI) based circuit technology (1978 onwards).

In [109] Tanenbaum more or less agrees with the first three generations mentioned above, although he defines the third generation to be the combination of Stallings' generations 3 and 4 (IC and LSI). Tanenbaum defines the fourth generation to have begun in 1980 in the form of VLSI based computers. In addition, he defines an additional generation, called the zeroth generation for mechanical computers (1642-1945).

Similar disagreements exist for the device to be credited as the first computer; Some see the ancient Greek Antikytheran device [90] as the first computer of all time[1]. In [109] Tanenbaum credits the first-ever computer to be

[1]The Antikytheran device was assumedly designed for calculating the future motions of

the difference engine built by British scientist Charles Babbage in 1834. In [104] Stallings does not attempt to name the first computer ever; instead, he names the ENIAC machine built in the University of Pennsylvania in 1946 to be the "first general-purpose electronic digital computer". Tanenbaum names a device called COLOSSUS built by the British government in 1943 to be the "first electronic digital computer", and credits ENIAC to have started "an explosion of interest in building digital computers".

In the context of this thesis we accept the fact that opinion is divided on the definitions of computer architecture generations as well as the concept of the world's first computer (whether digital or analog, electrical or mechanical), and will therefore pursue these topics no further. Instead, we will focus our discussions in this chapter on current typical computer architectures. However, the previous discussion on historical architecture generations sheds some light to the relation between computer architecture and microprocessor architecture; the early digital electronic computers did not have microprocessors but the functionality was implemented as a circuit board with discrete components. Later, when gradually larger and larger sets of such discrete components were incorporated into one component, namely an integrated circuit, the microprocessor was born. With microprocessors it became possible to integrate parts of, or even complete, computer architectures into a single chip instead of the earlier discrete component circuit board structure. For this historical reason, printed work describing basic microprocessor architectures still often contains the words "computer architecture" within the title.

In the following, we will briefly study some of the most significant types of computer, or nowadays microprocessor, architectures. The discussed architectures are the traditional Complex Instruction Set Computer (CISC), the Reduced Instruction Set Computer (RISC), the Very Long Instruction Word (VLIW) Computer, the superscalar architecture and the Transport Triggered Architecture (TTA). Since the TACO architecture is based on TTA, in the following we take the TTA discussion somewhat further than the discussions on the other mentioned architectures.

3.1.1 Complex Instruction Set Computer

The early computers were quite simple and they could be programmed with simple and compact instructions. The instruction sets were also very compact. As computers became more powerful and more complex applications needed to be executed on them, and on the other hand as memories were slow and expensive, processing complexity was moved from software to hardware: with more complex instructions, a single instruction could be used to

stars and planets. The device was found in 1900 in an ancient Greek ship wreck in the Mediterranean close to the island of Antikythera.

perform an operation that earlier required several consequent simpler operations. This meant more processing speed for the application due to fewer instruction fetches from the program memory. Also, with complex instructions, application programs became shorter in terms of code lines and thus less memory was needed to store the program in the computer.

In addition to growing program complexities, a need for executing the same application in a variety of computers (low cost and speed vs. higher cost and speed) had risen. To deal with this need, already in the 1950's a method called interpretation was suggested (see [109] for a discussion on the history of interpretation). The idea was that complex application instructions were interpreted inside the computer into native microinstructions for the particular computer type. The interpreter was actually a kind of embedded program running in the computer in addition to the application software. Interpretation made it possible for manufacturers to generate product families of computers, in which only the most expensive machines directly supported the complex instructions in hardware (and hence were very fast), and the rest used interpretation.

These development tracks lead in the 1970's to a situation where almost all computers were based on complex instruction sets, microcode and interpretation [109]. These computers and their later descendants are called Complex Instruction Set Computers (CISCs). A clear advantage of CISCs at the time was that compiling high level programming languages to complex machine instructions was far less demanding than compiling them directly to microcode (here it is necessary to realize that compiler technologies and compilation speeds of that time do not even nearly scale up to their modern-day counterparts). On the other hand, interpreting complex instructions into microcode inside a computer or a processor and then executing this microcode takes time and worsens overall system performance. Also, nowadays memory price and instruction fetch performance are no longer limiting factors in computer design, and on the other hand processing speed still is. Therefore the advantages gained by producing compact and complex CISC code (instead of more elaborate yet faster low level code) are nowadays compromised rather than clear. However, in embedded systems the amount of available memory is often limited, for which reason the amount of memory allocated for application code needs to be carefully considered. CISC-like approaches might thus be worth consideration in embedded applications.

A good example of CISCs is Intel's 80x86 processor family. Even the newest processors in this product line are backwards compatible with the 8088 processor used in the original IBM PC (i.e. are able to execute code written for the 8088). However, starting from the 80486 generation, Intel processors have gradually adopted more and more features from RISC and superscalar computer architectures to improve overall processor performance.

Such enhancements to the traditional CISC approach include for example instruction level parallelism and execution prediction as well as non-interpreted execution of the simplest instructions in the instruction set [109].

3.1.2 Reduced Instruction Set Computer

In late 1970's and early 1980's, as instruction set complexity in computers was growing to meet the capabilities of high level programming languages, some research was also carried out in the opposite direction, i.e. to reduce instruction complexity for better performance. This research direction introduced the concept of Reduced Instruction Set Computer (RISC) architectures. Early RISC architectures include the IBM 801 Minicomputer [93], the Stanford MIPS [42] and the Berkeley RISC-I and RISC-II [86, 87] processors. A detailed comparison of IBM 801, RISC-I and MIPS is given in [87].

RISC architectures trade backwards compatibility for instruction sets that do not require interpretation: all instructions are directly executed by hardware. RISC instructions should execute as fast as CISC microinstructions and have similar or lower complexity [104]. This way the performance penalty of interpreting complex code is removed; on the other hand, the program compiler will now have to convert code written in high level programming languages into very elaborate machine code instead of the traditional complex instructions. A design goal for RISCs is that each instruction is executed in one machine cycle. Also, instead of resorting to frequent data memory accesses (as in CISCs), RISCs encourage the use of register-to-register transfers. This is achieved by providing only simple load and store operations as well as more general-purpose registers than in CISCs. Finally, an important characteristic of RISCs in comparison to CISCs is that RISC instructions typically are of a fixed size that matches the processor's word length. Thus, instruction decoding in the processors is simpler and faster than in CISCs that support variable lengths and formats for instructions.

For novel computer and processor designs, RISCs seem to be a good modern-day approach. However, pure RISC designs have not become as popular as one might expect. Instead, CISC devices like the Intel 80x86 processors have started to incorporate more and more RISC-like characteristics (like executing simpler instructions directly without interpretation) in newer product generations. These kinds of hybrid CISC-RISC devices are not as fast as pure RISCs with same development and manufacturing costs would be, but still provide better performance than pure CISCs [109].

An example of modern RISCs is the PowerPC family of processors that resulted from a co-operation between IBM, Apple and Motorola. The original 32-bit PowerPC processors were designed along the RISC design guidelines discussed above. Current PowerPCs are 64-bit superscalar RISC machines.

As with the Intel processors slowly leaning more and more towards RISC, a similar, although perhaps not so obvious nor strong, trend can be observed for the PowerPC processor family; the increasing instruction set complexity and the increasing complexity of supported hardware operations drive the PowerPC family slowly, yet gradually into features and characteristics that are usually linked more to the CISC rather than the RISC design space. However, a key definition of RISCs, completely uninterpreted code execution, still holds for current 64-bit PowerPC processors. Also, 64-bit PowerPCs are able to natively execute old 32-bit PowerPC RISC instructions.

Nowadays PowerPC processors are mostly used in Apple computers and many embedded systems (e.g. in cars), and e.g. in high performance FPGA platforms such as the Xilinx Virtex-II Pro family.

3.1.3 Superscalar Architecture

Superscalar architectures rely on execution unit replication to achieve parallel execution of sequential code. This kind of approach requires additional dispatch circuitry in the architecture to deliver sequential instructions to available execution units: instead of waiting for a particular unit to complete its execution, the dispatch circuitry immediately assigns the next operation to another execution unit of the same kind (if one is available). The compiler does not need to schedule code to support the replicated execution units; the scheduling is done dynamically by the dispatch circuitry. A superscalar processor potentially executes sequential code much faster than a similar sequential processor would at a given clock speed due to parallel code execution on replicated units. On the other hand, this means that dispatch circuitry optimization becomes a very important task in designing a superscalar processor.

According to [104] and [109], the term *"superscalar"* was first used to describe these kinds of computers in 1987 by researchers at IBM. Superscalar features were initially introduced to RISC-based architectures: RISC simplicity (small size, simple instructions) allowed reasonably easy execution unit replication and simpler-than-CISC dispatch circuitry design [104]. Advancements in chip manufacturing processes and technologies later made it possible to start gradually adding superscalar features into CISC-based processor families like the Intel 80x86 processors. The Intel Pentium processor and its descendants are an example of modern superscalar CISC-based processors, and the PowerPC family an example of modern superscalar RISC processors.

3.1.4 Very Long Instruction Word Computer

Very Long Instruction Word (VLIW) architectures are very closely related to superscalar architectures. The goal in both approaches is to speed up execution by resorting to instruction level parallelism and replicated execution units. The key differences lie in code scheduling and instruction formatting: from the previous section we recall that a superscalar processor takes in sequential code and dynamically schedules it in the hardware for parallel execution. In VLIWs, the program compiler is responsible for efficiently scheduling the code for execution. This approach has some clear benefits as well as some clear drawbacks. In the following we discuss some of the key aspects in both respects.

Moving the scheduling from hardware to software considerably simplifies the processor architecture (circuitry needed by scheduling is removed) and speeds it up (no time is lost in dynamic scheduling decisions made in hardware) with the cost of additional effort in application software and/or intelligent compiler design. This also makes it possible to update and improve the scheduling algorithm whenever necessary, whereas in superscalar processors the scheduling algorithm can not be changed later. Also, compile-time (or static) scheduling allows scheduling of the entire code before it is executed on the hardware, which is not the case for superscalar architectures. In practice however, the entire code can often not be optimally scheduled into separate execution tracks because of difficulties in data dependency, conditional execution, looping and jumps. For this reason, VLIWs give only a moderate performance improvement over superscalar implementations.

Static scheduling means on the other hand that the machine code can only be run in a microprocessor that exactly fulfills the scheduling constraints such as a specified (minimum) number of execution units and control and data paths. Thus, product differentiation and designing product family generations becomes much more difficult than in commercial superscalar architectures (where the same compiled code can usually be executed on each device in a manufacturer's product lineup). Because of the static scheduling the compiled code must be in a format that can be easily decoded in hardware into instructions for each execution unit. This is accomplished by using a fixed length instruction, in which each execution unit has its own control, or subinstruction, field. The control field length may be different for each execution unit type (the length depends on the number of bits needed to operate the particular execution unit type).

Since each execution unit requires its own portion of the instruction word (typically 16-32 bits), in a processor with several execution units and capabilities to execute several subinstructions in parallel the instruction word length grows rapidly. Typically the instruction word length in a VLIW is between 128 and 1024 bits [101]. Comparing this to the instruction word lengths of 32 or 64 bits used in modern RISCs and superscalar processors gives context to the name *"Very Long Instruction Word"* computer given to the architecture.

The Texas Instruments TMS320C6X DSP (digital signal processor) family is one of the most successful commercial approaches to VLIW. These processors are used mostly in embedded systems for communication, for example in ADSL modems and wireless communication devices. The C6X processors have an instruction word length of 256 bits. The processors support eight parallel operations per clock cycle, using two data paths. Other commercial VLIW products include e.g. the Transmeta Crusoe processor for low power mobile applications requiring Intel 80x86 compatibility[2].

3.1.5 Transport Triggered Architecture

The most important difference between Transport Triggered Architectures (TTAs) and more conventional computer and microprocessor architectures such as the ones described in this chapter is the way the processors are programmed. Traditionally processors are programmed by specifying operations, which cause data transports to occur in the processor as a side effect. In TTA processors this programming paradigm is mirrored [20]: Instead of programming operations, in TTAs the data transports are explicitly programmed, and operations occur as a side effect of these transports.

The concept of transport triggering was first introduced for digital controllers by Lipovski et al. in [107] in 1980. In the 1990's the TTA paradigm was developed further by H. Corporaal's group in the Delft University of Technology, the Netherlands. Their work resulted in the *MOVE* TTA architecture template described in [20].

It is suggested in [20] that TTA processors are actually a superset of RISC and VLIW architectures: a RISC processor could be seen as a VLIW with only one FU; a VLIW could be represented as a TTA processor by using the same execution units and connectivity, and specifying the data transports as they would occur in the VLIW (i.e. not optimize the data transports for the TTA implementation). Thus, a performance improvement over a similar VLIW processor can be expected from a TTA implementation for the same application: parallel program execution can be optimized further due to direct programmability of data transports. TTAs could also be seen as ultimate RISC processors, or OISCs (One-instruction Instruction Set Computers), since at the most basic level there is only one machine instruction available: the *move* operation for transporting data from a source to a destination location. Since the *move* instruction causes different operations in the processor to occur depending on the transport destination, the term OISC can still be a bit misleading in the context of TTAs.

According to [20], VLIW scalability worsens rapidly as data transport capacity and connectivity increase (especially register files and bypass logic

[2]The Crusoe processor uses a technology called "Code Morphing" to convert 80x86 CISC instructions into native Crusoe VLIW instructions.

become very complex). Also, VLIWs are rarely able to utilize their available data bandwidth even when all FUs are busy. TTA alleviates these problems by making data transports visible; this way the programmer and the compiler are able to optimize data transports. For this reason TTA also strongly reduces the underutilization and complexity of the data path [20].

A TTA processor is formed of functional units (FUs) that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit. The FUs connect to the buses through modules called sockets. Each functional unit has input and output registers, and each of these has a corresponding socket. Operations are triggered as a side effect of data transports to specific addressable locations (e.g. certain registers in FUs). The number of transport buses and the number and type of FUs depends on the target application and usually also on design constraints for physical characteristics like clock frequency, power consumption, chip area etc.

By changing the type and number of FUs and by changing the connectivity and capacity of the interconnection network, a wide range of TTA processor architectures can be specified. Since the number of FUs and buses in the interconnection network is (in principle) not restricted and the design of these elements is independent, TTA is a very flexible platform in terms of hardware design. There are practically no constraints on designing the interconnection network and different kinds of FUs as long as both are in accordance with a given socket interface specification.

Like VLIWs, TTAs do not require logic for execution optimization (i.e. run-time instruction reordering to improve concurrent use of functional units etc.) Instead, TTA processors rely on the program compiler to perform instruction scheduling in an optimal way. In general, TTA instructions resemble VLIW instructions. They consist of several RISC type subinstructions that each define a data transport by specifying a source and a destination identifier. Conditional execution can be implemented e.g. by using special guard identifiers; if the condition specified in the guard identifier is not met, the corresponding data move is cancelled.

MOVE TTA Template

The *MOVE* framework TTA template [20] is without a doubt currently the best known actualization of the TTA paradigm and as such is a de-facto point of reference. Also, the TTA paradigm itself does not dictate any module specifications for processor implementations, only the general concept. Therefore we resort here to briefly discussing the key building blocks of the *MOVE* TTA template [20] to exemplify the kinds of structures needed to actualize the TTA paradigm.

Functional Units Functional units (FUs) are the modules that carry out execution of specified operations in a TTA processor. The *MOVE* framework [20] distinguishes between two classes of functional units: FUs and SFUs. FU operations typically resemble operations performed by general purpose processors. These operations are regular, commonly used tasks like ALU functions (in fact, an ALU can be considered to be an FU). In contrast, SFUs (Special Functional Units) perform operations that are application-domain specific and are not very frequently needed in general purpose processing.

An FU has a set of input and output registers. The main types of FU registers are *input* and *output*, used for inputting data to or outputting data from the functional unit. There are two subtypes of input registers, namely *operand (OP)* and *trigger (TR)* registers. Output registers are called *result (R)* registers.

The difference between operand and trigger registers is that a data transport to a trigger register triggers an FU operation. The FU operation uses the transported data word as an operand. The operand registers are used for inputting additional operands (for operations that need more than one value to compute, e.g. addition). For such operations, data needs to be transported to the operand register prior to triggering; data transports to operand registers do not trigger operations. The results of FU operations, if there are any, are stored in one or more result registers.

Each functional unit can provide more than one operation to be performed on the input data. This is possible by using opcodes that are extracted from socket addresses by the sockets (discussed shortly). For example, a functional unit that supports logical shifting could make a left or a right shift based on the extracted opcode.

Sockets A socket is a gateway between the interconnection network and a functional unit. Each socket is connected to one or more buses and to one FU register. A socket can send or receive one data word per clock cycle to or from the FU it is connected to.

An input socket evaluates if a destination identifier on a bus connected to it matches its own identifier. If the identifiers match, the socket passes data from the bus on which the identifier was found to the FU (more precisely, to the register in the FU to which the socket is connected). The number of destination identifiers a single socket can recognize is not limited to one. If the socket has more than one identifier, an *opcode* is extracted from the identifier. This opcode is passed to the FU at the same time as the data is, and the operation performed by the FU on the data is specified by the extracted opcode.

Trigger sockets are a special kind of input sockets. Trigger sockets function like input sockets, but upon passing the data from the bus to the FU the trigger socket also signals the FU to start executing its operation. Data transfers through regular input sockets do not cause FU operations to start executing.

An output socket is similar in implementation to an input socket. It compares the source identifier(s) on the connected source bus(es) to its own identifier(s) and if there is a match, the possible opcode is extracted and data is passed from the FU to the bus on which the identifier was found. Also the output sockets can have more than one identifier.

Interconnection Network In *MOVE* TTA each bus on the interconnection network actually consists of data, address (source and destination) and control buses. Source and destination buses transport the *move* instructions to the sockets and data buses transport data from one FU to another. Control buses are used, among other things, for protecting unfinished execution and for conditional execution.

The interconnection network can be partly connected (each socket connects to only some of the buses), or fully connected (each socket is connected to every bus). If the interconnection network is fully connected, each register in each FU can move data on any of the buses, thus making code generation for the processor easier. The number of buses in a processor is not restricted; However, the size of the instruction word grows rapidly with the number of buses. To deal with this issue, the use of compression techniques for reducing TTA instruction word size has been suggested for example in [41].

3.2 TACO Protocol Processor Architecture

An important goal in the TACO framework has been to be able to take as much advantage of design automation as possible in rapidly specifying, simulating, estimating and synthesizing protocol processors. Ultimately this becomes a concern of defining an underlying hardware platform. Such a hardware platform needs to provide a design interface and a design abstraction level that facilitates design automation. These requirements among others lead to choosing the TTA paradigm as the basis for the TACO hardware platform: we saw that the TTA concept should have the necessary potential for providing the kind of modularity and scalability that is essential to automated or semi-automated component library based processor design.

With TTA it is possible to construct a library of functional unit descriptions written in a hardware modeling language. From such a library, specific FUs can be conveniently selected for use in a particular processor architecture instance. Adding new protocol processing functional units into such a library is also quite straight-forward since the communication interface differs very little from one unit type to another. FUs created for an earlier application can be expected to be reusable in later design projects with this kind of a library based approach.

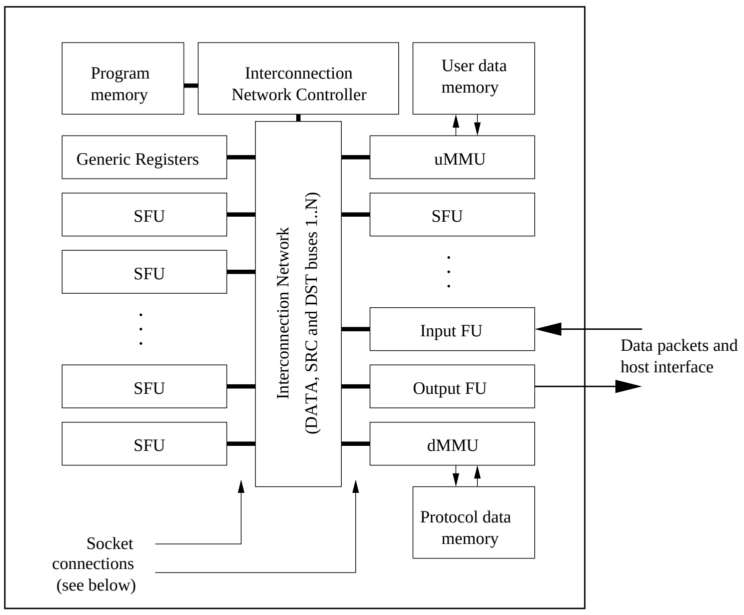
Resorting to TTA also reduces the amount of control logic (and the processing overhead produced by it) needed in a processor, since many traditional hardware tasks are taken care of already at the time of designing the application software; for this reason, for example dynamic code scheduling in hardware is not necessary in the TACO architecture. Scheduling the code in a way that eliminates hardware access conflicts and the need for run-time optimizations and checks is left to the programmer (and/or an intelligent compiler).

Using the terminology defined in [20], TACO processors are constructed solely of SFUs; there are no general purpose FUs (e.g. like ALUs or multipliers) in TACO processors. However, for simplicity, in the rest of this thesis we will use the terms "functional unit" and "FU" when referring to the TACO special functional units. The protocol processing functionality of a given TACO processor is defined by the functional unit configuration of the architecture in question. Each FU is designed and optimized for executing a particular protocol processing task. Figure 3.1 *a)* shows a general overview of the TACO protocol processor architecture. The functional units marked with "SFU" are the protocol processing units. Since their type varies from one protocol and/or application to another, and therefore the types are not specified in this figure. The rest of the functional units have a more specific purpose; there are units for I/O and memory management, and optional generic registers.

For clarity, Figure 3.1 *a)* shows only one user memory management unit (uMMU), protocol data memory management unit (dMMU), Input FU and Output FU. However, the number of these units is not limited in a TACO processor. Naturally the same holds for the memory blocks associated with the aforementioned MMUs. We will return to details of the I/O and memory interfaces later in this chapter.

In our implementations thus far we have used FUs that execute their operations in one machine cycle; this RISC-like feature has made application code scheduling considerably easier. However, the TACO hardware platform (nor the TACO framework) does not in any way limit the possibilities of pipelining FU execution, and in future projects we expect FU pipelining to become necessary in order to ensure support for increasing operation complexity. Based on the findings made in Chapter 2, TACO processors support only unsigned arithmetic to reduce hardware complexity.

a)



Program memory
Interconnection Network Controller
User data memory
Generic Registers
uMMU
SFU
SFU
SFU
SFU
SFU
Interconnection Network (DATA, SRC and DST buses 1..N)
Input FU
Output FU
dMMU
Data packets and host interface
Protocol data memory
Socket connections (see below)

b)

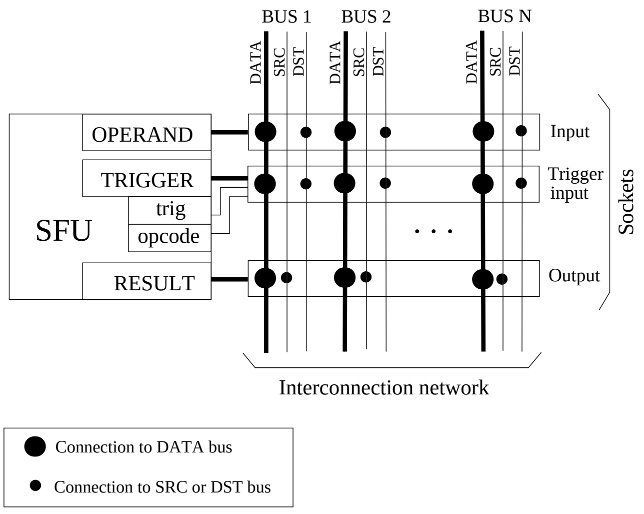


Figure 3.1: Overview of the TACO protocol processor architecture. a) general overview, b) overview of the FU-socket-bus interface. *SFUs* are special protocol processing functional units, *OPERAND*, *TRIGGER* and *RESULT* are data registers, *trig* is used for triggering FU operations, and *opcode* is used for FU operation selection.

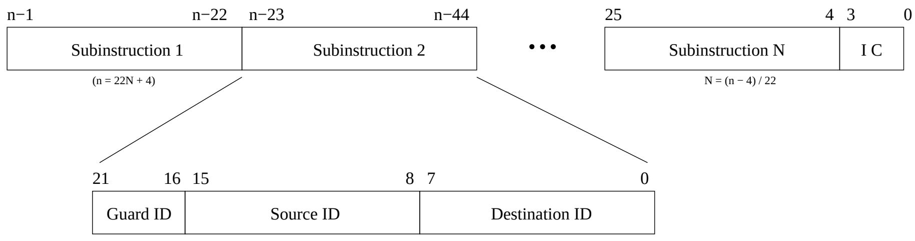


Figure 3.2: TACO protocol processor instruction word (N buses).

3.2.1 Interconnection Network

Figure 3.1 *b)* shows a more detailed view of the connections between functional units and the interconnection network. There can be more than one operand and result registers in a functional unit. Each FU register requires its own socket to connect to the interconnection network. The interconnection network is formed of one or more data buses and the same number of SRC and DST buses. Data buses carry data between FUs. SRC buses carry source addresses (i.e. which register in an FU should send data onto the data bus corresponding to the SRC bus), and similarly DST buses carry destination addresses (i.e. which register in an FU should receive data from the data bus corresponding to the DST bus). The number of possible data moves in one clock cycle is equal to the number of data buses in the interconnection network.

The TACO interconnection network can be fully connected (i.e. all buses have connections to all sockets), or partially connected. However, in our implementations thus far we have resorted to fully connected interconnection networks; full connectivity of the network makes automated hardware and software generation less demanding and ensures maximal use of bus bandwidth: with partial connectivity, situations in which a bus is idle, but can not be used due to a lack of necessary connections, may arise.

3.2.2 TACO Instruction Word

The TACO processor architecture is not limited to specific bus configurations or data word lengths. However, for each architecture instance the processor designer has to make decisions on the number of buses in the interconnection network, and on the data word length of the processor. The decision on the number of buses is directly linked to the instruction word length of the processor: the instruction word must be long enough to provide source and destination identifiers to all the buses.

As seen in Figure 3.2, a TACO instruction word consists of subinstructions

Guard ID	Boolean	ASM code	Guard ID	Boolean	ASM code
000000	a	a:src > dst	001000	e	e:src > dst
000001	$\neg$a	!a:src > dst	001001	$\neg$e	!e:src > dst
000010	b	b:src > dst	001010	a $\wedge$ b	a.b:src > dst
000011	$\neg$b	!b:src > dst	001011	a $\wedge$ ($\neg$b)	a.!b:src > dst
000100	c	c:src > dst	001100	($\neg$a) $\wedge$ b	!a.b:src > dst
000101	$\neg$c	!c:src > dst	001101	($\neg$a) $\wedge$ ($\neg$b)	!a.!b:src > dst
000110	d	d:src > dst	001110	($\neg$a) $\vee$ ($\neg$c)	!a/!c:src > dst
000111	$\neg$d	!d:src > dst	111111	TRUE	src > dst

Table 3.1: Example of using Guard IDs in TACO assembler commands, with corresponding Boolean expressions. a, b, c, d and e are guard signals from FUs. Boolean operations: $\neg$ negation (NOT), $\wedge$ conjunction (AND), $\vee$ disjunction (OR).

and an immediate control (IC) field. Each subinstruction specifies a data move for its corresponding bus (subinstruction 1 defines a data move for bus 1 and so on). The subinstructions are constructed of a Guard ID (GID), a Source ID (SRC) and a Destination ID (DST). The source and destination IDs define the addresses from/to which data is moved. The addresses refer to registers in functional units.

In the TACO hardware platform the lengths of the GID, SRC, DST and IC fields are fully parameterizable. However, in all our implementations thus far we have adopted 4 bits for IC, 8 bits for SRC and DST, and 6 bits for GID. With these values a processor with one bus in the interconnection network would have an instruction word length of 26 bits, whereas a processor with 6 buses would have an instruction word length of 136 bits.

The assembler notation for TACO subinstructions is seen in Table 3.1. Basically the assembler representation of a subinstruction is constructed of the optional guard expression (discussed below in "Guard IDs"), the SRC ID, a greater-than sign (>) indicating move direction, and finally the DST ID.

Immediate integers and IC bits TACO processors support short immediate integer generation. The immediate integer size is equal to the length of the SRC subinstruction field. Immediate integers are detected in program code by decoding the four IC bits in the TACO instruction word (see Figure 3.2). These bits specify the subinstruction (and hence the bus) that contains an immediate integer in place of an SRC identifier. Thus, dispatching a short immediate integer does not have an effect on the number of available data transports per cycle. The suggested way of using larger integer data values in TACO processors is to initialize the required number of User data memory locations (see Figure 3.1 *a)*) with the needed values at compile time. In TACO assembler notation immediate values are differentiated from SRC values by placing a "+" sign in front of the value in the assembler command; e.g. `+25 > dst` indicates that the immediate integer 25 should be dispatched to the destination indicated by `dst`.

Guard IDs The guard ID (GID) is used in defining conditional execution: if a non-zero GID exists in a subinstruction, the data move specified by SRC and DST may be carried out only if the logical condition specified by the GID exists in the processor. Such a logical condition could be for example a boolean false result from two specified functional unit operations. The functional units report these logical conditions to the interconnection network controller by using special one-bit signals called "guard signals". If for example six bit Guard IDs are used, it is possible to define up to 64 different logical conditions for conditional execution. A Guard ID of all ones (i.e. the value "TRUE") indicates that no guard evaluation is needed and the specified move is allowed regardless of the guard signal values.

Table 3.1 gives an example of using guard IDs in assembler notations of TACO subinstructions. A negative condition is expressed using the exclamation mark (!), a Boolean AND using the period (.) and a Boolean OR using the slash (/).

3.2.3 Interconnection Network Controller

The structure of the interconnection network controller is reasonably simple because it does not include any logic for execution optimization, e.g. dynamic scheduling. The instruction scheduling is done already at the assembler code level, since the assembler code itself is a list of data moves. Thus, instruction decoding in TACO processors is very simple; it more or less becomes the task of splitting the long instruction words into SRC and DST addresses, and dispatching these addresses. The cost of this simplicity and efficiency in instruction decoding is an increase in the amount of program memory needed (due to the length of the instruction word). The interconnection network controller has no support for operating system functions such as virtual memory and multitasking.

The key tasks the Interconnection Network Controller performs are:

- fetching instructions from the Program memory
- maintaining the Program Counter (PC)
- evaluating guard signals and guard IDs for conditional execution

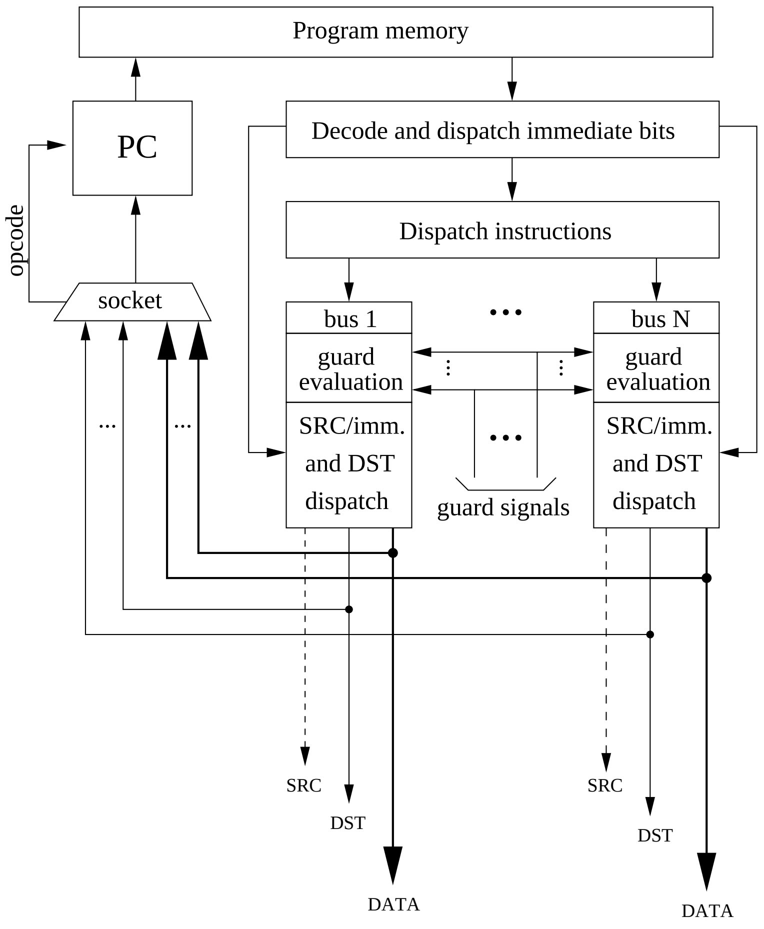


Figure 3.3: Functional view of the interconnection network controller in a processor with N buses.

- splitting long instruction words into subinstructions
- dispatching subinstructions onto the buses
- extracting and dispatching immediate integers specified in program code

Figure 3.3 shows a functional view of the network controller. The network controller retrieves a long TTA instruction word from the program memory. Then, the long instruction word is divided into subinstructions for each bus. We recall from earlier that each subinstruction consists of a source address (SRC), a destination address (DST) and a guard expression (GID). If the guard bits are all ones, a guard expression has not been specified. If the guard bits specify some other value, the programmer has specified a conditional data move. In this case, the guard expression is compared to the values of the guard signals from the functional units (see Figure 3.3). If the guard expression is satisfied by the guard signals, or if there is no guard expression, the execution of the subinstruction is allowed. At this point, the SRC and DST values are written onto the SRC and DST address buses.

If the Network Controller detects immediate control bits that specify an immediate integer, the SRC value of the specified subinstruction is treated as a data value instead of a socket address. This value is dispatched on the corresponding DATA bus, and a zero is dispatched on the corresponding SRC address bus.

Four stage pipeline Instruction execution in TACO processors is carried out in four pipeline stages:

- *fetch*: In this stage the next instruction is fetched from program memory.
- *decode*: This stage consists of two steps: in the first step source and destination identifiers are put onto the instruction buses, and in the second step sockets decode the identifiers locally. If there is a match between a hard-coded identifier and an identifier in the instruction bus, the socket stores the result of the decode process to be used in the next pipe stage (i.e. the socket becomes enabled).
- *move*: In this stage FUs with enabled sockets write/store data to/from the buses.
- *execute*: In this stage the FUs execute their operations.

The pipeline is shown in Figure 3.4.

Program counter The interconnection network controller is also responsible for maintaining, updating and loading the program counter (PC). For the loading functionality the network controller has a built-in trigger socket. This makes jumps in program code possible. The program counter socket has three logical triggers:

- TAPC: Program counter is loaded with the value specified as the trigger data (TR). The resulting action is an absolute jump to the specified program code line (PC = TR).
- TUPC: Program counter is incremented by the value specified as the trigger data (TR). The resulting action is a relative PC increment (PC = PC + TR).

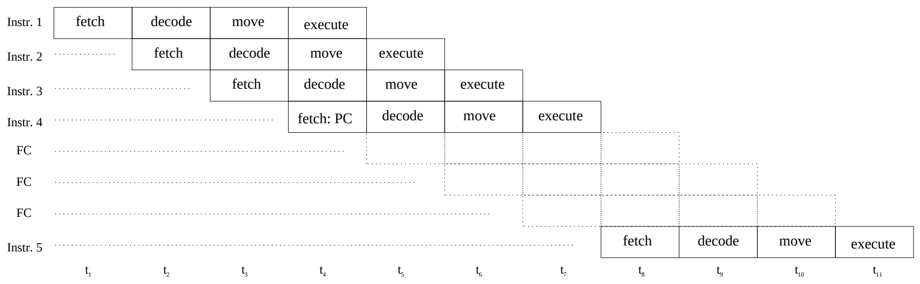


Figure 3.4: TACO protocol processor pipeline and its operation during programmed jumps. A programmed jump is detected in instruction 4 (labeled *fetch: PC*). *FC* = fetch cancelled.

- TDPC: Program counter is decremented by the value specified as the trigger data (TR). The resulting action is a relative PC decrement (PC = PC - TR).

The program counter socket addresses occupy the last three locations in the address space for move destinations (DST addresses).

Programmed jumps also require some pipeline management. Figure 3.4 shows how the pipeline is emptied when programmed jumps occur. When the network controller detects that one of the subinstructions in the long TACO instruction word is a program counter load, it does not allow further instruction fetches for three machine cycles (i.e. the pipeline is locked for three cycles). This delay is needed for the already pipelined subinstructions (N active subinstructions when there are N buses in the interconnection network) to finish. The program counter load is performed in the execute stage, since program counter loads from the functional units are possible.

3.2.4 Sockets

Functional units are connected to the interconnection network through input, trigger and output sockets as seen in Figure 3.1 *b)*. In TACO processors each input and output socket has one hardcoded logical identifier (address), and each trigger socket has at least one hardcoded identifier. Multiple identifiers in trigger sockets are used to specify opcodes for functional units that are able to perform more than one operation on the input data (e.g. a boolean evaluation unit: operations like "=", "≥", "≤", ...). Multiple logical identifiers belonging to a particular trigger socket have consecutive integer values, so opcode extraction is done by subtracting the value of the first hard-coded logical identifier from the identifier read from the DST bus. This opcode is dispatched as a four-bit signal (integer value 0..15) to the functional unit,

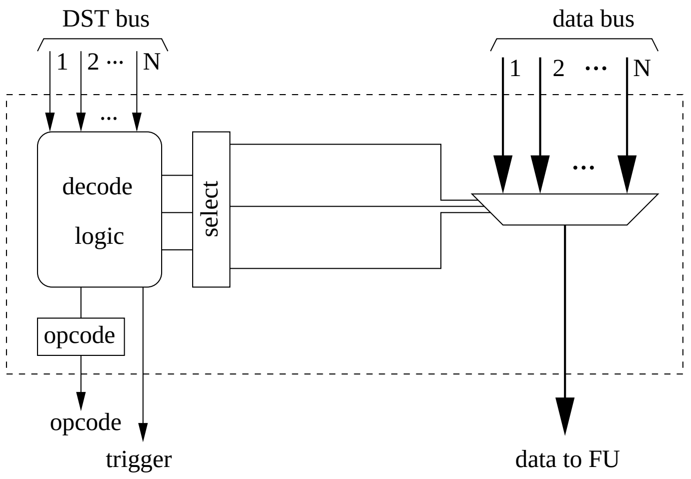


Figure 3.5: Implementation of a TACO trigger socket. The implementation of an Input socket lacks the opcode and trigger characteristics, but is otherwise exactly the same.

and the functional unit performs the operation corresponding to the opcode. Naturally, only one hard-coded identifier per socket can be addressed during one cycle.

Input and Trigger Sockets

In the TACO architecture an input or a trigger socket can have an active connection to only one bus at a time, i.e. it can not pass data from multiple buses towards its target FU register during one cycle. Input sockets do not include any logic for managing resource conflicts, i.e. situations in which the same socket is addressed from multiple sources. Such situations are expected to be dealt with and prevented already in the application software design process (i.e. programming and scheduling).

Input sockets decode destination addresses from the DST buses. If a DST address on one of the buses matches one of the hard-coded logical identifiers, the corresponding bus ID is stored in a select register (see Figure 3.5). If there is no match, the value zero is stored. On the next machine cycle, if there is a non-zero value in the select register, a connection between the selected data bus and the receiving register in the functional unit is opened.

Trigger sockets (Figure 3.5) function like regular input sockets except for two additional pieces of functionality:

- A trigger socket always signals its host FU to start executing its operation when data is written through the socket into the corresponding

FU register. This is implemented using a one bit trigger signal.

- A trigger socket always extracts an opcode from the DST IDs. The extracted opcode is passed to the FU when the FU is triggered.

Output Sockets

Output sockets decode source addresses (SRC) just as the input and trigger sockets decode DST addresses. A data connection is opened between the corresponding FU register and ALL the data buses for which the decode process found a match.

The output socket implementation is very similar to that of the input socket. The differences are that the direction of data flow is opposite, and an output socket can open a connection between the FU register and multiple data buses.

3.2.5 Functional Units

In this section we discuss the specifications for existing TACO functional unit descriptions in the TACO module libraries. All protocol processing FUs described in this section have been optimized for a specific protocol processing task: some of the FU operations have been chosen for hardware implementation based on the findings made in Chapter 2, and others based on application analyses carried out in later case studies such as the ones discussed in Chapter 5. Protocol application implementations using these FUs can be expected to achieve better execution times, smaller silicon areas or lower power consumption than a general purpose processor implementation with similar design constraints due to hardware execution of frequently needed functionality.

Figure 3.6 shows the general structure of all FUs. For simplicity, there is only one input operand register and one output result register (and corresponding sockets) pictured. However, many FUs actually have several operand inputs and result outputs. Still, there is always only one physical trigger register in an FU. If the FU in question supports more than one operation, the trigger socket connected to the trigger register is able to recognize more than one logical address from the DST buses as described in section 3.2.4. The trigger socket also triggers the FU operation by raising a one-bit signal; this is the signal connected to T in Figure 3.6.

The FU operation resides in the combinatorial logic part of Figure 3.6. There is no limit for the number of FUs of the same kind in a processor. If the application to be implemented requires the same operation frequently, improved performance can be achieved through FU replication (having two or more of the same kind of FUs in a processor). Some functional units also have

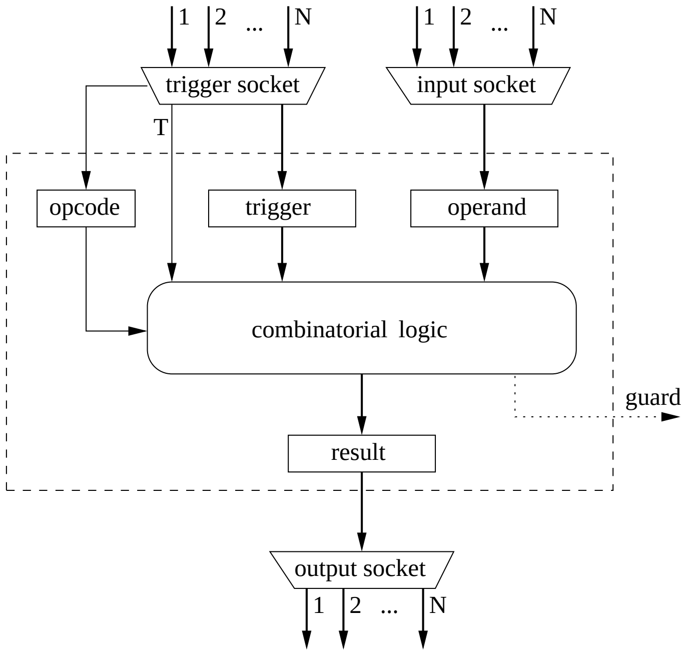


Figure 3.6: General structure of the functional units. Note that there can be (and often is) more than one operand inputs and result outputs. *Trigger, operand* and *result* are FU data registers, *T* is used for triggering FU operations, and *opcode* is used for FU operation selection. The optional *guard* signal is used for guard evaluation in the network controller.

a guard signal (result bit signal) connected directly to the network controller (*guard* in Figure 3.6). These signals are used when the network controller is evaluating guard bits, i.e. logical conditions for conditional execution. Guard bits were discussed earlier in sections 3.2.2 and 3.2.3.

Most of the functional units are parameterizable in terms of data word length. However, we have conducted all our case studies of the TACO architecture using 32-bit data word length. Thus, in the following discussion we describe the TACO functional units as they would be implemented using 32-bit data word length. Also, we will use the following symbols to describe Boolean operations: $\neg$ negation (NOT), $\wedge$ conjunction (AND), $\vee$ disjunction (OR).

Comparator FU

Comparators are used for Boolean comparisons between specified data words.

Interface:

- Operand register OP (input operand type)
- Trigger register TR (input trigger type)
  Eight opcodes
- Result register R (output result type)
- Has a guard signal to network controller
- No external connections

Operation:

```
GuardBitSignal = 0;          // Reset guard bit
R = 0;                       // Reset result

switch(opCode) {
  case 0:  If TR == OP {     // TEQ: TR equal to OP
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
  }
  break;
  case 1: If TR != OP {      // TNEQ: TR not equal to OP
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
  }
  break;
  case 2: If TR > 0 {        // TGZ: TR greater than 0
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
  }
  break;
  case 3: If TR == 0 {       // TEQZ: TR equal to 0
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
  }
  break;
  case 4: If TR <= OP {      // TLEQ: TR less than or equal to OP
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
  }
  break;
  case 5: If TR < OP {       // TLT: TR less than OP
     R = MaxInt;             // MaxInt = binary all ones data word
     GuardBitSignal = 1;     // Raise guard signal
```

```
      }
      break;
      case 6: If TR >= OP {        // TGEQ: TR greater than or equal to OP
         R = MaxInt;               // MaxInt = binary all ones data word
         GuardBitSignal = 1;       // Raise guard signal
      }
      break;
      case 7: If TR > OP {         // TGT: TR greater than OP
         R = MaxInt;               // MaxInt = binary all ones data word
         GuardBitSignal = 1;       // Raise guard signal
      }
      break;
   }
```

Functional description of operation: The value written into the trigger register (**TR**) is compared to the value already stored in the operand (**OP**) register. The type of comparison that is carried out depends on the opcode received from the trigger socket. If the comparison result is true, an all-ones value is stored into the result register (**R**), and the guard signal is raised. If the result is false, zero is given in the result register, and the guard signal is reset to zero.

Example: A comparator unit has been assigned logical socket addresses 51..58 (decimal notation). The value 100 has already been stored into the operand register (**OP**). The value 99 is written into the trigger register (**TR**), using the socket address 57. The opcode is 57-51 = 6, which corresponds to the"$\geq$" operation (TGEQ). Since the expression "$99 \geq 100$" is false, the guard signal is reset to zero and the value zero is stored into the result register (**R**).

Masker FU

Maskers are used to replace specified bits in a data word with other bits.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  One opcode
- Result register R (output result type)

- No guard signal
- No external connections

Operation: `R = (OP ∧ OD) ∨ (TR ∧¬OP);`

Functional description of operation: Any part(s) of the data word given in the trigger register (**TR**) are replaced with bit sequences defined by a mask (**OP**) and another data word (**OD**).

Example: The original data word is 1100 0101 0011 and is input to the trigger register **TR**. The 0101 sequence in the middle needs to be changed to 1010. Thus, we define the mask **OP** = 0000 1111 0000, in which a zero indicates a bit in the original data word that should not be modified, and a one indicates a bit that should be modified. Then, as the replacement bitstring we transport the data word 0110 1010 0110 to the **OD** register. In this data word the first and last four bits can (in this case) be either ones or zeros without effecting the outcome of the operation. Now, according to the function given above, we first calculate **OP** ∧ **OD** = 0000 1010 0000. Then, we calculate **TR** ∧ ¬**OP** = 1100 0000 0011. Finally, we do an **OR** between these minterms and obtain **R** = 1100 1010 0011.

Shifter FU

The Shifter FU is used for shifting data words left or right a given amount of bit positions. The unit can also be used for moving a specified number of most significant bits to least significant bits and zeroing the rest of the bits.

Interface:

- Operand register OP (input operand type)
- Trigger register TR (input trigger type)

  Three opcodes
- Result register R (output result type)
- Has a guard signal to network controller
- No external connections

Operation:

```
if ((OP > 32) | (OP < 1)) R = 0;
else {
  switch(opCode) {
  case 0:        // TLR, logical right shift
    R = TR.range(31,OP);
    GuardBitSignal = TR(OP - 1);
    for(int idx = 0; idx < OP; idx++){
      R[31-idx]='0';
    }
    break;
  case 1:        // TLL, logical left shift
    R.range(31,OP) = TR.range(31-OP,0);
    GuardBitSignal = TR(31 - OP + 1);
    for(int idx = 0; idx < OP; idx++){
      R[idx]='0';
    }
    break;
  case 2:    // TML, shift n MSBs to n LSBs, zero rest of bits
    R.range(OP-1,0) = TR.range(31,32-OP);
    GuardBitSignal = TR(31-OP);
    for (int idx = OP; idx < 32; idx++){
      R[idx] = '0';
    }
    break;
  }
}
```

If the operand value is 32, only update the guard signal to reflect the last removed bit.

Functional description of operation: With opcodes 0 and 1, the value given in the trigger register (**TR**) is shifted logically left or right (depending on the logical trigger address used) as many positions as defined by the the value given in the operand register (**OP**). The value of the guard signal is equal to the last removed bit: e.g. in a left shift with 5 positions, bit 27 is the last removed bit. In a right shift with 5 positions, bit 4 is the last removed bit. If the value in **OP** is greater than or equal to 32, the result given in **R** will be zero.

With opcode 2, a specified number of most significant bits in a data word become the least significant bits of the result. The rest of the bits are zeroed. The original data word is input in the trigger register (**TR**), and the number of bits to move is specified in the operand register (**OP**). The resulting data word is placed into the result register **R**.

Examples: TLR: the original data word is 1100 0101 0011. This value is given as trigger (**TR**). The programmer wishes to perform a logical right shift for 4 positions, so the logical trigger used is TLR (opcode 0). The value 4 is stored into the operand register (**OP**). The result of the logical right shift is then 0000 1100 0101.

TML: the original data word is 1100 0101 0011. This value is given as trigger (**TR**). The programmer wishes to move the three most significant bits to least significant bits and zero the rest of the bits in the data word. The logical trigger used is TML (opcode 2). The value 3 is stored into the operand register (**OP**). The result of the operation is 0000 0000 0110.

Matcher FU

Matchers are used to determine whether specified bits exist at specified locations inside a data word.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)
  One opcode
- Result register R (output result type)
- Has a guard signal to network controller
- No external connections

Operation:

```
R = 0; GuardBitSignal = 0; // Reset result and guard
R = (¬ OP ∨ OD ∨ TR) ∧ (OP ∨ ¬ OD ∨ ¬ TR);
If R == MaxInt GuardBitSignal = 1;
                          // MaxInt = binary all ones data word
```

Functional description of operation: The Matcher FU operation specified above is derived from the bitstring matching circuit suggested in [67]. The operand (**OP**) and data (**OD**) registers specify a bit pattern (range of bits and their values). This pattern is compared to the data word given in the trigger register (**TR**). If the pattern matches the corresponding portion of the data word in the trigger register, the guard signal is raised (i.e. the result of operation is true). **OP** contains the correctly aligned bit pattern(s) that need to be matched in the **TR** value, and **OD** contains the correctly aligned negation(s) of the desired bit pattern(s). Irrelevant bits are indicated by ones in both **OP** and **OD**.

Example: The original data word is 1100 0101 0011 and is given in the trigger register **TR**. The 0101 sequence in the middle is the one that needs to be matched. Thus, we define the operands **OP** = 1111 0101 1111 and **OD** = 1111 1010 1111. The bit positions with the value one in both **OP** and **OD** have been specified irrelevant for the evaluation.

Now, according to the function specified for the operation, we first evaluate $\neg\mathbf{OP} \vee \mathbf{OD} \vee \mathbf{TR}$ = 1111 1111 1111. Thereafter we evaluate $\mathbf{OP} \vee \neg\mathbf{OD} \vee \neg\mathbf{TR}$ = 1111 1111 1111. Both evaluations resulted in data words of all ones. Thus, also the final **AND** results in all ones which indicates a true result (the desired bit pattern was found). An all-one data word is written into the result register **R** and the guard signal is raised.

Had the value stored in **TR** been 1100 1101 0011, the first maxterm of the match equation would still have produced all ones. However, the second maxterm would have been 1111 0111 1111. This would have caused the final **AND** to produce a data word not consisting of all ones, indicating a false result.

Internet Checksum FU

The Internet Checksum FU is used to calculate Internet checksums as specified in [11, 94].

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  Two opcodes
- Result register R (output result type)
- No guard signal
- No external connections

Operation:

- Opcode 0, TRC: Reset checksum (initialize for new calculation)
- opcode 1, TCC: Calculate checksum:

  ```
  A = OP_MSW + OP_LSW + OD_MSW + OD_LSW + TR_MSW + TR_LSW + R';
  B = A_MSW + A_LSW;
  R' = B + B_carry;
  R = ¬R';
  ```

  R' is a 16-bit internal register used for storing the cumulative one's complement sum. **A** represents a 32-bit summation stage, **B** is a 16-bit summation stage. **MSW** and **LSW** indicate most and least significant 16-bit words, respectively.

Functional description of operation: The Internet checksum is calculated using 16-bit one's complement summation. The Internet Checksum FU has an internal register (**R'** in the above operation specification) for storing a temporary result needed in consequtive calculations.

For a new checksum calculation, **R'** is initially zeroed. The datagram for which the checksum is calculated is fed into the Internet Checksum FU as 32-bit words, three words at a time (32-bit words to **OP**, **OD** and **TR**). The FU splits the inputs into six 16-bit words and sums them up with the value in **R'** using 32-bit summation: the **MSW** of the result will then contain a sum of the carry bits, and the **LSW** will contain the 16-bit sum. To convert the result to a one's complement sum, the carry bits are added to the 16-bit sum in the **LSW** (stage **B** in the TCC trigger operation specification). The carry from this summation is again added to the result, and the result is stored into **R'**. The negation of this value is placed into the result register **R**.

This FU could also have been implemented with only one input register. However, with three input registers up to three data words per clock cycle can be transported into the FU for Internet checksum calculation. Depending on the application and its data transport requirements, with this FU implementation checksum calculation speed can be up to 300 % higher than with an FU that would have only one input register.

Example: In the following, for the sake of simplicity in representation, we consider a four-bit checksum calculated from three eight-bit inputs. **R'** already contains a value, which needs to be included in the calculation.

$\mathbf{R'} = 1011, \mathbf{OP} = 1010\ 0101, \mathbf{OD} = 1110\ 0111, \mathbf{TR} = 1100\ 0011$
$\mathbf{A} = 1011\ +\ 1010\ +\ 0101\ +\ 1110\ +\ 0111\ +\ 1100\ +\ 0011 = 11\ 1110$

$\mathbf{B} = 11\ +\ 1110 = 1\ 0001$
$\mathbf{R}' = 0001\ +\ 1 = 0010$
$\mathbf{R} = \neg\mathbf{R}' = 1101,$

which is the result to be placed into the result register **R**. The value in **R'** is needed when the next two input data words are processed.

Counter FU

Counters are used for counting up and down a specified number of positions.

Interface:

- Trigger register TR (input trigger type)
  Three opcodes
- Result register R (output result type)
- Guard signal
- No external connections

Operation:

```
GuardBitSignal = 0;          // Reset guard bit
R = 0;                       // Reset result
switch(opCode) {
  case 0:  R = TR;          // TSC: Set Counter
  break;
  case 1:
     R = R + TR;             // TIC: Increment Counter
     If R == 0
       GuardBitSignal = 1;   // Raise guard if counter zero
  break;
  case 2:
     R = R - TR;             // TDC: Decrement Counter
     If R == 0
       GuardBitSignal = 1;   // Raise guard if counter zero
  break;
}
```

Functional description of operation: Before the counter unit can be used, it has to be initialized by writing a start value to the trigger register (**TR**) using the logical trigger **TSC**. Then, whenever necessary, the counter is incremented or decremented using the logical triggers **TIC** and **TDC**. The data value written into **TR** is added to or subtracted from the value currently output as result in the result register (**R**). If the new value is zero, the result bit is raised.

Example: The counter is initialized to the value 10 by writing an immediate integer into the address that corresponds to the logical trigger **TSC**. Then, in the following cycles, the value 1 is written into the address corresponding to the logical trigger **TDC**. After 10 writes (or 10 cycles in this case), the result is zero, and the guard signal is raised.

HEC FU

A HEC FU is needed in ATM processing to calculate the ATM Header Error Check checksum (CRC-8, generator polynomial $G(x) = x^8 + x^2 + x + 1$) as described in Chapter 2.

Interface:

- Trigger register TR (input trigger type)

  One opcode

- Result register R (output result type)

- No external connections

Operation: Calculate CRC-8 for the 32-bit data word given as input in **TR**, output result in **R**. The benefit of designing a CRC module that operates on a specific generator polynomial is that the resulting implementation is efficient in terms of execution time and required silicon area. A run-time parameterizable implementation would become considerably more complex. The VHDL code for the HEC FU's parallel CRC-8 calculation was generated using the Easics CRCTool [30].

Router Local Info FU

The Router Local Info FU provides locality information of an IPv6 router.

Interface:

- Operand register OP (input operand type)

- Operand register OD (input operand type)

- Trigger register TR (input trigger type)

  Eight opcodes

- Result register R (output result type)
- No guard signal
- No external connections

Operation:

- Opcode 0, TMTU: return maximum transmission unit (MTU) size in bytes for an interface

  ```
  R = MTU[TR];
  ```

- Opcode 1, TLLA: return 32-bit part of local link address for an interface

  ```
  R = LLA[TR].range(32·(OP + 1) - 1, 32·OP);
  ```

- Opcode 2, TUNI: return 32-bit part of unicast address for an interface

  ```
  R = UNI[TR].range(32·(OP + 1) - 1, 32·OP);
  ```

- Opcode 3, TCST: return cost for sending a datagram on an interface

  ```
  R = Cost[TR];
  ```

- Opcode 4, TSMTU: store MTU size in bytes for an interface

  ```
  MTU[TR] = OD;
  ```

- Opcode 5, TSLLA: store 32-bit part of local link address for an interface

  ```
  LLA[TR].range(32·(OP + 1) - 1, 32·OP) = OD;
  ```

- Opcode 6, TSUNI: store 32-bit part of unicast address for an interface

  ```
  UNI[TR].range(32·(OP + 1) - 1, 32·OP) = OD;
  ```

- Opcode 7, TSCST: store cost for sending a datagram on an interface

  ```
  Cost[TR] = OD;
  ```

The 128-bit words are input and output 32 bits per cycle. The value in the operand register (**OP**) specifies the ordinal number of the 32-bit data word that is to be input or output (3 indicates the most significant and 0 the least significant 32-bit word of the 128-bit value), and the value in the trigger register (**TR**) specifies the local interface in question. The **OD** register specifies the data to be written when storing information into the unit.

Local link addr.	Local unicast addr.	MTU	Cost
128 bits	128 bits	11 bits	4 bits

Table 3.2: Data organization per interface in the Router Local Info FU.

Functional description of operation: The Local Info unit is used for accessing and updating information regarding the router and its interfaces. The values stored into or read from the Local Info unit are used in the process of making routing decisions. The internal data organization of the Local Info unit is shown in Table 3.2. The storage space provided by the Local Info unit is implemented using a small dedicated memory block for each of the table columns. The Local Info unit is thus actually a special memory management unit. The maximum number of entries in the local information table depends on the chosen memory blocks.

Example: Reading the local link address for interface 3 takes five cycles.

- cycle 1: write 3 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 2: read first 32 bits of the address from **R**, write 2 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 3: read next 32 bits of the address from **R**, write 1 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 4: read next 32 bits of the address from **R**, write 0 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 5: read last 32 bits of the address from **R**

Routing table FU

The Routing Table FU is used for storing and accessing the routing table in an IPv6 Router.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  Nine opcodes

- Result register R (output result type)
- No guard signal
- No external connections

Operation:

- Opcode 0, TRN: return number of entries (prefixes) in table
- Opcode 1, TRP: return 32-bit part of 128-bit prefix;
  prefix specified by **TR**, part specified by **OP** (3 = MSW, 0 = LSW).
  ```
  R = prefix[TR].range(32·(OP + 1) - 1, 32·OP);
  ```
- Opcode 2, TRL: return length of prefix (8 bits) specified by **TR**
  ```
  R = prefixLength[TR];
  ```
- Opcode 3, TRI: return interface ID (8 bits) specified by **TR**
  ```
  R = interface[TR];
  ```
- Opcode 4, TRM: return metric (8 bits) for interface specified by **TR**
  ```
  R = metric[TR];
  ```
- Opcode 5, TSRP: store 32-bit part of 128-bit prefix;
  prefix specified by **TR**, part specified by **OP** (3 = MSW, 0 = LSW),
  value specified by **OD**.
  ```
  prefix[TR].range(32·(OP + 1) - 1, 32·OP) = OD;
  ```
- Opcode 6, TSRL: store length of prefix for a prefix;
  prefix specified by **TR**, value specified by **OD**.
  ```
  prefixLength[TR] = OD;
  ```
- Opcode 7, TSRI: store interface ID (8 bits) for a prefix;
  prefix specified by **TR**, value specified by **OD**.
  ```
  interface[TR] = OD;
  ```
- Opcode 8, TSRM: store metric (8 bits) for prefix specified by **TR**
  ```
  metric[TR] = OD;
  ```

Table 3.3 shows the structure of the internal routing table. The 128-bit words are input and output 32 bits per cycle. The value in the operand register (**OP**) specifies the ordinal number of the 32-bit data word that is to be input or output (3 indicates the most significant and 0 the least significant 32-bit word of the 128-bit value), and the value in the trigger register (**TR**) specifies the ID of the prefix in question. The **OD** register specifies the data to be written when storing information into the unit.

Prefix	Prefix Length	Interface ID	Metric
128 bits	8 bits	4 bits	4 bits

Table 3.3: Data organization for the routing table accessed by the Routing Table Unit.

Functional description of operation: The Routing Table unit is used for accessing and updating routing table information. The routing table is implemented using a small dedicated memory block for each of the table columns. The routing table unit is thus actually a special memory management unit. The maximum number of entries in the routing table depends on the chosen memory blocks.

Example: The most significant 32-bit word of a prefix corresponds to the ordinal number 3 and the least significant word to the ordinal number 0. Thus, to store the second 32-bit word of a 128-bit IPv6 address as the next hop address for prefix 5:

- The value 2 is input into the operand register (**OP**). This corresponds to the second most significant 32-bit word of the address.
- The 32-bit data word is input into the data register (**OD**).
- Finally, the value 5 is input into the trigger register (**TR**) using the **TSRH** logical trigger identifier (opcode 8).

ICMPv6 FU

The ICMPv6 FU is used for generating header information for ICMPv6 messages in an IPv6 router.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  One opcode
- One Result register R (output result type)
- No guard signal
- No external connections

Operation:
```
R.range(31,24)= OP.range(7,0);
R.range(23,16)= OD.range(7,0);
R.range(15,0)= TR.range(15,0);
```

Functional description of operation: This unit is used to construct the first of the two 32-bit data words needed for creating an ICMPv6 header. The second word is directly written into the memory, since it is directly obtainable from the Local Info FU (MTU for target interface) or through an immediate integer (pointer to erroneus field).

R is the first 32-bit word of the ICMPv6 header.

Contents of **OP** (MSB..LSB): Unused (24 b), Type (8 b)
Contents of **OD** (MSB..LSB): Unused (24 b), Code (8 b)
Contents of **TR** (MSB..LSB): Unused (16 b), Checksum (16 b)

Contents of **R** (MSB..LSB): Type (8 b), Code (8 b), checksum (16 b)

Example: An ICMPv6 message "Packet too large" needs to be sent. The value "2" is written into **OP** (type), the value "0" into **OD** (code) and the IPv6 checksum value (let us choose 0x5A for this example) from the checksum unit to **TR**. On the next cycle, the value 0x0200005A is output from **R**.

Memory Management FUs

The Memory Management FUs are used for accessing the different memory blocks in a TACO processor. The protocol data memory management FU additionally provides DMA access for the Input and Output FU.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)
  Two opcodes
- Result register R (output result type)
- User memory management unit: no guard signal

- Protocol data memory management unit: two guard signals
- No external connections
- dMMU has DMA interfaces for input and output FUs; this functionality is discussed in section 3.5. uMMU has no DMA interfaces.

Operation:

- Opcode 0, TRMM: read from memory

  Read data word from memory address [**OP**+**TR**]
  (OP is base address, TR is offset).

- Opcode 1, TWMM: write to memory

  Write data in **OD** to memory address [**OP**+**TR**]
  (OP is base address, TR is offset).

Functional description of operation: The memory management FUs are used by other FUs to access the memories. The memories the MMUs are connected to are fast enough to provide one memory access per clock cycle, thus an MMU can provide its result in one clock cycle. The memory address to be accessed is determined by adding the *offset* value from the trigger register to the *base* value from the operand register. The MMUs access the memory from the calculated *base+offset* address when triggered.

Both MMU types provide mechanisms for reading and writing data into/from their corresponding memory blocks. There can be more than one of each kind of MMU in a TACO architecture. uMMUs are used for storing and retrieving user data. Since a uMMU provides its result in one clock cycle, no general purpose registers are needed for variables and constants (however, such registers can optionally be included in a TACO architecture). The user memory locations can be initialized to specific values (e.g. constants needed by the application) at the time of application software design. Figure 3.7 shows a functional view of a uMMU. uMMU replication makes it possible to serve several memory accesses (to separate memory blocks) in parallel, provided that sufficient data transport capacity is available (i.e. there are several data buses available for use in the interconnection network).

dMMUs are used for storing and accessing PDUs. In addition to normal access through the interconnection network (like uMMUs), dMMUs also provide DMA access to their corresponding memory blocks for the Input and Output FUs. During DMA access the dMMU raises one of its guard signals to indicate it is busy either reading from or writing to its memory block in DMA mode. A Functional view of a dMMU is given in Figure 3.8. For clarity, the figure shows separate diagrams for memory writes (*a)*) and memory reads

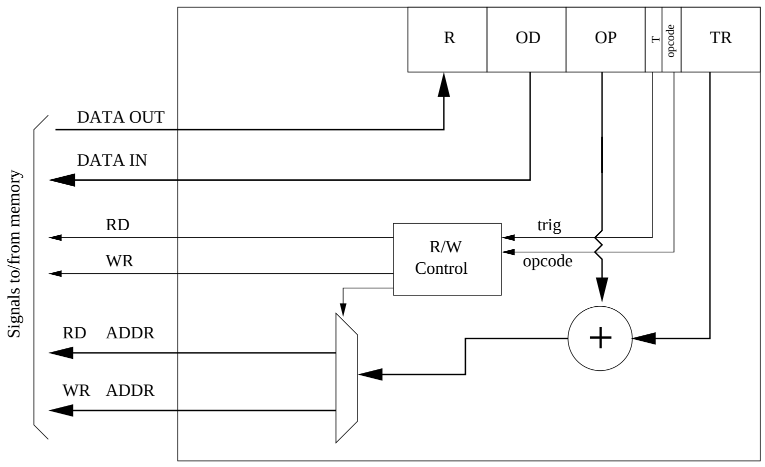


Figure 3.7: Functional view of a user memory management unit (uMMU). *R, OD, OP* and *TR* are FU data registers, *T* is used for triggering MMU operations, and *opcode* is used for read/write selection.

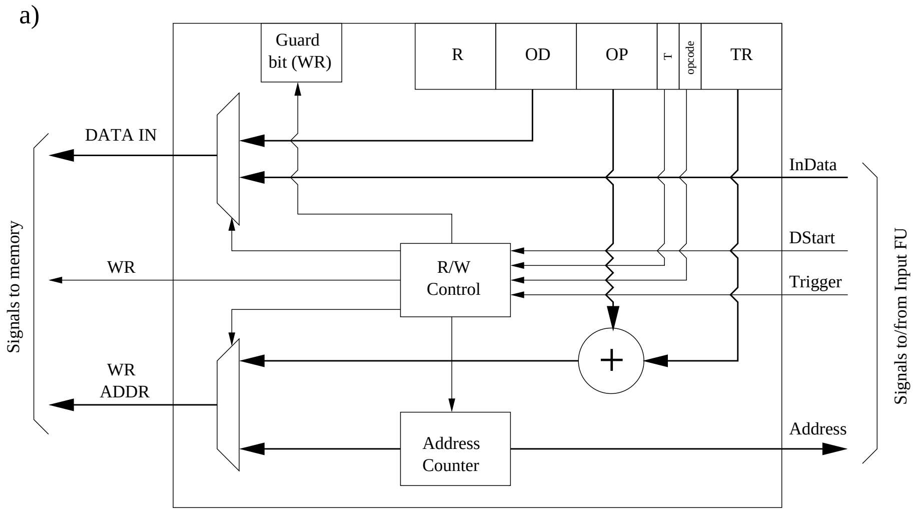
(*b)*). The dMMU maintains a counter for determining the next memory address usable for DMA access from the Input FU (*Address counter* in Figure 3.8 *a)*). This counter is initially zero, and after reaching a value corresponding to the highest possible memory address in the corresponding memory block it is again zeroed. In other words, DMA access uses the protocol data memory in a circular fashion, overwriting the oldest (and obsolete) data when necessary.

Replicating dMMUs is not useful unless also a matching pair of Input and Output FUs is replicated. The benefits of such replication are unclear; in all the TACO experiments we have carried out so far we have included only one dMMU-Input FU-Output FU set into the architectures. We will discuss the dMMUs as part of a TACO processor's I/O interface in section 3.5 later in this chapter.

Example: To read data from memory address 150 with the base address set to 128 (i.e. the value 128 is already stored in **OP**), the value 22 is written into **TR** using the logical trigger TRMM (opcode 0).

To write data to memory address 150 with the base address set to 128 (i.e. the value 128 is stored in **OP**), the data value to be stored into the memory location is written into the data register **OD**, and the value 22 is written into **TR** using the logical trigger TWMM (opcode 1).

a)



b)

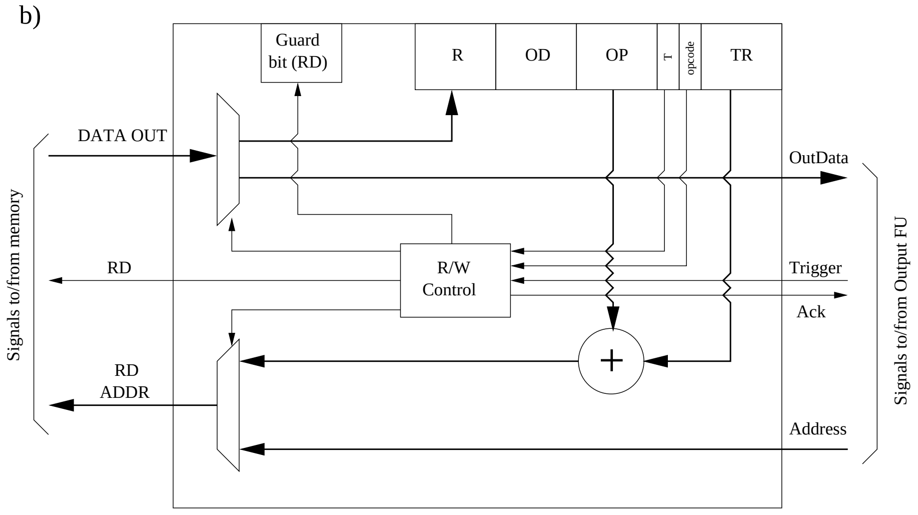


Figure 3.8: Functional view of the protocol data memory management unit (dMMU). *a)* Writing to memory, *b)* reading from memory. *R, OD, OP* and *TR* are FU data registers, *T* is used for triggering MMU operations, and *opcode* is used for read/write selection. For DMA writes, *DStart* indicates the beginning of a new PDU, and *Trigger* is used for requesting DMA access. For DMA reads, *Trigger* is used for requesting DMA access and *Ack* is used for granting it.

Input FU

The Input FU maintains information on incoming PDUs, and stores them in the protocol data memory. Together with the Output FU and the Protocol Data Memory Management FU it forms the TACO I/O interface described in section 3.5 of this chapter.

Interface:

- Trigger register TR (input trigger type)
  One opcode
- Three result registers R1, R2, R3 (output result type)
- Guard signal
- External connections; discussed in section 3.5.

Operation: When triggered, write the oldest entry in the

- Memory address FIFO to R1
- External interface ID FIFO to R2
- PDU length FIFO to R3

The data value used in triggering (i.e. data moved to **TR**) has no relevance; the trigger register is used only for triggering the unit.

Functional description of operation: The Input FU acts as a triple read-only FIFO for FUs that access it through the interconnection network. For each incoming PDU it holds the starting memory address, the ID of the interface the PDU came from (if there is more than one network interface), and the length of the PDU. When the Input FU is triggered, it places the oldest entries in its three FIFOs into corresponding result registers (**R1, R2, R3**).

Example: To get the starting memory address, input interface ID and PDU length for the oldest PDU in the protocol data memory, write any non-zero value to the trigger register. The PDU information is given in the result registers **R1, R2** and **R3**.

Output FU

The Output FU sends outgoing PDUs from the protocol data memory. Together with the Input FU and the Protocol Data Memory Management FU it forms the TACO I/O interface described later in this chapter.

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)
  One opcode
- No result registers
- No guard signal
- External connections; discussed in section 3.5.

Operation: When triggered, add the value in

- **OP** to the Memory address FIFO
- **OD** to the PDU length FIFO
- **TR** to the External interface ID FIFO

Functional description of operation: The Output FU acts as a triple write-only FIFO for FUs that access it through the interconnection network. For each outgoing (i.e. processed) PDU, it holds the starting memory address, the ID of the interface the PDU should be sent to (if there is more than one network interface), and the length of the PDU. When the Output FU is triggered, it places the information given in the input registers into its FIFOs.

Example: To store the starting memory address, output interface ID and PDU length for an outgoing PDU, the corresponding data words are written into the three input registers.

3.3 Assembler Programming of TACO Processors

TACO processors are programmed by specifying a source address (SRC) and a destination address (DST) for each bus in the interconnection network. SRC and DST are typically registers in functional units. The move from SRC to DST can be made conditional by specifying a Guard ID as explained earlier in this chapter. An operation is executed on the target functional unit if the register indicated by DST is a trigger register. We recall that in TACO processors the maximum number of data moves that can be made during one clock cycle equals the number of buses in the interconnection network. Thus,

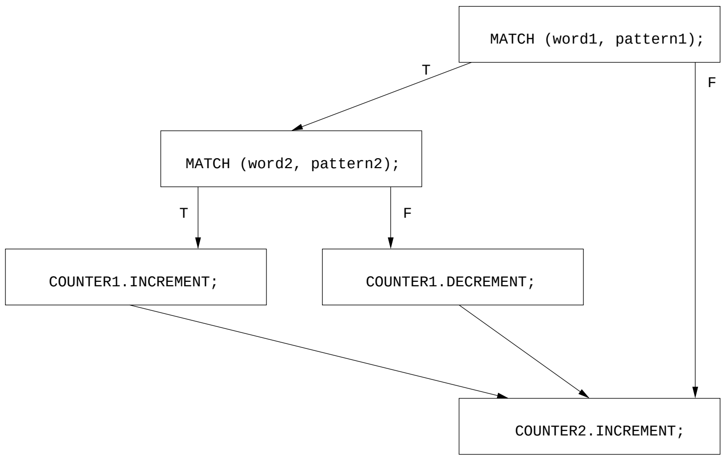


Figure 3.9: Algorithm used in the programming example.

if there is more than one bus in the interconnection network, the architecture in question supports instruction level parallelism (ILP) in code execution.

As a programming example we will consider the algorithm in Figure 3.9. Clearly, the processing of this algorithm would require up to five cycles to complete in traditional sequential programming (assuming each operation takes one instruction cycle to execute).

If we consider a TACO processor with three buses, two counter FUs and two matcher FUs (already holding the match patterns *pattern1* and *pattern2* as operands), we have the following assembler code for this particular TACO processor (each line represents one 70 bit TACO instruction, i.e. is executed in one instruction cycle):

```
word1 > TM1; word2 > TM2; 1 > TIC2;
a.b:1 > TIC1; a.!b:1 > TDC1;

// TM1, TM2 = trigger IDs for Matchers 1 and 2
// TIC1, TDC1, TIC2 = trigger IDs for Counters 1 and 2
// (I=increment, D=Decrement)
// a, b = true result guard expressions for Matchers 1 and 2
// !b = false result guard expression for Matcher 2
```

The assembler code for this algorithm requires five data moves, and can be executed in two instruction cycles in the TACO architecture in question. In the code, both of the match operations are carried out in parallel. The third move in the first cycle is used for updating *Counter 2* (in the original algorithm *Counter 2* is updated at the end of the algorithm, but updating it earlier does not have an effect on the outcome of the algorithm). *Counter 1* is incremented in the second cycle only if both match results are true, and decremented, if the result from *Matcher 1* is true and the result from *Matcher 2* is false. If the result from *Matcher 1* is false, *Counter 1* is not updated. Note the utilization of conditional (or guarded) transports.

3.4 Memory Configuration

As stated earlier, TACO processors can have several uMMUs and dMMUs with associated memory blocks. In addition, the processors have a program memory block which is managed by the Interconnection Network Controller. Besides these, also the FU registers are addressable locations of TACO processors. Figure 3.10 shows a summary of the possible addressable locations in TACO processors.

If optional general purpose registers are included in a TACO architecture, their addresses are included in the FU input and output register address spaces. The optional routing table and local info units have their own address spaces for storing routing information. For all the separate address spaces in TACO processors the addressing starts from address 0. The size of each address space is fully parameterizable. If the short immediate integers specified using the SRC subinstruction field are not long enough to absolutely specify a memory address in a particular memory block, a user memory block can be initialized to specific long integer memory address values that can be retrieved using short integer addressing. In addition, all the memory management units fully support *base-offset*-type addressing.

In our TACO processor designs so far we have implemented support for including SRAM (static RAM) as on-chip memory. On-chip memory is most often produced into a layout at the time of manufacturing the chip. The memory manufacturer provides information of the necessary signals for using the memory block, and a simulation model of the memory. The designer can choose the word sizes etc., but is not able to modify the actual memory implementation. Since the detailed memory interface is not known until the memory/chip manufacturer has been chosen, we can not design a memory interface unit that would be compatible with any on-chip memory IP block. However, since the SRAM memory interface is quite simple, not much design effort is needed to connect such a memory block onto a TACO processor.

SRAMs provide excellent performance with some cost on power consumption. In most TACO designs the target clock speed is below the memory access speed of a modern SRAM cache memory block. Thus, one memory access per clock cycle can be executed. Choosing SRAM for the memory type also makes it possible to use a third party processor for fast memory access

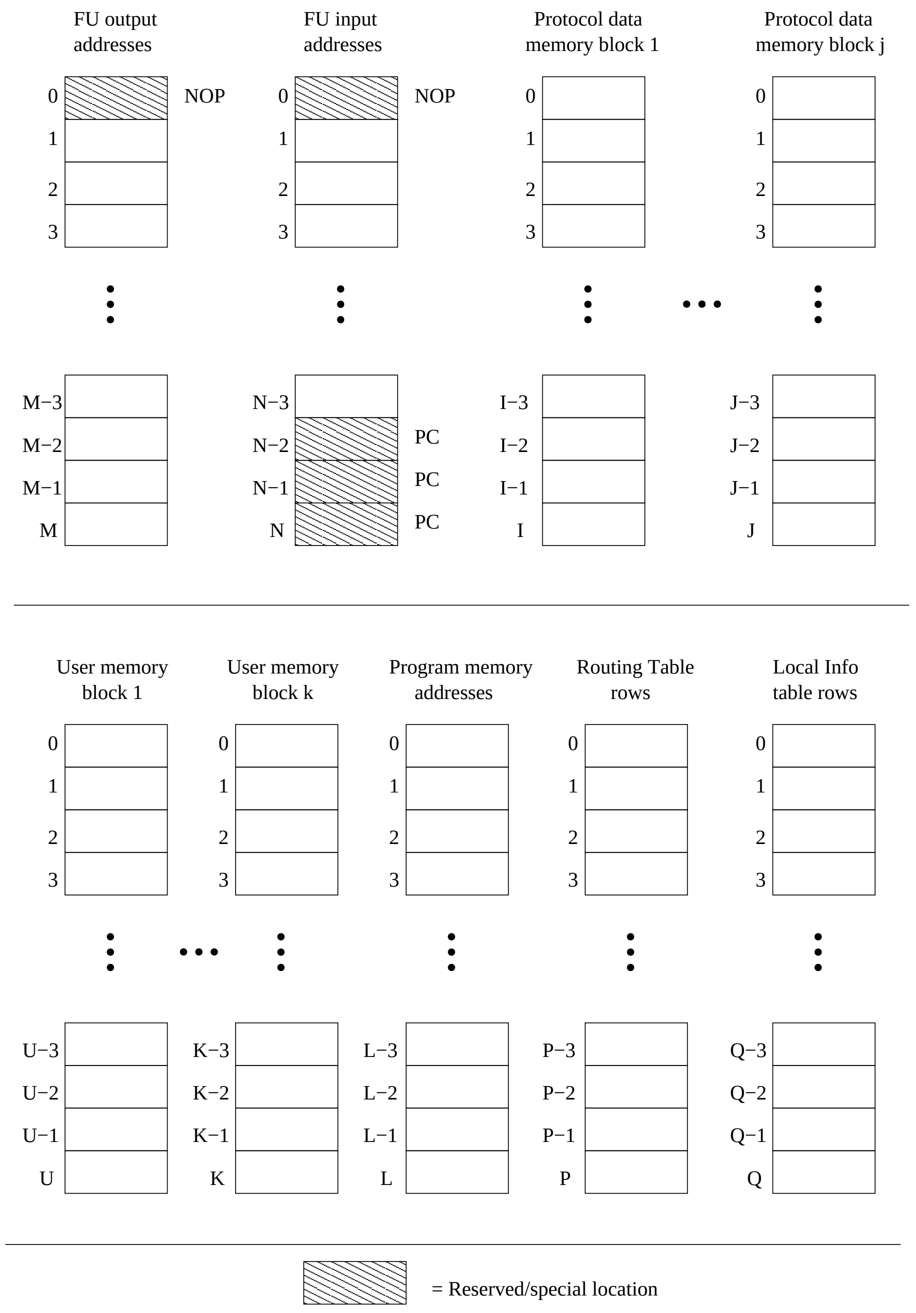


Figure 3.10: Addressable locations in TACO processors.
I, J, K, L, M, N, P, Q, U, j and *k*: parameterizable values.
NOP: location reserved for "no operation" instructions.
PC: location reserved for maintaining and updating the program counter.

and table lookup. One such processor is the iFlow address processor [82], designed to act as a co-processor for speeding up internet routing table lookups. The host network processor sees the iFlow processor as SRAM, and reads from and writes to the iFlow processor using typical SRAM mechanisms.

3.5 Input/Output Interfaces

There are two kinds of I/O communication needed in TACO processors:

1. Reading data from and writing data to the network,
2. Communicating with a host processor, or with on-chip IP (Intellectual Property) blocks (if necessary).

The mechanisms for these tasks can be designed individually to suit the needs and the functioning environment of a certain protocol processing application, or a generic solution can be used. Application-specific solutions usually provide better performance at the cost of interconnectibility. In the following pages we will discuss some possible solutions for both kinds of I/O tasks.

Connection to Network

The network interface of any device is defined by the type of the physical network medium. In copper-wired networks it is usually necessary to enhance, filter and convert the incoming signal before it can be interpreted as digital data words. In optical networks this task is often simpler, since usually the incoming signal only needs to be converted from optical to electrical form. In any case, it is only after these conversions that the actual protocol data is ready for analysis. Figure 3.11 shows some possible tasks that need to be carried out in copper-wired and optical networks before the received data is ready for processing.

In TACO processors the I/O functionality is produced jointly by three functional units: the dMMU (protocol Data Memory Management Unit), the Input FU and the Output FU. Since the tasks needed to be performed on signals coming from and going to a copper-wired network vary from one type of physical medium to another and one type of protocol to another, placing the entire signal processing into one FU would require a separate FU for each kind of physical medium and communications protocol. For this reason, the standard Input and Output FUs do not concern themselves with management of the physical connection. Naturally protocol-specific FUs with all the necessary functionality for the particular protocol's physical connection could also be constructed; this way additional circuitry required by an off-the-shelf standard interface would not be needed.

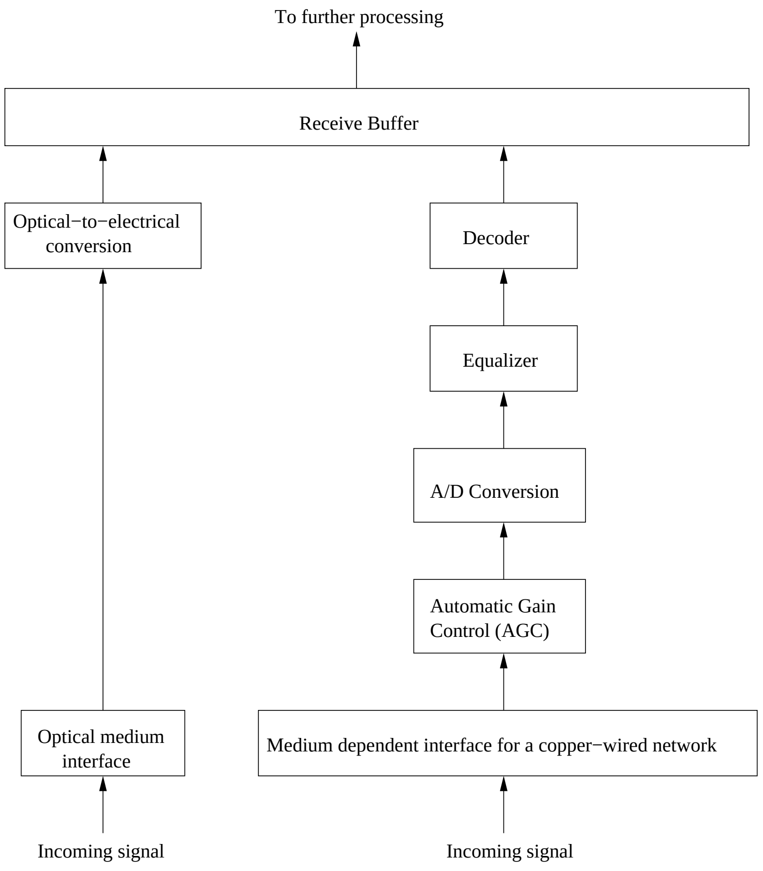


Figure 3.11: Block diagram of possible tasks in preparing a signal from the network for data processing.

If a TACO processor is used as a part of a Network-on-Chip (NoC) device, it is also possible to use the NoC interface for PDU input and output. A NoC interface for the TACO hardware platform is discussed at the end of this chapter.

Generic Solution For TACO processors, the generic solution for network I/O is to connect the Input and Output FUs directly to a send/receive buffer. The generic I/O interface is shown in Figure 3.12. The send/receive buffer may be e.g. a buffer in an Ethernet device used for storing deframed data coming in from the network. The Input and Output FUs have DMA access to the protocol data memory, and they are also connected to the interconnection network (like all other FUs).

If incoming data from the send/receive buffer can be written into the

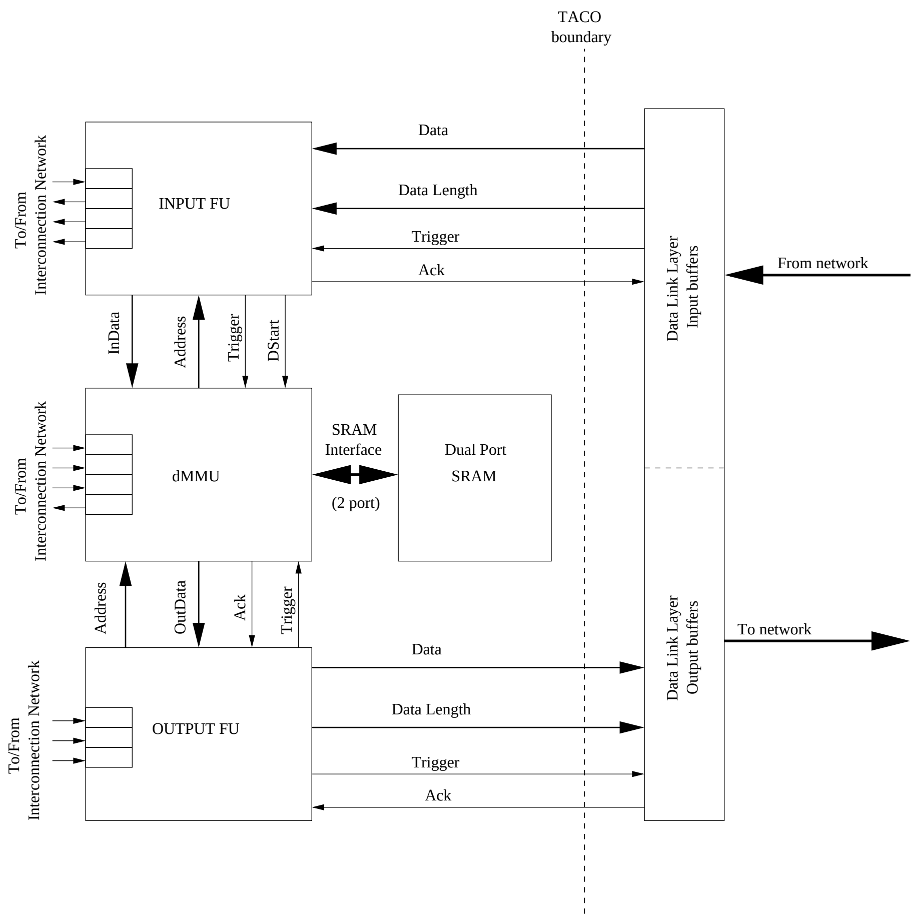


Figure 3.12: A generic network interface for TACO processors. Thick arrows indicate signals with processor word width, thin lines indicate one-bit signals.

memory, the Input FU passes the data to the dMMU for storing it into the memory. The *Trigger* signal (see Figure 3.12) is used to request DMA transfer to the protocol data memory through the dMMU, and the *DStart* signal informs the dMMU that the next data word to be transferred is the first word of a new PDU. Information like the base memory address of the incoming protocol data unit (PDU) as indicated by the dMMU is stored into the Input FU. While the PDU is being written into the memory, its processing can be started by the other functional units through the interconnection network assuming dual-port memory is used (dual-port memory can simultaneously be read from and written to as long as the operations do not access the same address). The other functional units access the PDU residing in the protocol data memory using the dMMU. The starting memory addresses of PDUs are stored in the Input FU.

To write modified data back into the protocol data memory, the functional units accessing the memory through the interconnection network need to either wait for the incoming PDU write to finish, or to use the user data memory for temporary storage. This of course depends on the application and its program code implementation.

Once the PDU has been processed, the Output FU is informed of an outbound PDU. The Output FU accesses the protocol data memory from a given base address and sends out a given number of data words. The memory space taken up by a PDU is released as soon as the PDU has been processed (forwarded, consumed or discarded). In our implementations this far the Output FU has blocked other read accesses to the protocol data memory while it sends a processed PDU. While the Output FU is sending a PDU, a new PDU can simultaneously be written into the protocol data memory through the Input FU and its processing can start immediately.

So far the above memory access scheme has not caused problems for us in terms of performance. Due to the basic nature of communication protocols, PDUs are in practice always of reasonable size (no matter which protocol is used). So, transferring PDUs does not block the rest of the processing for too long. Also, in copper-wired networks PDU processing can usually be carried out at a higher speed than the maximum speed at which the PDUs can appear at the inputs of the processor (however, this might not be the case in optical networks). The reason we have implemented the memory access this way is to be able to use dual-port memory; if the Input and Output FUs and the interconnection network should all have simultaneous access to the protocol data memory, a four-port memory implementation would be needed. Another solution that could be applied here would be to design and implement an equal-priority arbitration scheme into the dMMU. Naturally, a four-port access scheme would still be the fastest alternative, if it could be implemented at the same speed as two-port access.

Custom solutions In our first protocol processor design experiment, the ATM AIS processing case study (discussed in Chapter 5 of this thesis), we used a custom solution for PDU I/O. The solution was based on the standard solution described above, but a pre-processing circuit was implemented between the inbound receive buffer and the Input FU. The pre-processing circuit synchronizes to incoming ATM data with the synchronization method described in Chapter 2 and thus also verifies the ATM cells. Once synchronized, the pre-processing circuit allows valid cells to be accessed by the Input FU. The VHDL code for the CRC-8 calculation needed in the pre-processing circuit was generated using the Easics CRCTool [30]. For outgoing cells, the standard TACO solution described above was used.

Also in our IPv6 routing experiments like e.g. the one discussed in Chapter

5 we have used a custom solution for PDU I/O. This solution is actually more of an extension to the generic TACO I/O interface of Figure 3.12: the protocol data memory is organized into "slots" consisting of 387 32-bit words. Each slot provides enough space for a 1500-octet IPv6 datagram and an additional pair of IPv6 and ICMPv6 headers. This way ICMPv6 messages can be constructed in the memory simply by adding the necessary headers to the beginning of a slot containing the IPv6 datagram to be sent as the content of an ICMPv6 message. For normal processing, the memory is simply accessed with an offset that bypasses the reserved ICMPv6 memory allocation. This approach can with little effort be modified for use with most packet-exchange protocols when necessary, although modifications are needed on a per-protocol basis (especially in terms of packet size).

Connection to a Host Processor or On-Chip IP

A TACO processor can operate in a system in one of three alternative ways:

1. As a stand-alone processor,
2. As a stand-alone co-processor,
3. As an IP block in a System-on-Chip or a Network-on-Chip device.

For the latter two, a connection and communication mechanism of some sort is needed. As a stand-alone co-processor we face another point of decision: whether the TACO processor should support the co-processor interface of a specific family of host processors, or if it should provide a generic interface to basically any kind of host processors.

If the decision is to use a TACO protocol processor as a stand-alone co-processor for a specific host family, the interconnection can be designed in a more optimal way to support only the needed communication between the two processor families. This type of connection is used in e.g. early Intel x86 CPUs and their x87 math co-processors, and Texas Instruments DSPs and TI MSP430 series microcontrollers. A good choice for this kind of connection might be something similar to what is used in the TI processors - their HPI (Host Processor Interface) communication resembles the fast path - slow path approach needed in protocol processing (fast path: DSP calculations, slow path: system control by the microcontroller). Although this kind of an approach would be advantageous in terms of performance, by using such an interface the TACO processors would no longer be able to function as generic co-processors.

For a generic interface to a multitude of host processors an industry standard interface is needed. Again, the interface solutions are different for stand-alone co-processors and for SoC/NoC IPs. For stand-alone processors, a generic external interface is needed, whereas for SoCs and NoCs the structures inside the chip used for interconnecting the IP blocks depend on the designer. For a stand-alone co-processor, an industry-standard approach would be to implement the PCI (Peripheral Component Interconnect) bus. PCI is widely supported by most modern general purpose controllers and processors as well as special purpose processors like The Intel IXP1200, Motorola PowerQUICC and Texas Instruments DSPs. PCI support would require a special FU into TACO protocol processors that converts the data from the processor into a format suitable for PCI, and that would manage the PCI communication independently.

At the time of writing, we have not yet decided to implement a bus interface for stand-alone use; instead, our focus has been on designing and implementing a SoC/NoC on-chip bus interface for the TACO architecture. The TACO VCI compliant on-chip bus interface is discussed below.

3.5.1 VCI Compliant Network-on-Chip Interface

To use a TACO processor as an IP block in a SoC or NoC device, a standard on-chip bus interface is needed. Recently we were invited to participate in a NoC design co-operation project within a national research programme in Finland [3]. We were asked to provide an IPv6 client implementation of our TACO protocol processor architecture for inclusion in a multimedia processing NoC platform. The IPv6 client core was required to be BVCI (Basic VCI, Basic Virtual Component Interface) [118] compliant (the IPv6 client is discussed in more detail in Chapter 5). Thus, we proceeded to specify, design and implement a BVCI interface module into our protocol processor architecture.

We recall that data and control I/O in TACO processors is normally managed by two FUs, namely the Input and Output FUs. These two units accompanied with the protocol data MMU (dMMU) and the protocol data memory form the I/O subsystem in TACO processors as described in the previous section (see Figure 3.12). This I/O subsystem is available as a library component in the TACO libraries, and can thus be conveniently included in any TACO architecture instance. With this in mind, there were two major design alternatives for implementing BVCI support into the TACO architecture:

1. Remove the input and output FUs and replace them with BVCI compliant FUs.

2. Design a wrapper module that maps signals from the existing FUs to BVCI signals.

A decision was made to proceed with alternative **2.** This way we would not have to redesign the existing FUs and we would be loyal to one of the key

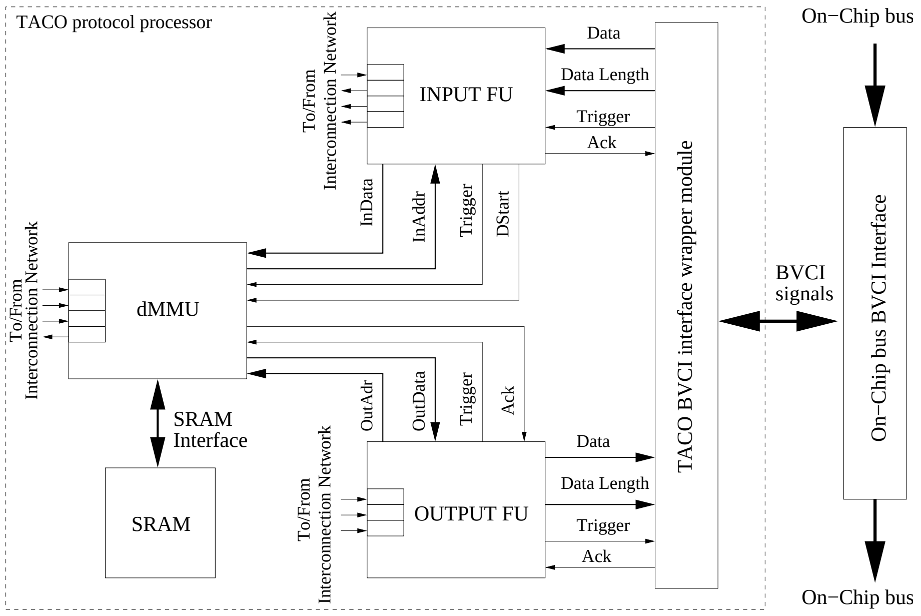


Figure 3.13: Connections between the interconnection network, internal protocol data memory, input and output FUs and the BVCI interface in a TACO processor.

principles within the TACO design framework: modularity. With a wrapper module BVCI support could be included into an architecture when necessary, and it could be left out of architectures that do not require NoC bus compliancy. Figure 3.13 shows the TACO I/O interface discussed earlier, enhanced with a BVCI wrapper module.

Figure 3.14 shows the internal structure of the TACO VCI wrapper module. On the input side (left side of Figure 3.14), the TACO protocol processor acts as a VCI target and the VCI interface on the on-chip network side acts as a VCI initiator. The communication is started by the initiator raising the CMDVAL signal and asserting certain other VCI signals (e.g. WDATA, CMD, CLEN). On CMDVAL, the TACO interface wrapper checks that the correct VCI command is issued. If the command is incorrect, the wrapper responds by generating a VCI error packet to the on-chip originator of the erroneous data. If the command is correct, the wrapper raises the trigger signal towards the Input FU. If the Input FU responds with an Ack, the CMDACK signal is raised, and data transfer from the on-chip network to the TACO dMMU begins.

On the output side (right side of Figure 3.14), the TACO wrapper acts as the VCI initiator. When the Output FU is ready to send data to an on-chip IP, it raises the trigger signal. The wrapper responds with an Ack towards

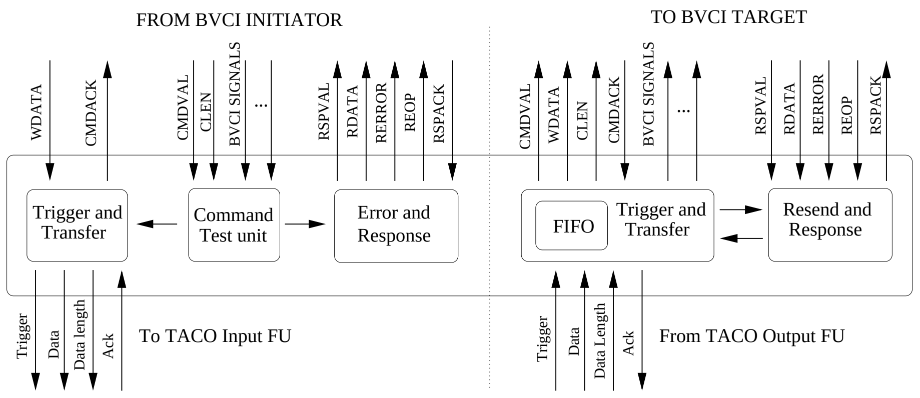


Figure 3.14: Functional block diagram of the TACO NoC wrapper module.

the Output FU if there is space left on the output FIFO of the wrapper. If the FIFO is full, the Ack is not asserted. The FIFO size is parameterizable, and is typically set to match the packet size of the on-chip network. With at least one item in the FIFO, the wrapper raises the CMDVAL signal towards the VCI interface on the on-chip network side. As soon as a CMDACK signal is detected by the wrapper, data from the FIFO is transferred over the VCI interface to the target.

The TACO IPv6 Client processor for which the NoC interface was originally designed is discussed in more detail in Chapter 5.

3.6 Chapter Summary

This chapter discussed the TACO architecture, a modular and scalable TTA-based hardware platform for the TACO protocol processor design framework. TTA was chosen to be the underlying base hardware solution for the TACO framework some time after seeing H. Corporaal's TTA presentation in the Tampere System-on-Chip seminar in 1999. Until then, no architecture had been found suitable for use in the TACO framework. This choice eventually lead to presenting the first TACO architecture instances in a conference paper in 2000 [116]. These initial architecture instances have now evolved to the parameterizable hardware platform discussed in this chapter.

In the TACO architecture each functional unit is optimized for a particular protocol processing task. Some of the tasks have been chosen for hardware implementation based on the findings of Chapter 2, and others based on application analyses carried out in later case studies such as the ones discussed in Chapter 5. There are no general purpose processing elements like multipliers or arithmetic-logic units (ALUs) among the TACO FUs. In this approach the overheads in e.g. processing speed and chip size potentially resulting from general purpose implementations are avoided. This approach and the choice of TTA as the base architecture were also determined to contribute to reduced hardware complexity, especially in control logic. The vital aspects of the TACO hardware platform, including its control structures, different FU types, programming and I/O interfaces were discussed in detail in this chapter. To our knowledge, the TACO hardware platform is the first and so far the only approach in which the TTA paradigm is applied to protocol processing. Similarly, to our knowledge the memory organization and access scheme of the TACO hardware platform is unique in comparison to existing TTA implementations.

The *MOVE* framework TTA template [20] is without a doubt currently the best known actualization of the TTA paradigm and as such is a de-facto point of reference. Below we summarize the key architectural differences between TACO processors and the *MOVE* framework TTA template of [20].

- TACO processors have support only for unsigned arithmetic due to the characteristics of protocol processing. *MOVE* TTAs have support for signed arithmetic. The benefit of resorting exclusively to unsigned arithmetic is that the resulting hardware is simpler.

- Unlike *MOVE* TTA, TACO processors do not have control buses. This means that the signals Global Lock (GL), Local Lock Request (LL) and Squash (SQ) do not exist in TACO processors. The functionality provided by these signals is provided to TACO processors in part by the application software design process (i.e. instruction scheduling), in part by the four stage pipeline, and in part by the Interconnection Network Controller.

- TACO processors have multiple memories: Program memory (for the program code), Protocol data memory (for storing/retrieving protocol data units), and one or more User data memories (for storing/retrieving user data). Also, In the TACO architecture there are FUs that have direct memory access (i.e. the Input and Output FUs have DMA access to the protocol data memory). *MOVE* TTAs implement a traditional Harvard architecture with separate program and data memories and have no FUs with DMA.

- Each memory block (program, protocol data, one or more user data memory blocks) in a TACO processor has its own address space, whereas in *MOVE* TTAs a common memory address space is implemented.

- All TACO MMUs support base-offset addressing. This is not the case with *MOVE* TTA load-store units.

- TACO processors are not required to have general purpose registers, although they are optional. Most register traffic can be managed by direct data transports between functional units. In addition, User data memories can be used for register-like data access and storage. In comparison, *MOVE* TTAs are required to have a register file and each interconnection bus needs to have access to it in order for the compiler to function.

With a TTA-based hardware platform it is possible to construct a library of functional unit descriptions written in a hardware modeling language. Designing new functional units for use in the TACO platform is quite straightforward since the communication interface differs very little from one unit type to another. Also, FUs created for an earlier application can be reused in later design projects with this kind of a library based approach. In the next chapter we develop this idea of component library based design further to reach a rapid design methodology for protocol processors. The methodology is built of simulation, estimation and synthesis models of the hardware platform described in this chapter, a design tool that integrates the models into a comprehensive processor design environment, techniques for analyzing protocol processing applications, and a documented design flow.

Chapter 4

The TACO Design Methodology

In Chapter 2 it was established that there are foreseeable benefits in designing processors with optimized hardware for protocol processing. This realization lead in Chapter 3 to specifying the TACO hardware platform, a family of processor architectures in which each execution unit is optimized for a particular protocol processing task. The modular structure of the platform was seen to potentially support automated component library based processor modeling.

In this chapter a rapid system level protocol processor design and evaluation methodology is built around the TACO hardware platform. We argue that the tools and techniques of the methodology can reliably be used to reach a gate-level synthesized, application-optimized TACO processor architecture and its program code with an application specification as a starting point. The process includes iterative and rapid design improvement at the system level. We also argue that the methodology is cost-efficient due to a short turn-around time and the ability to use low-end computer systems and affordable or free software for system level simulations and estimations. Finally, we argue that by using object-oriented programming techniques and analysis methods several advantages are gained for hardware design.

TACO designs start from protocol processing application analysis to determine the functionality needed to process the target application. The goal in this process is to find frequently needed functionality in the application, and to map this functionality to existing functional unit descriptions. If no suitable FUs are found for the desired functionality, new FU descriptions are created. This naturally saves time, since all functionality does not need to be implemented from scratch every time. Another benefit of such a library based approach is that FU descriptions already in the library have been verified for simulations, estimations and synthesis; it is enough to verify only FUs that are newly added to the library.

After application analysis the design space is explored by evaluating the quality of different architectural candidates using simulation and estimation models for the hardware platform. This is an iterative process; results of simulations and estimations of architectural candidates for the same target application guide the task of specifying new candidates for the next round of design space exploration. For this kind of an approach, it is obvious that the simulation, estimation and synthesis models for the hardware platform need to be parameterizable and configurable. Otherwise each model would need to be rewritten every time a change is made into an architecture instance to be examined. The parameterization should be carried out at a high level of abstraction to ensure that simulation and estimation models for a set of architectures can be constructed in little time. Also, the simulation and estimation models need to execute fast so that the time-consuming tasks of design space exploration and design quality evaluation can be carried out as time-wise efficiently as possible. Once a suitable (in terms of performance and cost) architecture has been found through this kind of designer-driven design space exploration, a synthesis model for it is configured and the architecture is synthesized.

Although the TACO processor models do not need modifications for each new architecture candidate thanks to their parametric configuration and instantiation capabilities, even the process of configuring the models manually for different architectures requires preciseness and is an error-prone process due to a large number of signals and modules that need to be correctly instantiated and connected. Valuable design time could be lost in evaluating incorrectly specified architectures and in debugging the instantiation files if errors are made in the configuration process. To deal with this issue, the TACO framework relieves the processor designer from manually configuring and instantiating the models by providing a graphical tool that performs these tasks automatically and also assists the designer in design quality evaluation.

We start the discussion in this chapter by specifying the TACO protocol processor design flow. The initial problem to be solved is to specify techniques for analyzing the target application. From this application analysis the flow continues to iterative design space exploration using the design tool and the processor models. Once the flow completes, an optimized hardware architecture, optimized program code and a synthesizable hardware model for the target application have been found. After discussing the design flow we take a look at turn-around times in the TACO methodology. Then we proceed to discussing details of the TACO processor models. Emphasis in this discussion is on the simulation model; for the estimation and synthesis models we limit the discussion to only an overview, since the maintenance and development of these models are topics in research directions pursued by other researchers in the TACO project. After discussing the simulation model we address the use of object oriented programming conventions in hardware design.

The chapter is concluded with a presentation of the TACO design tool that is used for generating instantiation and configuration files for the processor models and for evaluating simulation and estimation results.

4.1 Design Flow

This section discusses the TACO protocol processor design flow, in which an optimized protocol processor architecture, optimized application code for it and a synthesizable model of it are rapidly derived starting from a high level application description. The flow requires application analysis, architecture design, simulation, physical parameter estimation, architecture exploration, design iteration and hardware synthesis. To achieve this, methods for analyzing protocol processing applications need to be defined, and the processor models and the design tool need to be efficiently incorporated into the design flow.

Figure 4.3 a few pages ahead displays a block diagram of the TACO design flow that supports all the design tasks outlined above. It is to be noted that the flow is not a completely automated process; rather it is a set of consecutive procedures the designer needs to follow and complete. The designer is required to make certain design decisions along the way, and to evaluate the quality of candidate solutions using e.g. the results from simulations and physical estimations.

4.1.1 Application Analysis

In the TACO framework protocol processor development is guided by the development of the protocol software. We have already seen in Chapter 2 that commonly used communications protocols exhibit very similar functional characteristics in their processing (see Table 2.1 for a summary of such characteristics), and the functional units in the TACO architecture have been specified in part based on the findings made in Chapter 2. However, knowledge of inter-protocol similarities may not be adequate when optimizing an architecture for a particular protocol processing application; therefore the design process should be such that it makes it easy to identify frequently used operations within the particular target application. Similarly, the target hardware platform should be (and in the case of TACO, it is) such that it allows easy integration of application-specific operations into hardware.

Although a variety of methods could probably be adapted for use within the TACO framework, object-oriented (OO) techniques [29] have been chosen as the basis for TACO application analysis. One of the main advantages provided by object-oriented methods is that since they are focused on iden-

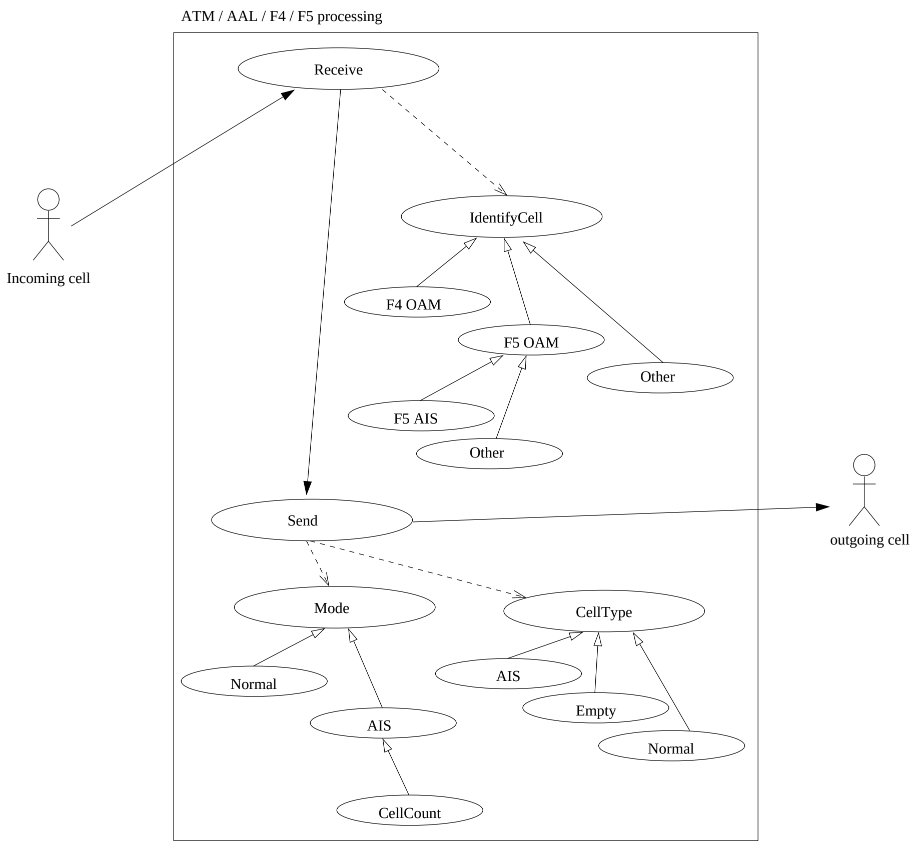
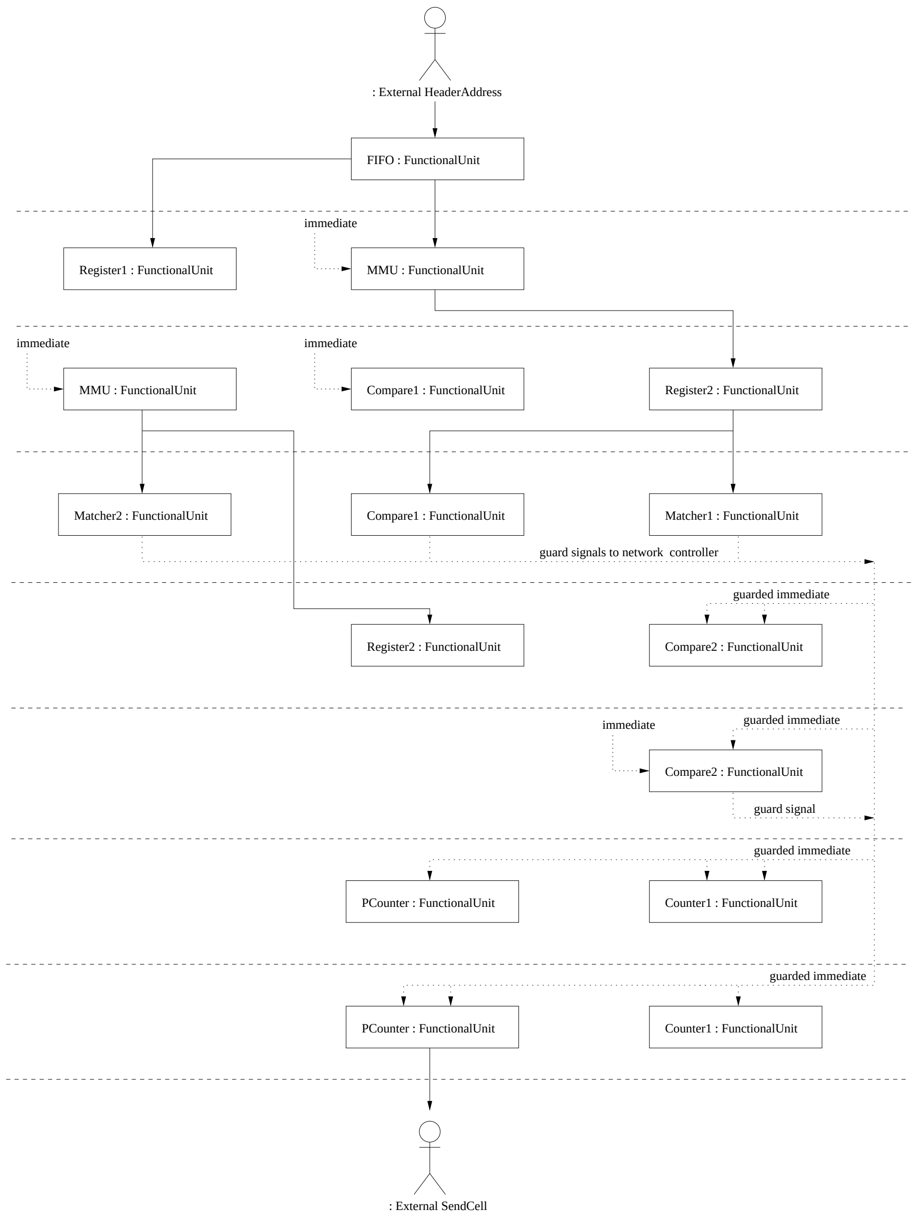


Figure 4.1: Use Case diagram for processing ATM F5 AIS cells.

tifying objects and implementing the functionality of a system using objects, the methods can be easily applied to identification of tasks that could be implemented in hardware as functional units.

In TACO, OO techniques are used so that objects encapsulate state and functionality, while classes describe the common properties of a number of objects. Thus objects and classes are suitable abstractions for modeling protocol processing operations. Many object-oriented methods contain techniques and notations for so called *domain analysis* (identification of common datastructures and functionality within a domain of interest).

As in most modern OO methods, also in the TACO framework application analysis is started by drawing up a use case diagram. The use case diagram shows the main functionality that needs to be implemented. In the context of TACO, use cases are often data transports between service access points (SAPs) of different protocol layers (see Chapter 2 for more information on SAPs). Such a use case could be for example sending a user protocol data unit (PDU), receiving one, or perhaps receiving an operation and maintenance



: External HeaderAddress
FIFO : FunctionalUnit
immediate
Register1 : FunctionalUnit
MMU : FunctionalUnit
immediate
MMU : FunctionalUnit
immediate
Compare1 : FunctionalUnit
Register2 : FunctionalUnit
Matcher2 : FunctionalUnit
Compare1 : FunctionalUnit
Matcher1 : FunctionalUnit
guard signals to network controller
guarded immediate
Register2 : FunctionalUnit
Compare2 : FunctionalUnit
immediate
guarded immediate
Compare2 : FunctionalUnit
guard signal
guarded immediate
PCounter : FunctionalUnit
Counter1 : FunctionalUnit
guarded immediate
PCounter : FunctionalUnit
Counter1 : FunctionalUnit
: External SendCell

Figure 4.2: Collaboration diagram for ATM F5 AIS cell processing. Arrows represent data transfers. Immediate value generation and control signals are managed by the interconnection network controller (not shown in the figure).

(OAM) PDU. As an example, figure 4.1 shows use cases for ATM F5 AIS[1] cell processing drawn using the UML [29] notation. The use case diagram can be used to guide the development process in an iterative way so that in each iteration the implementation of one use case is added to the prototype (e.g. the ROPES process of [29]).

The first object diagrams of the system are obtained from the use cases. The analysis then continues with refinement of the object diagrams until operations that are suitable for hardware implementation are identified (*domain analysis*). Once this process finishes, knowledge of required functional unit types has been acquired, and the hardware part of the application analysis is completed.

Once the tasks that could be performed in hardware (and implemented as TACO functional units) have been identified, the original application and its data dependencies need to be mapped to the required set of functional units. In this aspect of the application analysis, collaboration diagrams [29] showing the interactions between the different functional units are constructed. In terms of the TACO framework, it is possible to take the application analysis further by refining the collaboration diagrams to a level at which timing is taken into account and at which each interaction in the diagram corresponds to a data move between registers in functional units. Figure 4.2 shows such an extensively refined collaboration diagram for a part of the interactions needed for processing ATM F5 AIS cells. The diagram has been drawn assuming adequate data transport capacity (at least three interconnection buses) and processing capacity (at least two of each required functional unit type). The dotted arrows indicate information generated by the interconnection network controller (e.g. immediate integer generation and conditional execution control), and the solid arrows indicate transfers between functional units. Two arrows pointing to a single FU indicate that both of the input registers of the FU are loaded with data (see Chapter 3 for more information concerning architectural details). Dashed horizontal lines indicate timing (i.e. cycles).

A collaboration diagram refined this far can be modified to match a particular TACO architecture (i.e. by specifying the number of functional units and interconnection buses in an architecture). From a diagram refined this far it is already possible to draw preliminary bus utilization and clock cycle count estimates for the target application running on a specified TACO architecture. For example, the interactions shown in Figure 4.2 require 8 clock cycles and use 21 out of 24 possible data transfers in a TACO processor with three buses and two FUs of each required kind. More typically however, the refinement of collaboration diagrams in the TACO framework aims at producing a time-dependent sequential mapping between the required operation types and the required data transports of the target application. The

[1]AIS = Alarm Indication Signal [60]. F5 AIS processing is an ATM OAM function.

resulting collaboration diagram is actually a block diagram of executing the target application on the simplest possible TACO architecture, and it serves as a specification for the sequential assembler code for a virtual processor (to which we will return shortly).

Further research on application analysis is nowadays conducted in another line of TACO research. Currently a key topic in this direction is finding ways of automatizing the application analysis process. See e.g. [4, 72, 113] for more discussion on this research direction.

Figure 4.3 shows a block diagram of the TACO design flow discussed in this chapter. The application analysis techniques discussed above reside in the first two steps of the flow, i.e. boxes *Analysis of protocol processing application specification* and *Information on required functionality / modules* in Figure 4.3). Once the required operations to perform the application have been determined, they are compared to operations provided by the FUs already existing in the TACO module library (i.e. SystemC, Matlab and VHDL component library).

If the required operations are found in the library or they can conveniently be performed by using the existing operations in the library, the library will remain unchanged. However, if the application requires an operation that is not directly mappable to the modules available in the TACO library, the operation is added to the library. This is done by creating SystemC and VHDL modules of the operation as seen in the dashed box at the top right of Figure 4.3. Once the VHDL module has been synthesized, the physical parameters of the module are inserted into the Matlab estimation model. These characteristics can be e.g. the number of logic gates and the ratio of combinatorial vs. non-combinatorial logic in the module that can be used to make the estimations more precise. If the module is not synthesized at this time, the Matlab model will calculate estimates for the physical characteristics of the module based on default settings.

Table 4.1 lists the protocol processing functional unit versions currently available in the TACO module library (the functional unit types are discussed in Chapter 3). As can be seen in this table, some units in the Matlab and VHDL models currently exist only as a 0.35 μm implementation, some as a 0.18 μm implementation, and some as both. The reason for this is that the older 0.35 μm implementations were made using a single manufacturer's technology libraries, whereas the 0.18 μm VHDL implementations are implemented using standard CMOS technology. It should be possible with little effort to adapt the 0.18 μm implementations for use also in newer technology generations such as 0.13 μm. For the Matlab implementations, technology dependent input parameters need to be updated to reach an estimation model in another technology generation. This is done either by using information from module synthesis, or by using designer-defined generic technology pa-

FU		Matlab		VHDL	
type	SystemC	0.35 μm	0.18 μm	0.35 μm	0.18 μm
Matcher	X	X	X	X	X
Shifter	X	X	X	X	X
Comparator	X	X	X	X	X
Counter	X	X	X	X	X
Masker	X		X		X
HEC FU	X	X		X	
ICMPv6 FU	X		X		X
Local Info FU	X		X		X
Routing Table FU	X		X		X
IP Checksum FU	X		X		X

Table 4.1: Available versions of TACO protocol processing FUs in the different processor models. "X" indicates that the FU type and version in question exists in the TACO component library. The operations performed by the FUs listed here are discussed in Chapter 3.

rameters. The SystemC modules are technology-independent. In addition to having module versions for different technology generations, it is naturally also possible to have different optimization versions of modules. For example, one version could be optimized for maximal processing speed, while another one is optimized for low power consumption. Still, both modules would perform the same operation for the same input data.

Upon its completion, the TACO application analysis process should provide the following results:

- A list of hardware operations needed for the target application.
- Existence of suitable TACO hardware modules for the listed operations.
- Allowable processing time per PDU. This value is based on the networking environment of the target application, and depends on the network speed and the datagram size (e.g. 100 Mbps Ethernet, 1500-octet frames: the maximum allowable processing time per PDU is 120 μs).
- A list of physical constraints for hardware design (dependent on the target hardware implementation, e.g. stand-alone vs. NoC): maximum tolerable values for power consumption and area use, and the technology generation (e.g. 0.18 μm) to be used.

4.1.2 Virtual Assembler Code

After establishing that all the operations in the application can be performed by modules available in the TACO libraries, the application specification is refined (i.e. the amount of details in the specification is gradually increased, thus causing the abstraction level of the specification to be gradually lowered) until it becomes a list of consecutive data moves between elementary TACO protocol processing operations (i.e. operations offered by the modules in the TACO libraries). A data move of this kind could be e.g. to move a counter's output value to the input of a greater-than operation. This list of data moves is called the *sequential assembler code for a virtual processor* in the TACO flow (see Figure 4.3 for a box with this label). The term "virtual processor" means a TACO processor with one functional unit of each needed type and one bus in the interconnection network. The application specification is refined into virtual processor assembler code by e.g. resorting to the use of collaboration diagrams as described in the previous section.

4.1.3 Iterative Design Space Exploration

Once the first virtual assembler code has been derived for the target application, the TACO design flow enters the iterative design space exploration cycle (From box *Sequential assembler for virtual processor* to box *Analysis of design quality*).

As defined in the introduction of this thesis, design space exploration is the task of evaluating the quality of several designs in terms of hardware architecture, application software, or a combination of both. In the TACO framework, this analysis is performed based on quality metrics specified by the target application; the metrics are typically a combination of acceptable ranges of values for power consumption, chip size and allowable execution time. We have already seen that the TACO hardware design space is threefold; design decisions are needed for the types of hardware blocks to be used, the number of such blocks, and the connections between them. Within this design space, only a set of hardware architectures is able to perform the target application correctly. Only a subset of these architectures is able to function within the timing constraints set by the protocol processing application. Again, only a subset of the architectures fulfilling the time-wise requirements is feasible in terms of power consumption and circuit size.

In the TACO flow, design space exploration is not completely automated: the designer relies on his experience from previous TACO projects in specifying typically five to ten architecture configurations for detailed evaluation. Each configuration is evaluated (simulated and estimated at the system-level) before the next one is specified. With this kind of an iterative approach to design space exploration, results and experiences from each configuration can

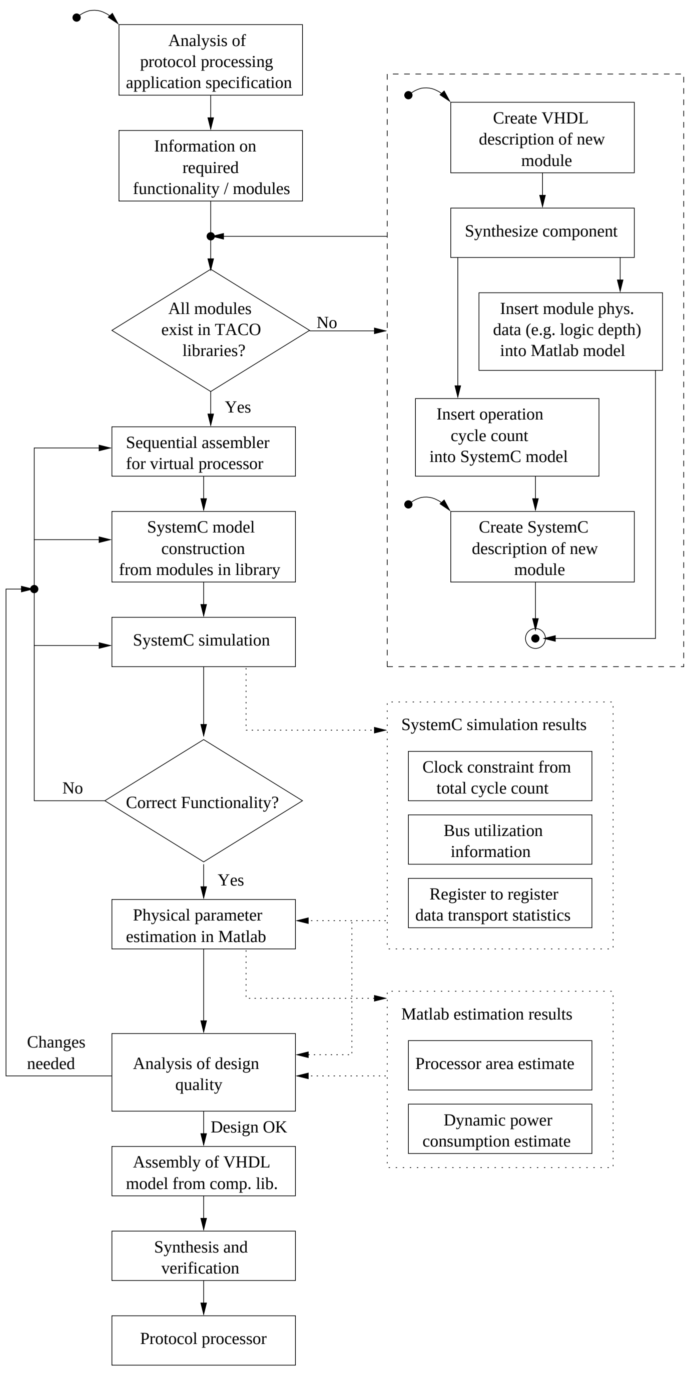

Analysis of protocol processing application specification
Information on required functionality / modules
All modules exist in TACO libraries?
No
Yes
Sequential assembler for virtual processor
SystemC model construction from modules in library
SystemC simulation
Correct Functionality?
No
Yes
Physical parameter estimation in Matlab
Changes needed
Analysis of design quality
Design OK
Assembly of VHDL model from comp. lib.
Synthesis and verification
Protocol processor
Create VHDL description of new module
Synthesize component
Insert module phys. data (e.g. logic depth) into Matlab model
Insert operation cycle count into SystemC model
Create SystemC description of new module
SystemC simulation results
Clock constraint from total cycle count
Bus utilization information
Register to register data transport statistics
Matlab estimation results
Processor area estimate
Dynamic power consumption estimate

Figure 4.3: TACO protocol processor design flow.

be taken into account when specifying the next one. The evaluation of an architecture may reveal that changes are needed even at the level of the original virtual assembler code; for example, simulations can suggest incorrect execution of the target application. However, most of the time the designer would detect either a need for increased performance (with the potential cost of increased area and power needs), or a need for reduced area or power (with the potential cost of reduced performance).

The *virtual assembler code* obtained through application analysis and specification refinement defines the types of functional units needed to perform the target application and is thus used as an initial basis for deriving different architecture instances. This is the *SystemC model construction* box in Figure 4.3. The SystemC simulation model of an architecture candidate is usually constructed by generating a top level file using the TACO design tool, but simulators can also be constructed manually. An experienced designer can derive good candidates for processor architectures quickly, since such a designer is able to predict the kinds of effects that adding or removing buses or functional units to/from an architecture will have on overall system performance. We will return to the construction and structure of the SystemC simulation model in section 4.2, and to the TACO design tool in section 4.5.

To exploit the increased parallelism offered by concurrent data transports in TACO processors the application code (initially the virtual assembler code) must be organized in an optimal manner for each given architecture instance. This optimization will later be carried out by a compiler that takes in program code written in a high-level language and produces optimized assembler code; however, with the current tools available in the TACO framework, the virtual assembler code needs to be optimized manually for each architecture instance.

After an architecture instance has been constructed and the application code has been prepared for the specific architecture instance, the system can be simulated (box *SystemC simulation* in Figure 4.3). If the SystemC simulation reveals incorrect or otherwise unwanted functionality in the processing of the application, the program code and/or the hardware architecture need to be modified. It is up to the designer's experience to decide at this point whether to modify only the application code, only the hardware architecture, or both. If the designer is satisfied with the overall simulated system functionality, physical characteristics (i.e. area use, power consumption and worst-case delay) of the simulated hardware architecture are estimated. To do this, the Matlab estimation model needs to be configured for the target architecture (e.g. using the TACO design tool). We will return to details of the Matlab estimation model in section 4.3. Depending on the designer's choices, the Matlab model may take some of the SystemC simulation results as parameters (boxes *Physical parameter estimation in Matlab* and *SystemC simulation results* in Figure 4.3). The designer's choices include for example power optimization by using register transfer statistics for preliminary processor floorplanning, and bus utilization information for a more precise estimate on the overall power consumed by bus drivers during data transfers.

As the last phase of the iterative exploration cycle of the TACO design flow, the Matlab and SystemC results are combined and analyzed to determine whether the system (i.e. the hardware architecture and the optimized application code for the architecture) meets the requirements of the original specification (box *Analysis of design quality* in Figure 4.3). The number of clock cycles it takes for the given application to be performed on the simulated architecture is obtained from SystemC simulations, and an estimate of the worst case delay (clock cycle) using a given manufacturing technology is obtained from the Matlab estimations. By multiplying the number of required clock cycles with the estimated cycle time, information on the architecture's capability of fulfilling the timing requirements of the target application is obtained. Similarly, the area use and power consumption estimates are compared to constraints specified for the target application. If the design was unable to perform the given application correctly, unable to match the design constraints (timing, power, area), or still a better solution is desired, the designer returns to earlier stages in the flow as indicated in Figure 4.3.

The goal of the iterative design space exploration in the TACO flow is to find at least one protocol processor architecture that correctly performs the target application and fulfills all given design constraints. If more than one such configuration is found, the one with best results in the most critical constraint(s) is chosen for further evaluation.

Once a satisfactory architecture has been found, the TACO flow continues with generating a synthesizable VHDL model (box *Assembly of VHDL model* in Figure 4.3). Generating the synthesis model requires a configuration file to be created. The configuration file specifies all components of the processor and their interconnections. The file is substantially more complex than the corresponding SystemC file due to VHDL's lack of support for object oriented programming conventions. Thus, creating the configuration file manually would result in increased design time and would require a thorough proofreading. For this reason, it is recommended that the designer generates the configuration file using the TACO design tool: this way all potential programming mistakes are avoided and the designer can be sure that the synthesis configuration matches the simulated configuration. Whether manually or automatically created, the VHDL configuration file is used to synthesize the processor architecture (box *Synthesis and verification* in Figure 4.3).

Prior to synthesis, the architecture and the program code are verified by VHDL simulation. If all requirements are fulfilled the design is ready to be synthesized as described in section 4.4 later in this chapter. Following synthesis, the design is put through the processes of placement and routing.

In these steps detailed information of the physical placement of the functional units and interconnections between and within them are derived. Successful completion and verification of the routing finishes the design process. The tasks and processes following VHDL simulation and synthesis are tool and manufacturing technology dependent. The details on work done and processes undergone in the stages following gate-level synthesis are beyond the scope of this thesis.

4.1.4 Turn-around Time in the TACO Design Flow

Since certain parts of the SystemC simulation model are implemented in high-level C++ (as will be seen in section 4.2), TACO simulators execute very fast. One run of a packet processing loop can be performed in less than a second or at most a few seconds on a standard PC. The simulation speed in terms of clock cycles per second depends on the complexity of the architecture being simulated. As an example, SystemC simulations of the IPv6 client processor core (which is discussed in Chapter 5) execute at the speed of 2 950 clock cycles per second in a PC equipped with a 1 000 MHz processor, 256 MB of main memory, an IDE hard drive and a Linux installation with kernel version 2.4.3. Since in the IPv6 Client case study 1 920 clock cycles were needed to process one IPv6 datagram, the simulation speed in that particular case study is about 1.5 IPv6 datagrams per second. Obviously the total simulation time depends also on the number of PDUs processed in the simulation; in the IPv6 client processor simulator the simulated processing of 1000 IPv6 datagrams takes 10-11 minutes. See Chapter 5 for more information and details about the IPv6 Client case study.

Also the Matlab model executes fast, a typical execution takes 3-4 seconds on a standard PC. Thus, one run of the design iteration cycle (from box *SystemC model construction* to box *Analysis of design quality* in Figure 4.3) can also be carried out in little time. In addition, by using the TACO design tool no time is lost in writing and debugging the top-level processor configuration files for the three models. Therefore the iterative design space exploration part of a TACO design project can be expected to complete in a short time frame (a day or two) while providing good results.

The most time consuming parts in the design process are the steps from logic synthesis onwards. Depending on the optimization constraints, already the synthesis of an architecture instance can be an overnight process on a typical workstation (we use a SUN Ultra 10 workstation). The process of placement and routing consumes even more time.

4.2 The Processor Simulation Model

This section focuses on the SystemC [71, 85] simulation model for the TACO hardware platform. The TACO SystemC model is implemented with object oriented programming techniques and a component library based approach. Key design goals for the simulation model have been on the other hand easy model instantiation and on the other hand high execution speed in simulations using affordable computers. We start the discussion with an introduction to executable specifications and SystemC. Then the design and implementation of the TACO SystemC model and its usage conventions are discussed. The discussion on the simulation model is concluded with a description of the problems that were encountered and needed to be solved to use object oriented C++[105] conventions for describing hardware in SystemC, and with a look at the suitability of object oriented programming techniques for hardware design.

An interesting and fairly recent development in the EDA (Electronic Design Automation) community is the adoption of executable specifications as a potential replacement for traditional written specifications. The idea behind executable specifications is that instead of reading through a large quantity of documents describing the desired functionality of a system, the system designer could simply run the executable specification and see how the system is supposed to work. Such an executable specification is gradually refined to contain more and more implementational details during system development. Thus, ideally an executable specification serves also as documentation throughout the design project. Executable specifications are expected to provide unambiguity, completeness and correctness to system specification [37].

Currently C and C++ are the most popularly chosen bases for implementing executable specification development languages and environments as seen in e.g. [36, 37, 92]. The choice to use these two languages is obvious in terms of availability and cost: existing tools and programming skills can be used, since companies already use C++ in their software development and C in their embedded system programming. Also, operating systems like Linux [112] provide C and C++ compilers and utilities free of charge. However, these languages are designed for writing computer programs, not for describing computers or other hardware devices. Therefore they lack necessary functionality and features for describing clocks, signals, reactivity and parallel processing. Also, most current design flows that start from a C or C++ based functional level description contain a “jump phase” in which the functional model written in C/C++ is translated into an HDL (hardware description language) [35, 71]. This phase is often manual and involves refining the model down to the RTL level. To solve this problem, either a system description language should be built on top of C or C++, or a language designed specifically for system description should be created.

SystemC explores the first option. It is a somewhat recently introduced C++ class library for designing executable specifications and cycle-accurate simulators of hardware in C++. SystemC is distributed under an open license and is supported by several of the major EDA companies. It provides support for hardware-oriented data types like modules, ports and signals. Originally there were two major goals in designing SystemC [71]: to provide a single language framework for co-verifying systems at varying, possibly mixed, abstraction levels, and to allow system designers to gradually refine their models towards the RTL level without translating them into a HDL.

4.2.1 TACO Configurable SystemC Simulator

When SystemC was introduced in 1999, it seemed to have a lot of potential in system-level modeling. Thus, with some enthusiasm, the decision to experiment with SystemC in the TACO project was made. SystemC development seemed to be on the right track - it was firmly believed in the TACO project that it would be beneficial to combine the advantages of object oriented (OO) programming techniques available in C++ with the support for cycle accurate simulations and hardware data types provided by SystemC. Surprisingly though, initially SystemC actively discouraged the use of C++'s object oriented techniques. The situation has improved in later versions, but even the current SystemC version still enforces certain limitations for object oriented programming. Also, for a long time it seemed that the SystemC community widely uses SystemC for constructing low level descriptions of hardware devices in a similar way as hardware is described in traditional HDLs. It has happened only quite recently that discussions in EDA conferences and online discussion groups have started covering the use of object oriented techniques in context with system design using SystemC.

To be able to rapidly simulate, evaluate and explore architectures for a given protocol processing application or algorithm, an object oriented configurable protocol processor simulation model was devised for the TACO framework. The component library based model is written in SystemC and maintained in a standard x86 PC running Linux. The model contains implementations of functional units, sockets, interconnection buses, and the interconnection network controller. Using the simulation model it is possible to construct a cycle accurate simulator of any given TACO architecture, and to simulate both the functionality of the hardware as well as the software. The application software code is input to simulations as hexadecimal values (i.e. compiled TACO instruction words).

The level of abstraction used in implementing the TACO simulation model is heterogenous in the following sense: the inter-module communication is handled at the RTL level, whereas the internal functionality of the modules is implemented as higher level C++ using SystemC's fixed bitstring length data types (e.g. `sc_uint<32>`, 32-bit unsigned integer). The motivation for using heterogenous abstraction in the simulation model implementation is that simulators are notably faster when the execution logic is implemented as normal C++ instead of RTL-style coding; high-level C++ can be used inside the modules as long as the correct amount of cycle delays (i.e. SystemC `wait` statements) is inserted into modules requiring more than one clock cycle to complete their execution.

Since the functional units of the TACO hardware platform have a very similar interface, inheritance is utilized in the SystemC simulator (see Figure 4.4). This is done by gathering the behavior and connectivity that is the same for all functional units into a parent class, and placing only the additions to the port/signal configuration required by individual modules as well as the code for the particular FU's execution logic to the child classes. The approach has obvious benefits: the code is more compact and readable (the interface code is not repeated multiple times), there are less errors to debug (only additions to the interface are coded) and adding new functional units to the SystemC component library is faster (most FUs in the hardware platform differ only in the internal implementation, not so much in the physical interface). New FUs are always verified for correctness at the time they are made available in the TACO component library. Thus, all FU descriptions in the library are known to have already been verified.

The execution of a TACO simulator for a given architecture consists of two phases. In the *setup* phase all modules are instantiated. This phase relies heavily on polymorphism[2] to allow automatic socket instantiation and addressing, and to connect different kinds of functional units to buses through sockets. After the *setup* phase all modules in the processor have been instantiated. No more modifications to the architecture are made, and polymorphism is no longer used. The second phase, *simulation*, is started when the command `sc_start()` is issued in the `sc_main()` routine (SystemC's equivalent to the `main()` routine found in all C++ programs). It is important to realize that at the moment the simulation starts, the processor architecture is completely static and no more modules are dynamically constructed. For this reason, it is possible (although it would require some effort) to mechanically remove the polymorphic code used for automatic simulator construction. Also the inheritance can be mechanically removed, thus making it possible to have static module instantiation and stand-alone modules without a class hierarchy. Such removal of polymorphism and inheritance would be useful, were it necessary to develop the code to RTL level for direct synthesis from SystemC;

[2]Programming convention in which routines can be applied to objects of many different types.

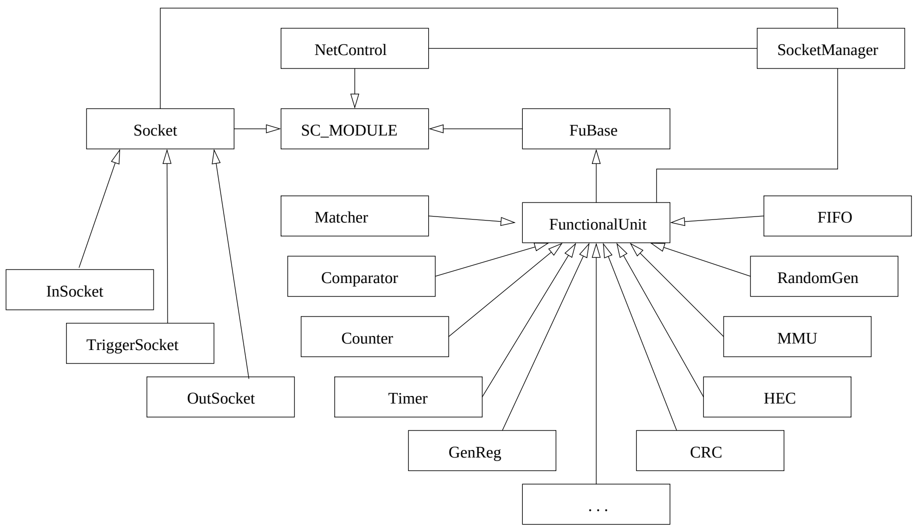


Figure 4.4: SystemC simulator class hierarchy. Arrows indicate inheritance (arrow head points to parent class), and lines indicate association.

certain commercial tools nowadays support synthesis of system descriptions written in a predefined subset of SystemC. However, in the TACO framework this is at least currently not necessary, since the TACO design tool (discussed later in this chapter) is able to set up a VHDL synthesis model for any given TACO architecture.

The classes that are used for simulating hardware are derived from the class `sc_module` provided by SystemC. This class provides among other things macros for simulating signals and ports. The TACO `SocketManager` class is not a hardware simulation module: it is used by objects from the functional unit classes during the *setup* phase for generating, connecting and maintaining sockets, socket ID's and signals dynamically. During the *simulation* phase, `SocketManager` is used for obtaining pointers to any modules that inherit from `sc_module` and were dynamically created in the *setup* phase of simulator execution. See the code example on the next page for an example of using the `SocketManager` for reserving a socket ID.

We recall that there are three different types of sockets in a TACO processor (see Chapter 3 for more information on sockets). Input sockets are used for writing data into operand registers in FUs, output sockets for reading data from result registers, and trigger sockets for writing data into trigger registers and simultaneously triggering FU operations. In the simulator all sockets are derived from the base class `Socket` that provides most of the socket interfacing and a state machine for each socket. The subclasses add their own internal functionality and interface requirements to the base class description.

The three level hierarchy for functional units was needed to overcome certain SystemC limitations. We will return to these limitations shortly. The base classes `FuBase` and `FunctionalUnit` provide the interfacing and a state machine needed by each FU. Additions to the base FU interface (e.g. an additional register) and the actual processing task to be executed by a specific functional unit are placed into the FU leaf classes (Matcher, Comparator, Counter etc. in Figure 4.4). A functional unit is added to the TACO SystemC simulation model component library by specifying a new leaf class under the `FunctionalUnit` unit class, specifying additional FU registers (if any), specifying the identifiers for logical trigger socket IDs, specifying whether the FU needs a result bit signal, and writing the code for the operation to be performed. However, due to limitations of SystemC, some additional code is also needed. For example, the system clock needs to be tied to the new FU at this level of the class hierarchy.

A simple adder FU with one operand, one trigger and one result register and no result bit would be added to the library in the following manner (for clarity, bypass code for the mentioned SystemC issues is not included):

```
class Adder:  public FunctionalUnit {
  void assignTriggerIds() {
     trigger->setId(SocketManager::reserveInSocketId("TADD"));
  };
  void triggerOperation() {
     resultReg = operandReg + triggerReg;
  };
}:
```

In the above code example, a new leaf class `Adder` is specified under the parent class `FunctionalUnit`. Then, a logical trigger ID called `TADD` is allocated to the new functional unit. Note that the sockets are defined in the parent class; thus, in the leaf class only the logical trigger IDs need to be defined. Next, the operation to be performed is defined. For FUs that are able to perform more than one operation on the data, additional logical trigger IDs need to be allocated. This is done by repeating the allocation line of the code example with different ID names.

The code in the function `triggerOperation()` defines that once the FU is triggered (i.e. when data is written into the trigger register), the values stored in the operand and trigger registers are added together, and the result of this addition is placed in the result register. Timing issues for register reads and writes are managed in the parent class; in the leaf class only the operation to be performed in the *execute* pipe stage needs to be defined.

The interconnection network controller (class `NetControl`) does not have any subclasses since there is always only one such module in a TACO processor. The controller is responsible for extracting bus instructions from TACO instruction words, generating immediate values and evaluating guard expressions for conditional execution (see Chapter 3 for more details on the interconnection network controller). The interconnection network controller implementation in the simulation model is state machine based.

Instantiating the SystemC model for a given architecture

The simulation model is set up for simulating a given TACO architecture by instantiating as many interconnection network buses as necessary and then instantiating as many functional units as necessary (and specifying their types). This is done either manually or using a tool like the TACO design tool discussed later in this chapter. The functional units are connected to the interconnection network by calling connect routines in the newly created bus objects. The creation of sockets and the signals required for connecting the sockets to buses and functional units is done automatically and dynamically by specifying which FU registers connect to which buses as seen in the code example below (line numbers have been added for commenting purposes).

```
0:  NetControl* nc = new NetControl("NC");
1:  Bus* bus1 = new Bus("Bus1");
2:  Bus* bus2 = new Bus("Bus2");
3:  Matcher* m1 = new Matcher("M1",clk);
4:  bus1->insertOperand(m1);
5:  bus1->insertData(m1);
6:  bus2->insertData(m1);
```

Line 0: *Create the interconnection network controller* nc.
Lines 1 and 2: *Create two buses and connect them to* nc.
Line 3: *Create a matcher functional unit* m1, *connect the system clock to it.*
Line 4: *Create an input socket for the operand register of* m1, *create signals for connecting the socket to* m1, *connect the socket to* m1 *and* bus1.
Line 5: *Create an input socket for the data register of* m1, *create signals for connecting the socket with* m1, *connect the socket to* m1 *and* bus1.
Line 6: *The data input socket already exists; just connect the socket to* bus2.

As can be seen in the code example, much of the complexity in specifying the interconnections between different modules has been abstracted away from the designer by resorting to object oriented programming techniques. Still, using the TACO design tool for simulator instantiation should be preferred: the tool generates the instantiation file completely and without errors every time, which is more demanding to achieve with manual instantiation. Once the architecture has been specified in the described manner in the top level TACO SystemC file (`main.cpp`), the simulator is compiled. This is done

by issuing the command `make` in the directory in which the SystemC model resides. Normally the compilation takes less than a minute. The executable produced by the compilation is the simulator for the specified architecture. The architecture can then be simulated by starting the executable.
The SystemC simulations of a TACO processor architecture provide the following results for design quality evaluation:

- **Functional verification.** The key result of any simulation is of course verification of correct system functionality. Such is also the case in TACO: the most important goal in a processor simulation is to find out whether the simulated architecture functions correctly when executing the target protocol processing application.
- **Clock cycle count.** TACO simulators count the number of clock cycles used in each simulation run. This information along with the network speed requirement of the target application (e.g. 100 Mbps Ethernet, 622 Mbps ATM etc.) and the estimated achievable clock speed (from physical estimations; discussed shortly) determines whether the architecture being simulated is able to execute the application fast enough.
- **Bus utilization.** During each simulation, TACO simulators calculate the number of possible data transfers and the number of actual data transfers for each bus in the interconnection network. Thus, when a simulation ends, the simulator is able to report relative bus utilization values for each bus (e.g. 150 data transfers actually made out of 200 possible ones: relative bus utilization = 75 %).
- **Register transfer statistics.** During simulations TACO simulators also record the source and destination addresses for each data move into a database. At the end of each simulation, the values in the database are used for calculating the frequency of data moves between individual registers. This list of data move frequencies is then sorted into descending order and reported as register transfer statistics to the designer.

The results are written into an XML file that can be used e.g. for passing the simulation results on to a result analysis tool like the TACO design tool discussed later in this chapter.

SystemC issues in TACO simulator implementation

There are certain issues about the design of SystemC that need to be pointed out. Some are more concerned with what could be called "good C++ practice" while others are more about the architecture and API of SystemC.

There are two rules that are taught in most elementary C++ programming courses: **do not use macros** (section 7.8 of [105]), and **do separate interface from implementation** (item 34 in [79]). In SystemC many constructs are implemented as macros that rearrange text in the program code before the compiler sees it. This causes many difficulties in debugging the code (e.g. debuggers like the Data Display Debugger (DDD) [24] show macros disassembled). For this reason an instruction (a macro) that causes an error or malfunction in the compiled program (e.g. a simulator for a TACO processor) is often difficult to find. Macros have importance in writing C code, but the use of macros in C++ is discouraged in literature.

A more general problem with the SystemC macro approach is that it breaks the distinction between interface definition files (header files, `*.h`) containing only declarations (of variables and methods), and implementation files (code files, `*.cpp`) containing the bodies of the methods. Although this distinction is not enforced by the C++ language, it is considered good software engineering practice for several reasons. First, readability of the code is considerably improved when interface definitions are not intermingled with implementation details. Second, recompilation speed is increased, because changing an implementation only causes the implementation file to be recompiled. Due to this SystemC feature, a small change in the TACO simulator may cause all files to be recompiled. Interestingly enough, the SystemC documentation actually recommends giving the module constructor implementation within the interface definition file.

Since the TACO simulator needed to be easily expandable, we early on decided to use inheritance as a structuring mechanism. However, it was soon observed that this kind of implementation technique seems to be actively discouraged in SystemC version 1.0.1.[3] After much debugging the cause of the problem was finally isolated into the constructors of the TACO functional unit classes: the constructor for an object of class `module` is declared as `SC_CTOR(module)` in SystemC. This is expanded to

```
typedef module SC_CURRENT_USER_MODULE;
        module(sc_module_name);
```

The first line defines a type name alias, that will be returned to shortly. The second line is interesting. It declares a constructor for objects of class `module` that takes an anonymous `sc_module_name` object as a parameter. The life-time of the object is the scope of the constructor and it is used to push the current module onto the simulation context stack. This makes sure that all ports declared in the class are attached to the correct instance of class

[3]In SystemC version 2.0, inheritance was allowed but still limited; For example, ports could not be defined inside a member function.

module in the simulator. So when building the inheritance hierarchy with sc_module as the base class, we must make sure that all the parent classes are instantiated within the scope of the correct sc_module_name object. This can be achieved by adhering to the following convention: each non-leaf class must have a default constructor (a constructor with empty argument list).

However, a second problem still remains. This one concerns the typedef in the first line, which declares the type SC_CURRENT_MODULE to be an alias for the name of the current module. This type is needed in macros that are used to connect the execution of a method to the correct simulation context. However, since non-leaf classes are created with the default constructor, the constructor will not define SC_CURRENT_MODULE and thus the program will fail to compile. It was necessary to resort to C++ templates to overcome this particular problem.

For this reason, a multi-level class hierarchy for the inheritance needed to be implemented: the actual functional unit implementations were implemented as leaf classes, and two intermediate levels before the SystemC level (the sc_module class) were needed to circumvent the problem described above.

The top-level class FuBase (see the code example following this paragraph, and Figure 4.4) is an abstract class[4] that defines the interface to the functional unit classes. The second-level class FunctionalUnit is a template class that contains the common functionality of the functional units. It is also used to set SC_CURRENT_MODULE to the correct type. Conceptually the template class FunctionalUnit and the class FuBase together form an interface and functionality definition shared by all functional units. The following code shows the class declarations of the three-level class hierarchy of the TACO simulation model as seen in Figure 4.4.

```
class FuBase:  public sc_module
  {...};

template <class CB> class FunctionalUnit:
  public FuBase{
     typedef CB SC_CURRENT_USER_MODULE; ...};

class Matcher:  public FunctionalUnit<Matcher>
  {...};
```

By reading through the SystemC source code it can be observed that by using the internal calls in the SystemC implementation a cleaner design of the TACO simulation model could probably have been obtained. Using the internals it would also have been possible to adhere to "good C++ practice".

[4]A class that cannot be instantiated.

However, since the SystemC internals are not really documented and standardized, the decision to use the official SystemC API was made in the TACO SystemC simulation model implementation.

4.2.2 Object Orientation and Hardware Design

In the preceding discussions in this chapter the idea of using object oriented techniques for describing hardware at various, but high, levels of abstraction has been found quite beneficial: OO techniques were used in context with application analysis and system-level processor simulations. On the other hand, the popular system description language SystemC was found to actively discourage the use of OO techniques for the same purpose. Thus, a literature survey on the topic became intriguingly necessary. The survey revealed that the hardware design community has not reached a consensus on the usefulness of object-oriented techniques in hardware design [98]. The most interesting effort in this context is the attempt to add objects and inheritance to VHDL, called OO-VHDL [97]. The research presented in [97] found several difficult object-oriented concepts that do not have a good mapping to hardware. These include inheritance, and method calls. The research thus seems to suggest that object-orientation does not fit to hardware design very well. This could arguably be deemed a misunderstanding. Object-orientation is more than an implementation language. It is a conceptual way of thinking and structuring the problem domain. In an object oriented design method objects can be viewed at three levels:

1. *Conceptual level*: The classes represent concepts in the domain of study. This view is taken very early in the analysis phase.
2. *Specification level*: The classes specify interfaces of the system. A type is the interface of the class. A type can have many classes that implement it, and a class can implement many types.
3. *Implementation level*: Classes represent code in a programming language.

Both OO-VHDL and SystemC view objects as implementation level entities that should be directly mappable to hardware. If one takes this view it is clear that especially inheritance is non-trivial to map into hardware. Inheritance is an implementation technique that is used to implement subtyping in many object-oriented languages. Subtyping on the other hand allows polymorphism, where a function may be called with several different argument types. The reason that polymorphism is difficult to implement in hardware is simply that in hardware all types (objects) have to be static, since they are physical objects. However, it can be claimed that in any practical hardware design these kinds of situations do not arise, or the corresponding code can be rewritten in a way that circumvents the problem. This holds, because in an instance of a hardware system the set of objects is fixed. It does not change and thus the types of the variables in the program do not change while the system is running. Indeed this is exactly the case with the TACO SystemC simulation model.

Polymorphism and inheritance are very powerful concepts that have allowed programmers to increase their productivity substantially mainly because they enable reuse of code. Therefore, since hardware design is more and more becoming a programming activity, the importantance of allowing hardware "programmers" to use these techniques should not be underestimated.

4.3 The Physical Estimation Model

The estimation model for the TACO hardware platform is designed, implemented and developed by PhD student Tero Nurmi (University of Turku). However, to be able to fully describe the potential of the TACO framework it is necessary to give an overview of the system level estimation model in this thesis. We limit the discussion to an overview of the model in relation to the TACO hardware platform as presented in joint publications of Mr. Nurmi and the author of this thesis [81, 117], and emphasize that the scientific aspects concerning physical and mathematical modeling of delay, area and power described in the following are results from Mr. Nurmi's research and not a scientific contribution of the author of this thesis.

The estimation model for the TACO hardware platform runs in Matlab and is implemented as scripts and functions in Matlab's native M-language. The model is maintained in a standard PC running Windows, but it could also be used in Matlab versions for other operating systems. It contains a set of equations for estimating delay, area and power consumption. The TACO estimation model was first introduced in [81]. It has been updated since to support estimation of power consumption and pipe stage delay and also to utilize block-wise Rent's exponents [2] in the estimation of the number of gates. Somewhat similar approaches for modeling system level physical characteristics have been presented in e.g. [9], [14], [31], [38], [106], but the TACO estimation model has been extensively optimized to support the TACO hardware platform.

Delay formation in TACO processors

The cycle time and hence the critical processing path of a TACO architecture instance is identified and defined by the processing delay produced by the

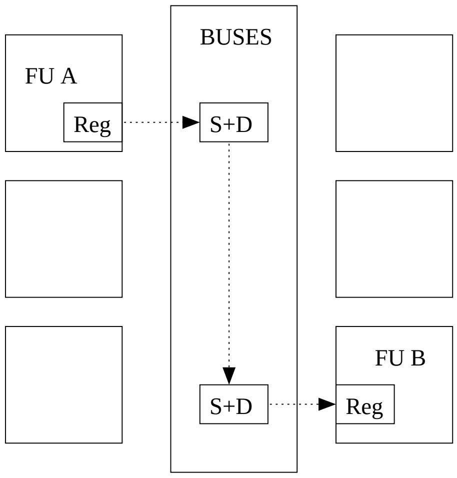


Figure 4.5: *Move* stage delay. The delay consists of socket delays and the bus wire delay including the sending socket's driver. *S* indicates a socket, *D* a driver and *Reg* a register.

slowest pipeline stage. In the estimation model the delay is calculated for a specified technology generation. Here we recall from Chapter 3 that TACO processors have a four-stage pipeline with pipe stages *fetch*, *decode*, *move* and *execute*. The slowest pipeline stage depends on the specified technology generation; as the effect of interconnects relative to logic increases in delay formation, the slowest pipeline stage shifts from the *execute* stage to the *fetch* and *move* stages. Thus, in order to estimate the cycle time of a given TACO architecture, the delays caused by each of the four pipe stages need to be calculated. In the following paragraphs we outline the mathematics used in calculating the delays for each pipe stage.

Move stage delay It is assumed that when socket IDs are sent to the corresponding input and output sockets the output driver of the output socket drives the signal on the bus. The signal is received in the receiving FU's input register, either in an operand or a trigger register. Thus, the *move* stage delay is defined as a combined driver/bus delay and the delay formed in two sockets (input and output).

The principle used in calculating the move stage delay is shown in Figure 4.5: the *move* stage delay is proportional to the distance between two FUs (and especially between the two registers). In the TACO architecture the distance can be approximated with the equation

$$FU_{distance} = \lceil(\frac{N}{2})\rceil\sqrt{area_{FU}} + total\_width_{buses} \tag{4.1}$$

where N is the total number of FUs around the buses, $area_{FU}$ is the area of the largest FU and $total\_width_{buses}$ is the total width of the bus structure. For $total\_width_{buses}$ it is assumed that the silicon area required by the sockets and drivers/repeaters fits under the metal bus structure.

The bus delay portion of the *move* stage delay is then defined as RLC wire delay [65] and takes also into account the possible need for repeaters. The maximum length for global wires without the need to use repeaters is defined in [121] as

$$L_c = 2th[(1+\alpha)\rho\sqrt{\varepsilon_k}\varepsilon_0 c K_c]^{-1} \tag{4.2}$$

where t and h are thicknesses of the conductor and the dielectric, α is a return/signal path resistance ratio (we have $\alpha = 1$), ρ is metal resistivity and K_c is a fringing factor for wire capacitance.

Execute stage delay The *execute* stage delay equals an average gate delay multiplied by the worst case logic depth of a signal path in an FU (worst case "chain" length of serially connected logic gates through which signals need to pass before a result is reached). Using the Matlab model we have been able to conclude that the *execute* stage delay sets the cycle time (and thus the critical path) for TACO processors both in the 0.35 μm and the 0.18 μm technology generation. This is due to the relatively large logic depths in the TACO FUs (see Figure 3.6 in Chapter 3 for the internal structure and organization of TACO FUs). However, in newer technology generations there is a shift towards wire dominance in delay formation. Therefore also the *move* and *fetch* stage delays have to be taken into account in the worst case delay estimation.

Fetch stage delay The *fetch* stage delay depends on the utilized program memory technology and memory hierarchy organization. There are some memory models that predict access time, area and power consumption of on-chip memories but usually it is better to use values provided by a technology supplier.

Area estimation

In area estimations the Matlab model adds up the total area of the functional units (i.e. combinatorial logic) and the area for the bus structure (including drivers and sockets). The timing constraint used for area estimations is $1.20 \cdot f_{\text{clock}}^{\text{min}}$, in which $f_{\text{clock}}^{\text{min}}$ is the minimum clock frequency requirement for running the target algorithm on the target architecture (the value is obtained from SystemC simulations). The 20% increase is inserted to ensure adequate processing performance after placement and routing of a design.

The area for the bus structure depends on the number of buses and the wire widths of the buses. Also the number of FUs has an effect on the size of the bus structure (see Eq. 4.1): the more there are FUs the longer buses are needed and hence the bus structure area increases. The number of gates in an FU is estimated by Rent's rule [69] as

$$N_{I/O} = K_p N_g^p \tag{4.3}$$

where $N_{I/O}$ is the number of signal I/O connections in an FU, N_g is the number of gates in an FU and K_p and p are Rent's constant and exponent, respectively.

Rent's constant and Rent's rule can be evaluated separately for each FU. This has been examined in [2] for blocks of a RISC-type processor.

Knowing the estimated number of gates and by using area information of a single, *average* gate (from a standard cell library data sheet) the total area needed by functional units can be estimated. However, space has to be reserved also for global power and clock delivery networks when estimating the total area of a processor. We estimate this by adding 20 % to the the resulting logic area.

Additionally, the area of the bus structure has to be estimated. Once the number and width of buses is known and a preliminary FU floorplan exists, we can estimate the area of the bus structure. However, depending on the size of the socket drivers and possible repeaters it may happen that buses of the minimum distance between consecutive wires can't be used.

Power estimation

The dynamic power consumption is defined by the capacitance of the logic gates and wiring, power supply voltage, cycle time and switching activity of electrical nodes in different FUs and bus drivers. The wiring includes inter-gate wiring inside an FU as well as the global bus structure including driver/repeater capacitance and wire capacitance.

The designer can affect dynamic power consumption mostly by preferring designs that operate on lower clock frequencies or by using lower supply voltage. In the former case the driving ability requirements for gates are less stringent and therefore gates can be smaller. Minimum gate size is defined by the technology used. However, the timing requirements of the application may not allow the use of minimum size gates. In this case a lower supply voltage has to be used. This may deteriorate performance because the supply voltage should be large enough compared to the threshold voltage (V_{th}) of transistors. New technologies make lower threshold voltages and thus also lower supply voltages possible, but with the cost of increased leakage in transistors. As for the buses, the designer can place the blocks that have much bidirectional traffic close to each other.

Instantiating the Matlab model for a given architecture

The Matlab estimation model is set up for a given TACO architecture using a plain-text configuration file. The file specifies the number of buses in the architecture, bus width, number and type of functional units, and certain technology generation dependent parameters. As an example, the following code instantiates two 32-bit data buses (`N_dbus` and `W_dbus`), two sets of address buses[5] (`N_abus` and `W_abus`),and two functional units. The functional units are input as a three-column matrix. The first column indicates the number of functional units of the selected type in the architecture, the second column indicates the number of registers, and the last one the total bit count of the registers (e.g. four 32-bit registers = 128 bits). After the FU matrix, labels (FU names) are given for each row. This is done mainly for better readability of both the instantiation file and the Matlab result file (which uses the same labels). The FU definitions end by specifying the number of FUs that have a result bit connected to the network controller (the `N_ctrl` parameter).

```
N_dbus = 2;
N_abus = 2;

W_dbus = 32;
W_abus = 16;

FU_spec = [1 4 128;
FU_spec = 1 3 96];
FU_spec(1,:)  = Matcher;
FU_spec(2,:)  = Shifter;
N_ctrl = 2;

gate_area = 54e-12;
fld = 22;
```

The technology dependent parameters shown after the FU definitions in the above example are set up similarly as the bus parameters. There are many more technology dependent parameters in addition to the three shown in the above example. However, analyzing all of them is beyond the scope of this thesis. The ones shown above are examples of some of the parameters the designer can define if no prior values from synthesis are available; `gate_area` gives the size of one single logic gate (needed for area estimations), and `fld` specifies the logic depth of the logically deepest FU in the architecture (used in *execute* pipe stage delay estimation).

[5]Two SRC and two DST buses, each of them eight bits. The Matlab estimation model treats the SRC+DST bus pair as a 16-bit address bus.

The Matlab model configuration file can be written by the designer for the architecture in question, or generated by the TACO design tool (recommended). The TACO design tool is discussed later in this chapter. Once created, the configuration file is fed to Matlab, and the estimation results are given in a similarly formatted plain-text file. The Matlab estimations of a TACO processor architecture provide the following results for design quality evaluation:

- **Area estimates.** The Matlab model produces estimates of a TACO processor's logic area as well as the entire processor's area. The logic area covers the area needed by logic blocks such as functional units, sockets and the interconnection network controller. In addition to these, the entire processor area includes also the area needed by e.g. buses, wiring, and power and clock distribution.

- **Delay estimate.** The worst case delay of the processor (and thus on the other hand the maximum obtainable clock speed and on the other hand the critical processing path) is estimated by analyzing delays in the different pipe stages as described earlier in this section. The *execute* stage delay is calculated for a specified (maximum) logic depth, i.e. for the slowest FU. The logic depth can again be a generalization or based on synthesis. The delay estimation in combination with the clock cycle count from SystemC simulations determines whether the architecture in question is capable of providing the performance required by the target application.

- **Power estimate.** The Matlab model estimates the power/energy consumption for each functional unit in an architecture as described earlier. This information in combination with the register transfer statistics from SystemC simulations give an estimate of task energy consumption, i.e. how much energy is needed to execute the target application on the given architecture.

4.4 The Synthesis Model

The TACO synthesis model is implemented as a library of module descriptions in VHDL. The first version of the synthesis model was written using Alcatel's 0.35 μm technology libraries. The current version is implemented using 0.18 μm standard CMOS technology. The TACO VHDL module library includes descriptions for all hardware blocks of the TACO hardware platform, including functional units, sockets and the interconnection network controller. Due to the lack of support for object oriented programming in VHDL, new functional units are added to the library in a "copy-paste"-fashion: the existing FU description that most closely resembles the structure of the new functional unit acts as a starting point for the VHDL implementation of the new functional unit. Then, the parts of the existing description that are not needed in the description of the new FU are removed, and the parts that are needed to describe the new FU are inserted. In comparison to the SystemC model, adding a new FU to the VHDL module description library clearly requires more time and is a more error-prone process.

To build a VHDL description of a TACO architecture, a top-level file specifying all needed modules and their interconnections needs to be created. The preferred way of doing this is to use the TACO design tool. However, it is also possible (although time-consuming, and risky in terms of coding errors) to construct the top-level file manually. For this reason, we will not cover manual VHDL model instantiation here but will return to the topic in the TACO design tool discussion that follows this section.

The TACO processor design methodology discussed in this chapter is concluded when a gate-level synthesized model of a given TACO processor architecture is reached. Thus, the procedures carried out after synthesis are beyond the scope of this thesis. These post-synthesis procedures are defined on the other hand by the tools used and on the other hand by advances in research in that area. Thus, in this section we outline the procedures to be carried out to reach a gate-level synthesized processor model based on the previously mentioned top-level file.

Prior to synthesis, the processor model defined by the manually written or automatically generated top-level file is simulated in a VHDL simulator. All VHDL descriptions of functional units as well as other modules in the hardware platform have already been individually simulated and synthesized at the time of adding these descriptions into the TACO library. Thus, the individual VHDL module descriptions have already been verified to meet the functional specifications given in the corresponding SystemC module descriptions at the time of writing or generating the top-level file. The top-level VHDL file defines exactly the same architecture as the architecture simulated in SystemC. Once the top-level VHDL file for a processor architecture has been generated, the architecture defined by it is simulated using VHDL simulation tools. In these simulations, the synthesis model is appended with a non-synthesizable part that is used for simulating the different memory blocks needed in the specified architecture. For example, program memory is simulated by reading the application code used in the SystemC simulations from a file. A similar solution is used for simulating the inbound and outbound network buffers. By using identical application code in both the SystemC and the VHDL simulations, identical functionality and execution scheduling is expected: in TACO processors, the application code has been scheduled already at the time of application software implementation. Thus, the synthesis model is verified by comparing the VHDL simulation results to the SystemC simulation results running the same application code and the same network data. Successful completion of the VHDL simulations permits proceeding to synthesis.

When the synthesis tool processes the top-level file, it first builds a so called generic pre-processed model of each module in the TACO library (i.e. translates the VHDL code to native code of the synthesis tool). Then, synthesis constraints like target clock speed are set. After this, the modules defined in the top-level file are selected for synthesis on a chosen (or available) manufacturing technology by the optimization process. Then, the architecture is synthesized. The synthesis process produces a gate-level netlist of the architecture, which is used as input for the remaining design steps like placement and routing.

4.5 Graphical Design and Evaluation Tool

The TACO design tool integrates the three previously discussed processor models into a graphical design and analysis environment for TACO processors. The designer is relieved of the task of manually setting up the top level files needed for simulation, estimation and synthesis of TACO architectures.

As seen in the previous sections, TACO protocol processor models are developed, maintained and used in a heterogenous computing environment. This heterogeneity (simulations in a Linux PC, estimations in a Windows PC, synthesis in a Unix workstation) made it obvious already at early stages of the TACO project that manually setting up simulations, estimations and synthesis is a time consuming, error-prone process often requiring more than one person. For this reason, work was started in the direction of creating a processor design and analysis tool. The first version of the tool was implemented in Delphi 2.0 (using the object Pascal programming language). However, this version was not easily modifiable to support adding new modules into the TACO libraries, and it could only be used in Windows environments. Therefore a decision was made to write a specification for a new design tool version. The new specification required, among other things, at least the following functions, features and capabilities to be coded into the TACO design tool:

- The tool needs to be platform independent. This should be accomplished by resorting to the Java programming language.
- The tool needs to enable graphical design of architectures.
- The tool needs to generate SystemC, Matlab and VHDL top level code.

- The tool needs to be able to import architectures (load/save features).
- Newly created modules need to be made usable in the tool without modifying the tool itself.
- The tool needs to support module feature browsing (e.g. results of the most recent physical characteristics estimation of a functional unit).
- The tool needs to have mechanisms for analyzing simulation and estimation results and for preliminary processor floorplanning.

With these requirements in mind, work was started to implement the tool in Java. The key functionality and characteristics of the tool are described in the following pages.

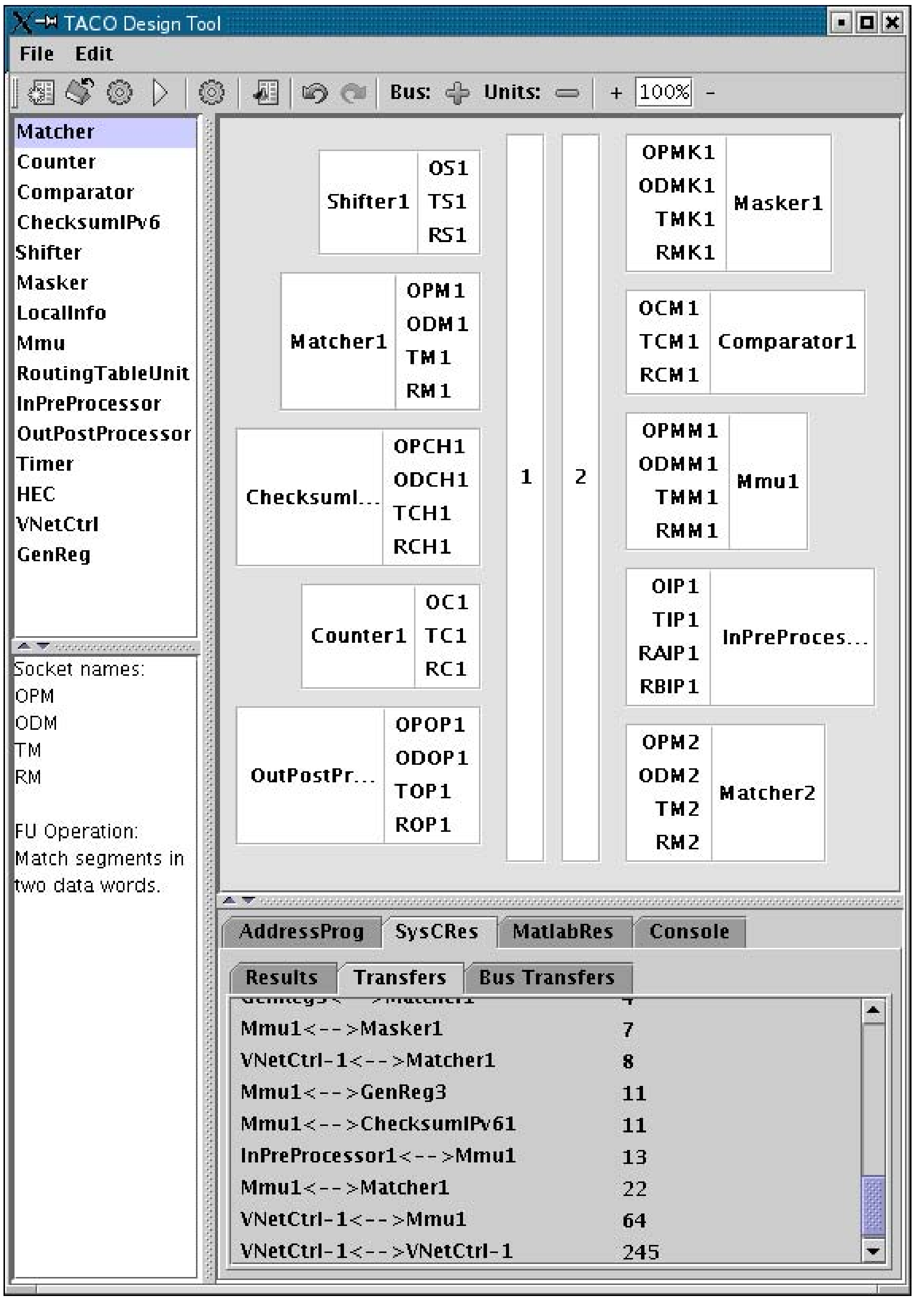
An architecture is designed in the TACO tool by specifying the number of buses and the number and type of functional units (FUs) in the architecture. A functional unit is inserted into the architecture by double-clicking its name on the FU list on the left side of the tool window (see Figure 4.6). The tool responds by placing an FU of the specified type into the architecture and giving it a meaningful and unique name, e.g. Matcher1, Matcher2, Counter1, Matcher3, Counter2.

Based on the type of the inserted FU, the tool automatically places the correct number of input, output and trigger registers into the graphical representation of the architecture as seen in Figure 4.6. The registers are also named automatically. Sockets are not shown in the graphical representation to make the view clearer. However, all necessary sockets are correctly instantiated in the processor models configured with top-level files generated by the tool. The tool also automatically instantiates specific control signals required by some FU types and hides these signals from the designer (e.g. a result bit signal is automatically created and connected from a Matcher unit directly to the interconnection network controller).

Buses are inserted into an architecture by clicking the “Bus: +” button in the tool. The buses are also automatically named. The interconnection network controller is also automatically generated by the tool into the top-level files and is therefore not shown in the tool window. Buses and functional units can naturally also be removed from the architecture.

In addition to placing FUs and buses into the architecture, the designer needs to specify the connectivity between FU registers and buses. The tool supports incomplete connectivity (i.e. an FU register does not have to be connected to every bus as long as it is connected to at least one). However, when an FU is inserted into the architecture, all of its registers are by default connected to each bus already present in the architecture.

The tool supports adding new functional units to the known FU types without modifying the tool itself. This is accomplished by modifying an



Socket names:
OPM
ODM
TM
RM

FU Operation:
Match segments in two data words.

```
Mmu1<-->Masker1               7
VNetCtrl-1<-->Matcher1        8
Mmu1<-->GenReg3               11
Mmu1<-->ChecksumIPv61         11
InPreProcessor1<-->Mmu1       13
Mmu1<-->Matcher1              22
VNetCtrl-1<-->Mmu1            64
VNetCtrl-1<-->VNetCtrl-1      245
```

Figure 4.6: The TACO design tool.

XML module configuration file that is read by the design tool at startup. The module configuration file holds the name, the number and the type of sockets, a short description of functionality and a VHDL code generation template for each FU. This way each time a functional unit or some other new module is created into the SystemC, Matlab and VHDL component libraries the tool does not require any modifications; only the required information of the new module needs to be placed into the configuration file.

The information given in the module configuration file is accessible to the designer by double-clicking the corresponding module in the design window. The designer also has access to a component browsing window, in which the characteristics (operation, registers+types, logic depth, power use, etc.) of the functional units available in the TACO libraries can easily be reviewed. These characteristics are based on previous simulation and estimation results, and can be updated either manually (e.g. to support technology or standard cell library updates) or by directly importing results produced by the simulator and the estimator.

Once the design is complete, it may be necessary to modify the default Matlab parameters for estimator code generation. For example, standard cell library dependent values like average gate sizes may need modification. The code generation process is started by clicking the "Generate" button in the toolbar. Before generating the code, the tool parses the design for inconsistencies like a completely unconnected register. Then the tool generates the top level code corresponding to the designed architecture for all three target environments (SystemC, Matlab, VHDL).

Figure 4.7 shows generated SystemC, Matlab and VHDL code for *Matcher1*, the two interconnection buses, the interconnection network controller, the required connectivity and the system clock for the architecture shown in Figure 4.6. Recall from the earlier discussion on the estimation model that the Matlab code actually covers all the Matchers in the architecture. As can be seen in the code examples, the SystemC and Matlab codes appear more readable and compact in comparison to the VHDL code. In the case of SystemC, this follows from the use of object oriented programming techniques to automatize as much of the module instantiation as possible. In the case of Matlab, the compactness follows from the fact that the Matlab model is used only for estimation of physical characteristics (i.e. mathematical calculations), not for simulations. Hence the information needed by Matlab consists of input parameters for the estimation equations.

The results produced by SystemC simulations and Matlab estimations can be imported into the tool for review. The types of results obtained from simulations and estimations have been discussed in previous sections of this chapter. Figure 4.6 shows a part of the results obtained from SystemC simulations at the bottom of the tool window. The tool is showing the register

a) Generated SystemC code

```
sc_clock clk("clock",20);
NetControl nc("NetCtrl1");
nc.clk(clk);
Bus* bus1 = new Bus("Bus1");
Bus* bus2 = new Bus("Bus2");
Matcher* m1 = new Matcher("M1", clk);
bus1->insertOperand(m1);
bus2->insertOperand(m1);
bus1->insertData(m1);
bus2->insertData(m1);
bus1->insertTrigger(m1);
bus2->insertTrigger(m1);
bus1->insertResult(m1);
bus2->insertResult(m1);
nc.initialize();
```

b) Generated Matlab code

```
N_dbus = 2;
N_abus = 2;
W_dbus = 32;
W_abus 16;
FU_spec = [1 4 128];
FU_spec(1,:)  = Matcher;
N_ctrl = 1;
fld = 22;
gate_area = 54e-12;
```

c) Generated VHDL code

```
signal m1_op1_load, m1_od2_load, signal m1_trg_load :  std_ulogic;
signal m1_OP1 :  unsigned(datawidth-1 downto 0);
signal m1_OD2 :  unsigned(datawidth-1 downto 0);
signal m1_TR : unsigned(datawidth-1 downto 0);
signal m1_ResultR : unsigned(datawidth-1 downto 0);
signal m1_guard_bit :  std_ulogic;
matcher1_operand :  input_socket
   generic map (socket_address => 16#OPM1#)
       port map (clk => clk, reset => reset, dst_address =>
       icnw_dst_address, net_data_in => icnw_data, load =>
       m1_op1_load, socket_data_out => m1_OP1);
matcher1_data :  input_socket
   generic map (socket_address => 16#ODM1#)
       port map (clk => clk, reset => reset, dst_address =>
       icnw_dst_address, net_data_in => icnw_data, load =>
       m1_od2_load, socket_data_out => m1_OD2);
matcher1_trigger :  input_socket
   generic map (socket_address => 16#TM1#)
       port map (clk => clk, reset => reset, dst_address =>
       icnw_dst_address, net_data_in => icnw_data, load =>
       m1_trg_load, socket_data_out => m1_TR);
matcher1_result :  output_socket
   generic map (socket_address => 16#RM1#)
       port map (src_address => icnw_src_address,
       socket_data_in => m1_ResultR, net_data_out => icnw_data);
matcher1_fu :  matcher
   port map (clk => clk, reset => reset, op1_load =>
   m1_op1_load, od2_load => m1_od2_load,
   trg_load => m1_trg_load, OP1 => m1_OP1,
   OD2 => m1_OD2, TR => m1_TR, ResultR =>
   m1_ResultR, guard_bit => icnw_guard());
```

Figure 4.7: Examples of SystemC, Matlab and VHDL code generated by the TACO design tool. The code excerpts concern *Matcher1* of the architecture shown in Figure 4.6. For SystemC, the code also instantiates buses, the system clock and the network controller. For Matlab, three technology parameters have been left in the code to demonstrate pre-synthesis estimation capabilities.

transfer statistics from the simulations mapped into FU-to-FU transfers. The design tool allows the designer to integrally examine these results and to make design decisions based on them. Using these results to iteratively improve the architecture for a given target application was discussed in section 4.1 of this chapter.

The TACO design tool considerably speeds up the setup of simulations, physical parameter estimations and synthesis: no manual code writing is necessary to reach a functioning processor hardware model in all three of the mentioned processor model development environments. The tool also plays an important role in eliminating potential coding errors that could occur when manually writing the top level files. The top level code in the SystemC and especially in the VHDL model requires instantiating quite a few signals and connecting them between modules, which is a very error-prone process. Using a tool to do this automatically greatly reduces the time needed to construct and debug the top level files. In addition to generating code, the tool was also seen to aid the designer in reviewing simulation and estimation results, which is an important benefit in the kind of iterative design space exploration carried out in the TACO framework.

4.6 Chapter Summary

In this chapter we discussed a rapid system level protocol processor design methodology that was specified and built around the TACO hardware platform discussed in Chapter 3. The most important characteristics of the methodology were identified to be the following:

- Flow completeness: the TACO design flow defines all design procedures to be taken, starting from an application specification, to reach a gate-level synthesized protocol processor model and its application code.
- Support for iterative design space exploration and design improvement at the system level.
- Short turn-around time at the system level. Very fast and reliable execution of simulations and physical estimations.
- Reliable synthesis model generation after finding a good combination of hardware architecture and application code at the system level.
- No processor model rewriting necessary with modifications to an architecture instance; processor models need modifications only when new functional units are specified.

- Tool support for specifying architectures and for generating configuration files for the processor models.

- Customizability: new functional units can be added to the component library whenever necessary and without recompiling the design tool.

- Cost-efficiency resulting from the short turn-around time and the ability to use low-end computer systems and affordable, or even free, software for system level simulations and estimations.

To create such a design framework, several issues had to be addressed. First of all, processor models for simulation, estimation and synthesis needed to be created for the hardware platform. A key issue in the development of the processor models was that they needed to provide a simple enough API (application programming interface) through which models for any given TACO architecture could be generated. This was especially important for system level simulations of architectures: in addition to functional verification, TACO simulations need to provide hardware-accurate results like cycle-by-cycle simulation and register transfer statistics. Whilst providing such precise information, the simulations are also required to be very fast (i.e. to run a test bench in a matter of seconds) to facilitate rapid architectural exploration at the system level. The TACO simulation model was seen to meet these requirements through use of object oriented programming techniques and a heterogeneous level of abstraction in the implementation. For the system-level estimation model the requirements do not cause potential performance bottlenecks since the estimations are carried out as one-time calculations of pre-defined equations. Synthesis on the other hand is relieved of such performance requirements since it is put to use only after a suitable architecture has been found through system-level simulations and estimations; the most important requirements for the synthesis model are that its API supports generating a synthesizable description of a given TACO architecture, and that the generated description can be reliably synthesized.

It was also seen that although the processor model APIs considerably raise the abstraction level in implementational details for the designer, there are still quite a few modules and signals to instantiate and interconnect. To relieve the designer of doing this manually, a processor design and evaluation tool has been built for the TACO framework. The tool allows the designer to graphically specify a processor architecture and by a single click of a button to generate top-level simulation, estimation and synthesis configuration files for it. The tool also allows the designer to integrally examine simulation and estimation results and to make iterative design decisions and improvements based on them.

In Chapter 2 certain characteristics of protocol protocol processing were found to be shared by a variety of industry-standard protocols. However, knowledge of inter-protocol similarities may not be adequate when optimizing an architecture for a particular protocol processing application. To make the process of analyzing protocol processing applications easier for the designer, we defined in this chapter a sequence of object oriented UML-based analysis and refinement methods. With these methods and the results obtained in Chapter 2 it is possible to use the TACO design tool and the processor models to find an optimized, even though not necessarily the most optimal, combination of hardware architecture and application code for the original target application. With the techniques and tools presented in this chapter a designer is able to tell already at early stages of the design process whether an architecture will be able to perform a given target application within a given set of design constraints. Based on this knowledge, the designer is able to iteratively improve designs and explore more architectures in a short time frame.

The next chapter presents three protocol processor design case studies that demonstrate the use of the TACO methodology for designing and evaluating protocol processor architectures for given target applications.

Chapter 5

Design Cases

Chapter 4 discussed the TACO design methodology, a rapid system level framework for designing protocol processors. The key building blocks of the methodology were the TACO hardware platform, the processor models for simulation, estimation and synthesis, and a documented design flow. The design flow defines all procedures to be completed in order to reach a gate-level synthesized processor description and its program code with an application as a starting point. The processor description and the program code are optimized for the initial application.

To verify its capabilities, the TACO design methodology has been applied to several case studies. In this chapter we will look more closely at three such design cases. In the first one we argue that the TACO methodology is capable of rapid designer-driven design space exploration, and that the simulation, estimation and synthesis models for TACO processors provide reliable results in different phases of the design process. The target application for the first case study is processing a part of the ATM AIS (Alarm Indication Signal [60]) function in a 622 Mbps ATM network.

In the second case study we argue that the TACO architecture scales up to meet the performance provided by typical processing elements found in modern commercial network processor architectures. The TACO design methodology is used to find optimized processor architectures for the task of IPv6 [28] packet forwarding/routing at a throughput speed of 2 Gbps. Then, the executional and physical performance of these TACO architectures in running the most vital functions of the target application is compared to the performance provided by the microengines of an Intel IXP1200 processor running the same functions.

In the third case study we argue that with the tools and methods of the TACO framework both the hardware and the software parts of earlier designs can conveniently be reused. We also argue that any TACO processor can with little effort be used as an intellectual property (IP) block in a Network-on-

Chip (NoC) device using the TACO BVCI interface discussed in Chapter 3. The target application in the case study is IPv6 client operation in a 100 Mbps network environment as a part of a multimedia processing NoC platform. Not counting the NoC interface, the TACO IPv6 client core is built exclusively of existing modules in the TACO library. In addition, it is a simplification of a more complex previous experiment (i.e. the IPv6 router case study with 2 Gbps throughput requirement). Thus, no new protocol processing FUs were created, and extensive design space exploration was not necessary. These two factors caused the design process to speed up considerably.

5.1 Architectures for ATM AIS Processing

This case study applies the TACO design methodology to design space exploration. The target application for this case comes from ATM segment OAM F5 AIS (Alarm Indication Signal [60]) processing. More precisely, we consider an algorithm that analyzes incoming ATM cells to find out if a cell is a regular user cell, an idle cell[1] or an AIS-type operations and maintenance (OAM) cell. Depending on the incoming cell type, the algorithm swithces between normal operation mode and AIS operation mode. Incoming AIS cells cause the system to enter AIS mode: an AIS cell is sent every 1024 regular or idle cells. The outbound regular cells are buffered cells; if a regular cell arrives at the input while the system is in AIS mode, the system returns to normal processing. If an idle cell appears at the input, it is discarded. If there are no regular ATM cells to be sent, idle cells are generated and transmitted.

TACO application analysis of the AIS processing algorithm and the standards associated with it [60, 64] lead to the following results:

- The data word length of 32 bits should be used, and 0.35 μm CMOS technology should be used (at the time of performing this case study, no other technology libraries were available for use in the TACO project).
- Processing ATM AIS cells requires the following protocol processing functional unit types: Comparator, Matcher and Counter. In addition, all ATM processing implementations require header error checksum (HEC) calculations. HEC calculation (8-bit CRC) was not efficiently accomplished with the functional units already existing in the TACO libraries at the time of starting this case study. Thus, a functional unit for calculating the ATM HEC was added to the libraries. After this addition, all FUs required by the target application existed as 0.35 μm implementations in TACO libraries (see Chapter 3 for more details on the FU types).

[1]Idle cells contain no information and are sent when there is no user data to be sent.

- The target operating environment for the application is a 622 Mbps ATM network. In such a network, a new cell arrives for processing approximately $1.47 \cdot 10^6$ times per second. This means a maximum allowable processing time of approximately 0.68 μs per cell.

- The control flow of the ATM AIS processing algorithm to be implemented in this case study is given in Figure 5.1.

- Recall from Chapter 2 that ATM cells consist of 53 octets, of which the header takes up the first 5 octets or 40 bits (octet 5 is an 8-bit HEC checksum of the first 32 header bits). Idle ATM cells are identified by analyzing the first 4 octets of the cell header: the first 3 octets have an all-zero value, and the last one is `0000 0001` in binary notation. ATM segment OAM F5 cells are identified by the binary value `100` in the cell header's PTI (payload type identifier) field. A segment OAM F5 cell is an AIS cell if the payload of the cell starts with the octet `0001 0000` (binary notation). Determining the type of an ATM cell thus requires processing of the first 6 octets (or 48 bits) of the cell.

- ATM cell I/O is managed as described in Chapter 3.

After establishing that all the functional units needed for the target application exist in TACO libraries, the control flow diagram of the AIS processing routine (shown in Figure 5.1) was refined into sequential assembler code for a virtual processor as explained in Section 4.1 of the previous chapter.

The iterative design space exploration phase of the TACO flow started in this case study with evaluating a simple architecture instance, namely an implementation of the virtual processor for which the sequential assembler code was previously constructed. After simulating and estimating this instance, complexity was incrementally added to the following instances using SystemC simulation and Matlab estimation results from previous instances to guide the process. Here the addition of complexity to the instances is defined as increasing the number of functional units and interconnection buses in the architecture. The instances were specified and the top level description files were generated using the design tool described in Chapter 4 (Figure 4.7 shows examples of generated code).

For clarity, in the rest of this case study different TACO instances are identified using the following convention: *single-2* indicates the use of one FU of each needed type and two buses in the interconnection network. *Double-3* indicates the use of three buses and two of each of the following FU types: Matcher, Counter and Comparator. Using this naming convention, the first instance that was evaluated (i.e. implementation of the virtual processor) is called *single-1*.

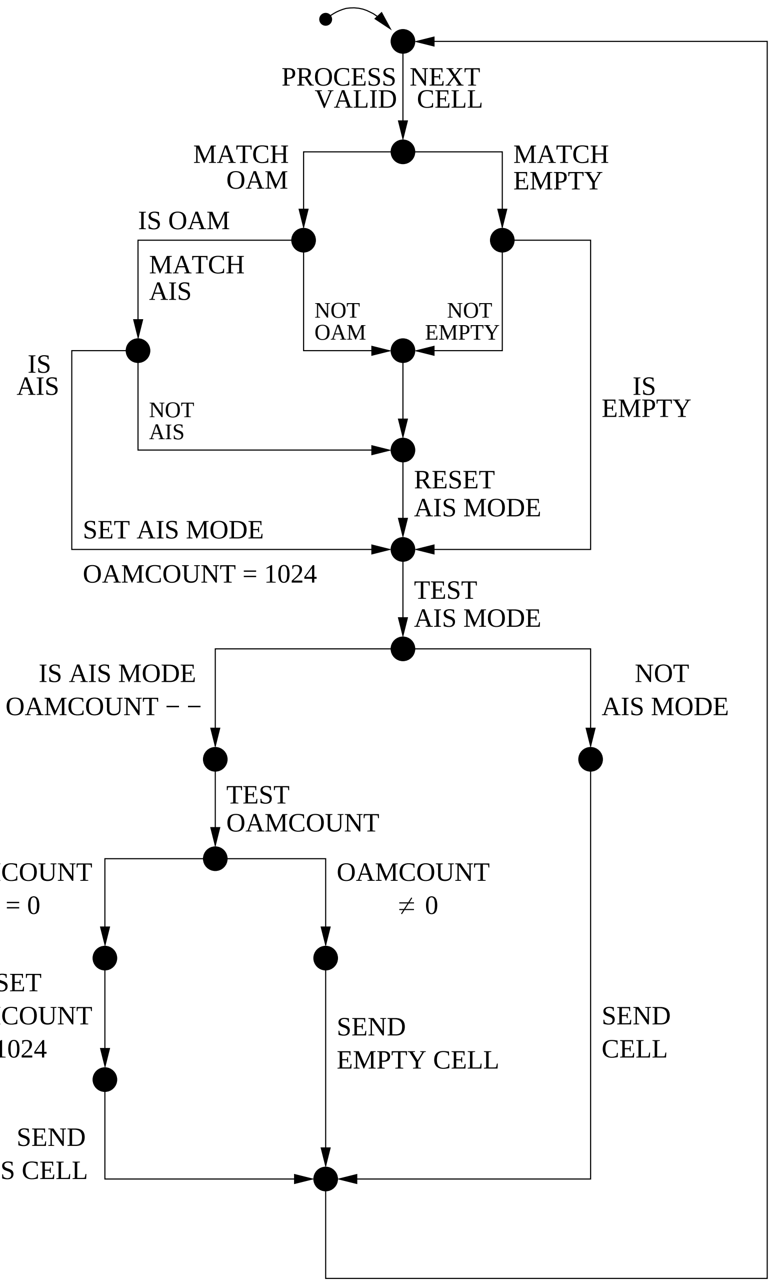


Figure 5.1: Control flow diagram of the ATM AIS (Alarm Indication Signal) cell processing loop of the case study. The starting point of the diagram is the situation at which a new valid cell is ready for processing.

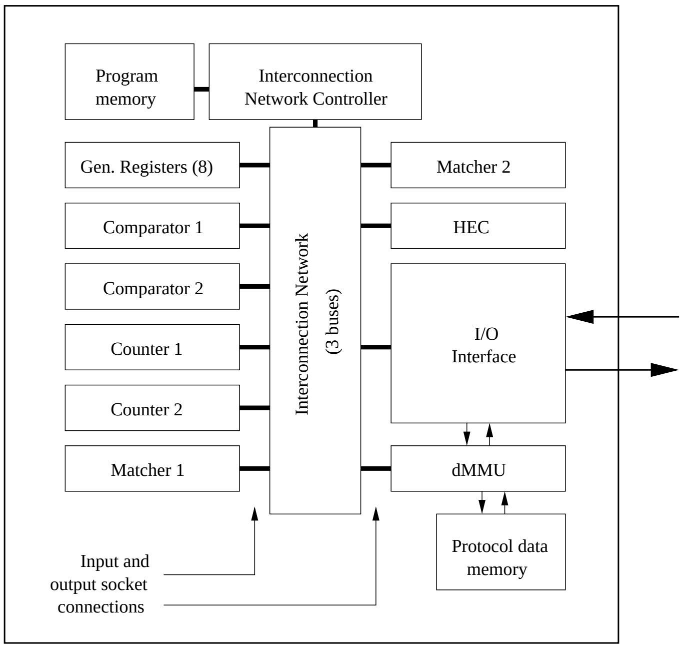


Figure 5.2: The TACO *double-3* ATM processor (functional view).

Altogether six architecture instances were explored in the iterative design space exploration part of the TACO design flow. As the last instance we simulated and estimated the *double-3* architecture. It turned out to be the least clock cycle consuming architecture of those explored for the target algorithm. A functional view of the *double-3* architecture is given in Figure 5.2. This figure also serves as a functional view of the rest of the explored architecture instances, since they differ from the *double-3* architecture only in the number of functional units and interconnection buses as described above. Note in this figure that the AIS implementation discussed in this case study did not require user data memory, and thus it was left out of the explored architectures to save chip area. Instead, a register unit with eight generic registers was used in the architecture.

The SystemC simulation and Matlab estimation results for the explored architecture instances are presented in Tables 5.1 and 5.2. In general, increasing the number of functional units and/or buses in the architecture decreased the minimum required clock speed for the target application. The bus utilization data from SystemC simulations (Table 5.1) indicates the amount of data transports on the interconnection network buses. Each data bus can be considered to have a data transport slot during each clock cycle. Therefore, the total number of available data transport slots (*move slots* in Table 5.1) is $n_{slots} = n_{buses} \cdot n_{cycles}$, where n_{cycles} is the number of clock cycles required to

a)

```
InBuffer > R1;          // cell header mem addr from input FU to R1
InBuffer > OMM;         // cell header mem addr into MMU op reg
0 > TRMM;               // trigger data mem read with offset 0

RMM > R2;               // put mem read result into register R2
1 > TRMM;               // trigger data mem read from address R1+1
1 > OCM1;               // Set up Comparator 1 for idle cell matching

R2 > TM1;               // perform OAM match operation
R2 > TEQ1;              // perform idle cell match operation
RMM > TM2;              // perform AIS match operation

RMM > R3;               // put data memory word to register R3
!a.!c:0 > OCM2;         // cell not idle/not OAM -> reset AIS mode
a.!b:0 > OCM2;          // OAM but not AIS -> reset AIS mode

a.b:1 > OCM2;           // if OAM and AIS -> set AIS mode
1 > TEQ2;               // Test AIS mode
NOP;                    // No Operation
```

b)

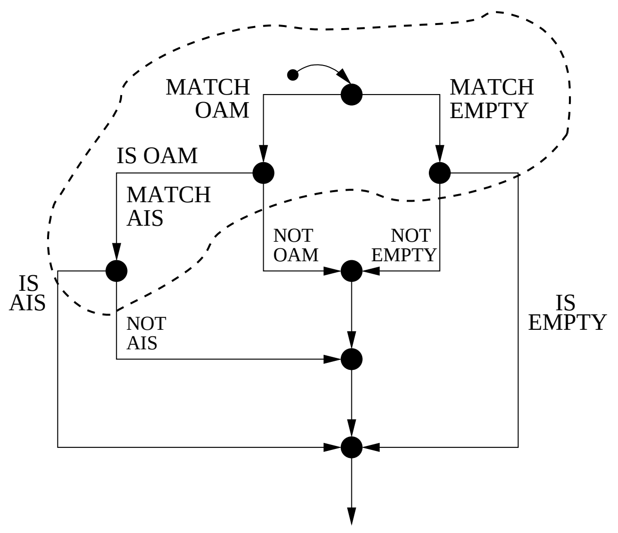


Figure 5.3: Relation between the number of processing elements and data transport capacity in a TACO processor. *a)* Assembler code excerpt for a part of the target application. *b)* The part of the control flow diagram that is implemented by the assembler code in *a)*. The assembler code in boldface performs the three *if-then* statements needed by the control flow specified by the dashed line.

perform the target algorithm for processing one PDU. In the simplest case, i.e. the *single-1* instance, bus utilization is naturally 100% (the only bus is used all the time until the algorithm finishes). The more interesting result is that with double FUs the bus utilization remains very high with the addition of more buses into the architecture. This is a major contributor to overall processor performance as seen in table 5.1; as the number of functional units of the same kind increases, more buses are needed to fully utilize the processing capacity of the functional units. The utilization information is also used in the Matlab model to estimate the total energy consumed during the execution of the algorithm.

The importance of providing adequate data transport capacity for the functional units is best highlighted using an example from the AIS processing algorithm's assembler implementation for the *double-3* architecture (Figure 5.3). In the example, each line represents one TACO subinstruction; thus, each block of three lines represents a single TACO instruction word executed in one clock cycle. In the code example, the incoming cell has already passed through the preprocessing circuit. Thus, the cell has been verified for correctness (i.e. the checksum has been calculated) and has thereafter been stored into the protocol data memory. The starting memory address of the cell has been stored into the Input FU (labeled InBuffer in the code example). Recall from the beginning of this section that to find out whether an ATM cell is an OAM AIS cell, the first two 32-bit data words of the cell need to be analyzed. In addition, it is also necessary to find out if a cell is idle. In a 32-bit processor this means altogether three *if-then* type operations, of which one is nested below another one (see Figure 5.3 *b)*, the part marked with a dashed line). However, in the *double-3* architecture it is possible to carry out these three *if-then* operations in one clock cycle, as long as the match patterns for OAM and AIS identification have already been stored into the two Matcher FUs in the architecture. The TACO subinstructions marked in boldface in Figure 5.3 execute the three *if-then* operations using a Comparator FU for idle cell matching. As seen in the assembler code in Figure 5.3 *a)*, in addition to two Matchers and a Comparator at least three interconnection buses are needed to execute these three operations in one cycle. This example clearly shows that adding functional units into an architecture does not necessarily improve algorithm execution speed; attention must also be paid to the overall data transport capacity of the architecture.

The previously defined execution cycle count per PDU (n_{cycles}) for an architecture instance is obtained through SystemC simulations. This value together with the maximum allowable processing time per PDU (T_{max}) obtained through application analysis define a minimum value for the required clock frequency of the processor:

Architecture instance	Execution cycles	Required clock frequency	Move slots	Unused slots	Bus utilization
single-1	121	178 MHz	121	0	100%
single-2	68	100 MHz	136	15	89%
single-3	49	72 MHz	144	42	71%
double-1	90	132 MHz	90	0	100%
double-2	53	78 MHz	106	1	99%
double-3	36	53 MHz	108	2	98%

Table 5.1: Simulation results for the six architecture instances explored in the ATM AIS case study. With the indicated *Required clock frequencies* all candidates have the same execution time.

$$f_{clock}^{min} = \frac{n_{cycles}}{T_{max}} \tag{5.1}$$

For example, for a maximum processing time of $T_{max} = 0.68\ \mu$s per ATM cell (obtained in the application analysis for this case study), and the cycle count of $n_{cycles} = 121$ per ATM cell (obtained from SystemC simulations of the *single-1* candidate) produces a minimum clock frequency requirement of $f_{clock}^{min} = 178$ MHz.

f_{clock}^{min} multiplied by 1.2 (as explained in Section 4.3 of the previous chapter) was used as a timing constraint in the Matlab model when estimating the area use and the power consumption of architecture instances. By combining the bus utilization results from SystemC simulation and power analysis results from Matlab estimation, an estimate of the energy consumed by a task running in the processor is obtained. The Matlab estimation results are shown in Table 5.2.

Co-analyzing the results from SystemC and Matlab revealed that in two and three bus configurations the double-FU instances require larger area than the single-FU -instances. This is not surprising; in the 0.35 μm CMOS technology logic gates are still relatively large and the more there are functional units the more there are gates which increase the total area. The cases with only one bus need further analysis; as seen in Table 5.1, these instances have the most stringent clock frequency demands. The Matlab estimations showed that minimum-sized gates are too slow in these cases. The first instance (*single-1*) needed gates of scaling factor 3 and the second one (*double-1*) gates of scaling factor 2 to reach the target clock frequency. For all the rest architecture instances minimum-sized gates could be used in order to meet the target frequency constraints and thus only the number of FUs affected the logic area.

Architecture instance	Relative task energy estimate	Logic area estimated	Logic area synthesized	Processor area (estimated)
single-1	1.00	2.08 mm^2	-	2.63 mm^2
single-2	0.40	0.89 mm^2	-	1.20 mm^2
single-3	0.31	0.89 mm^2	0.87 mm^2	1.27 mm^2
double-1	0.96	1.88 mm^2	-	2.39 mm^2
double-2	0.56	1.41 mm^2	-	1.96 mm^2
double-3	0.43	1.41 mm^2	1.25 mm^2	2.09 mm^2

Table 5.2: Estimation and synthesis results for the six architecture instances explored in the ATM AIS case study. The *Relative task energy estimates* indicate the amount of energy needed to process the target application in relation to the candidate with the highest task energy consumption (1.00 indicates highest consumption).

The same analysis can be extended to average power consumption, or more precisely, task energy consumption (see Table 5.2). Task energy is the total amount of energy needed to execute a task once and is directly proportional to the average power consumed by the corresponding architecture instance. For the two 1-bus instances, the single-FU instance consumes more energy than its double-FU counterpart; for the two- and three-bus instances the situation is vice versa. All double-FU instances consume more energy than the single-FU instances during *one* clock cycle. However, the larger number of clock cycles needed by single-FU instances for executing the same algorithm more or less compensates this difference. This and the difference in gate sizes are also the reasons for which the *single-1* instance requires more energy than the *double-1* instance for the same task.

Two of the six architecture instances were synthesized at 200 MHz to verify the simulation and estimation results. *Single-3* was chosen for synthesis because of its excellent power and area characteristics, and *double-3* because of its ability to operate at a low clock frequency (this allows for the most cost-efficient co-circuitry). The synthesized logic areas for these two instances are shown in Table 5.2. The Matlab model is able to estimate both the logic area and the entire processor area; Thus, also the processor area estimates have been included in Table 5.2. As can be seen in this table, the logic area estimates and the values obtained from synthesis are quite close.

The ATM AIS case study verified that the tools and techniques provided by the TACO framework were well applicable to this kind of rapid system-level protocol processor design. The SystemC simulations and Matlab estimations were seen to provide reliable results, thus aiding the designer in making design decisions already early in the design process.

5.2 Architectures for IPv6 Routing

This case study applies the TACO design methodology to finding architectures for IPv6 routing and packet forwarding. The key goal in this case study is to compare the executional and physical performance of TACO architectures to the performance provided by a commercial programmable network processor, namely the Intel IXP1200 processor.

The IPv6 protocol was discussed in Chapter 2. IPv6 routing involves receiving datagrams from adjacent networks, checking datagram validity, deciding the outbound interfaces to which the received datagrams should be forwarded, and sending the datagrams out on the appropriate interface. Also, a router should maintain and update a routing table that contains information about the network topology. Routing table updates are made based on special datagrams received occasionally from adjacent routers. In addition, the local routing table information needs to be broadcasted occasionally to adjacent routers to facilitate their routing table updates. Thus, two types of packets need to be handled by IPv6 routers: regular datagrams that pass through the router (for which the router makes a routing decision), and administrative datagrams consumed by the router (e.g. the ones used for routing table updates). Clearly, executing the actual routing process is the fast path and also the potential performance bottleneck in an IPv6 router; the routing table updates (slow path of processing) are sent and received quite infrequently compared to the rate at which regular datagrams pass through the router. Thus, emphasis in an IPv6 router implementation should be in efficient processing of the tasks that are needed in the fast path of processing described above.

Routing decisions are made by searching the routing table for an entry that most closely matches the destination address of the datagram to be routed. Depending on the type of router in question (e.g. small office/campus area router vs. national or international trunk network router), the routing table can consist of less than ten entries or more than a thousand entries. Finding the correct entry in the routing table can therefore require a long computation time. In addition to varying routing table lengths, different routing algorithms and hence different routing table access schemes are used in different types of IPv6 routers: for a small office router a single-chip router using sequential routing table access might be adequate, whereas for a trunk network a complex system constructed of for example several protocol processors, content-addressable memories (CAMs) and memory access accelerators like the iFlow address processor [82] might be needed.

In this case study a single-chip small office router built on the TACO hard-

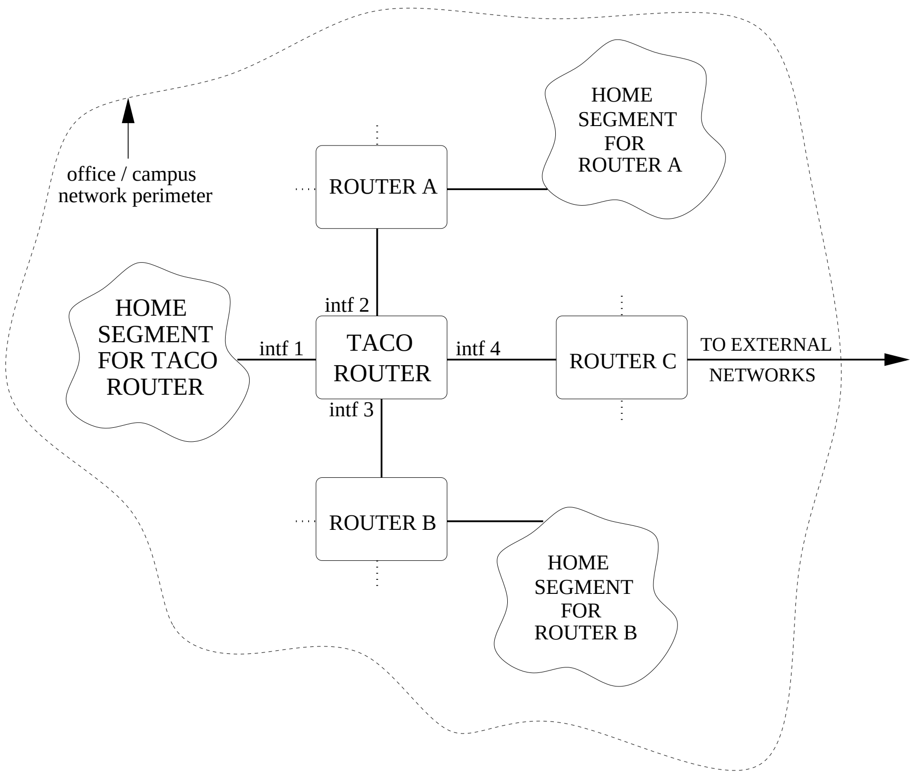

office / campus network perimeter
ROUTER A
HOME SEGMENT FOR ROUTER A
HOME SEGMENT FOR TACO ROUTER
intf 1
intf 2
intf 3
intf 4
TACO ROUTER
ROUTER C
TO EXTERNAL NETWORKS
ROUTER B
HOME SEGMENT FOR ROUTER B

Figure 5.4: A possible network topology for the IPv6 router case study. The TACO router is connected to a home network segment and three other routers. Router C provides access to networks outside the office or campus LAN in which the TACO router resides. Routers A and B provide access to their home segments and to other routers and networks in the office or campus network.

ware platform is considered. A possible operating environment, i.e. a network topology, for such a router is given in Figure 5.4. The router runs a sequential routing table lookup algorithm. The implementation covers OSI layer 3 and expects the underlying network to be an Ethernet; Ethernet processing is expected to be managed by appropriate off-the-shelf LAN circuitry for each of the router's interfaces. The way the Ethernet circuitry and a TACO processor are interconnected depends on the products used. The TACO side of this interconnection was specified in Chapter 3 (See Figure 3.12). In this case study the LAN circuitry of each network interface is expected to provide fully assembled decapsulated IPv6 datagrams into a buffer accessible by the router processor (more precisely, by the Input FU of Figure 3.12). The same kind of interconnection is assumed for outbound IPv6 traffic.

Some of the key processing tasks needed in IPv6 routing have been implemented for a 200 MHz Intel IXP1200 network processor in a recent Master's thesis [120] at the University of Turku. The work also involved making performance analyses for the implemented IXP1200 routing functions. The functions were implemented and tested on the IXP1200 developer workbench, a software environment for developing and simulating applications for the IXP1200 processor. In this case study an important goal is to compare the results of executing these tasks on the IXP1200 to the results of running them on TACO processors. The aim is not to see which of the two architectures is faster, but to obtain some understanding on the characteristics and capabilities of the TACO hardware platform in comparison to a typical commercial programmable network processor architecture.

The IXP1200 is a multiprocessor with six programmable processing elements called microengines accompanied with a programmable general purpose processor core. The microengines are intended for executing protocol processing operations, while the processor core is intended for system management and general purpose processing tasks. Each microengine executes four threads, which means that the same code can be executed for four PDUs in parallel. Additional PDU-level parallelism is accomplished in the IXP1200 by using the same program code in several microengines. Something similar could be achieved in the TACO hardware platform either by using several TACO processors running identical application code in parallel, or by constructing an architecture in which dedicated sets of buses and FUs execute identical routines in parallel.

Specifically, in the rest of this case study we will focus on the following IPv6 routing functions that have also been implemented on the IXP1200 processor:

- Datagram header validation (fast path and slow path)
- Routing table lookup using sequential access (fast path)
- Calculation of the Internet checksum for outbound ICMPv6 messages (slow path)

Simplified control flow diagrams for these functions are given in Figure 5.5. Of these functions, header validation and routing table lookup form the most vital processing track in routing IPv6 datagrams. In IPv6, Internet checksums are no longer used in regular datagram headers. Instead, they are used in upper layer protocols and inter-router control messages (more specifically, ICMPv6 messages). However, in the currently used Internet protocol version 4 (IP, IPv4) the Internet checksum is calculated for each datagram header in each router (i.e. Internet checksum calculation is needed in the fast path of processing). In general, as seen at the end of Chapter 2, checksum calculations are performed in quite a few protocols. Thus, it is important to know

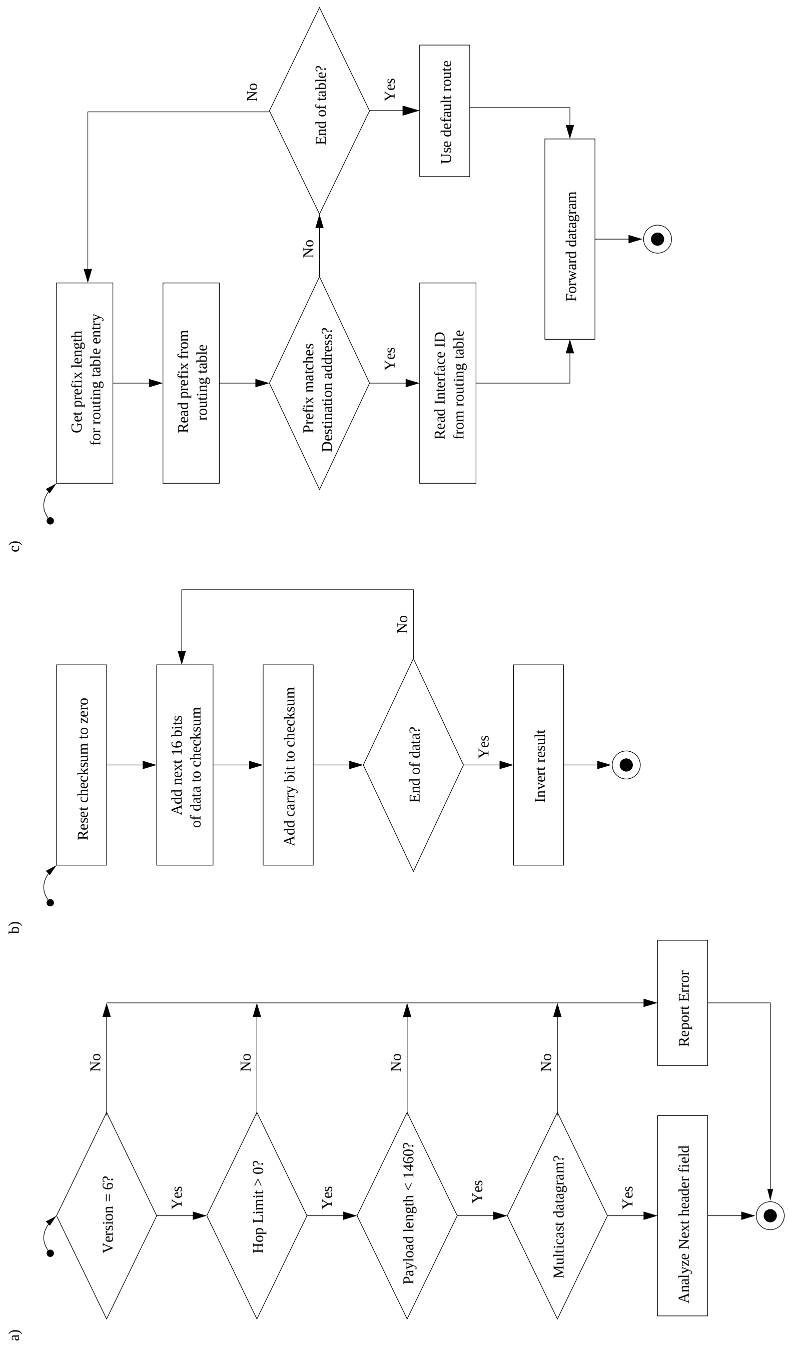

a)
Version = 6?
Hop Limit > 0?
Payload length < 1460?
Multicast datagram?
Analyze Next header field
Report Error
Yes
No
b)
Reset checksum to zero
Add next 16 bits of data to checksum
Add carry bit to checksum
End of data?
Invert result
Yes
No
c)
Get prefix length for routing table entry
Read prefix from routing table
Prefix matches Destination address?
End of table?
Read Interface ID from routing table
Use default route
Forward datagram
Yes
No

Figure 5.5: Simplified control flow diagrams for IPv6 routing tasks analyzed in the case study. *a)* IPv6 header validation, *b)* Internet checksum calculation and *c)* routing table lookup.

how well a protocol or network processor performs in checksum calculation. In terms of the TACO architecture, its capabilities in checksum calculation have not been properly tested prior to this case study: in the ATM case study the HEC checksum was calculated for only the 32 ATM header bits, which can not be seen as an intensive and performance-critical processing task. On the other hand, in IPv6 routing the checksums are calculated for entire datagrams carrying ICMPv6 messages (i.e. for up to 1500 bytes), which makes them a suitable checksum testbed for TACO architectures. Also, checksum calculation should provide an interesting subtopic for the performance comparisons between the TACO and the IXP1200 architectures: TACO processors have hardware-optimized execution for checksums, for which reason checksum calculation should be quite efficient in a TACO processor. The IXP1200 architecture on the other hand relies more on general purpose programming and processing techniques in calculating checksums, which in some situations could result as a bottleneck in processing performance.

The IXP1200 implementations of the routing functions discussed in this section have been coded using IXP1200's native assembler language (the *microengine instruction set*). The microengine code has not been extensively optimized, and each function has been implemented for execution in all four threads of a single microengine. This means that each time the processing of a function is completed in a microengine it has actually been completed four times (e.g. for four different PDUs).

As with the design space exploration case study discussed in the previous section, also this case study starts with application analysis as described in Chapter 4. The analysis of IPv6 routing and the standards associated with it (e.g. [28, 76]) lead to the following results:

- The data word length of 32 bits should be used, and 0.18 μm CMOS technology should be used (at the time of performing this case study, the 0.18 μm technology had become available for use in the TACO project).

- Processing IPv6 datagrams requires the following protocol processing functional unit types: Comparator, Matcher, Counter, Internet Checksum, Masker, Shifter and ICMPv6 FU. In addition to these, a routing table unit and a router local information unit are needed. These units are more like specialized registers or memory blocks than functional units. Since the 0.18 μm CMOS technology had become available for use in the TACO project, a decision for technology transition was made. Thus, the VHDL descriptions for all functional units needed in the IPv6 router case study were rewritten using the 0.18 μm libraries, and the Matlab and SystemC models were updated correspondingly (see Figure 4.3 in Chapter 4 for more information). After this transition, all FUs required by the target application existed as 0.18 μm implementations in TACO libraries (see Chapter 3 for more details on the FU types).

- The router should have four interfaces. Additionally, each interface of the router can be either sending or receiving datagrams at any given moment; an interface can not be both sending and receiving datagrams at the same time. This kind of router could be connected to e.g. one local network segment and three other routers, of which one could provide a route to external networks. A network topology matching this description, i.e. a possible operating environment for this router, is shown in Figure 5.4.
- IPv6 is a connectionless network layer (OSI layer 3) protocol and its average throughput speed in a given network environment depends on the type and load of the underlying LAN(s). Thus, a timing constraint can not implicitly be specified. However, the following characteristics of IPv6, Ethernet and the target application define an exaggerated worst-case timing constraint:
  - The routing process to be implemented in this case study expects the underlying network to be an Ethernet. Detailed guidelines for transmitting IPv6 datagrams over Ethernet networks are given in RFC 2464 [22]; most importantly, it specifies a default (maximum) IPv6 datagram length of 1500 octets for the transmission. This is due to the maximum Ethernet frame size of 1526 octets (see Chapter 2): with MTU = 1500 octets, each Ethernet frame is able to carry one maximum-size datagram, and on the other hand any valid IPv6 datagram will with certainty fit into a single Ethernet frame.
  - The router should be able to operate on top of 1 Gbps Ethernet networks. In normal networking conditions a situation in which each of the router's interfaces would be carrying data constantly at its peak bandwidth is an unlikely and unusual event and would probably be a sign of a severe network problem: a basic characteristic of most computer networks is a bursty nature for data transmission in which short high-bandwidth bursts are followed by periods of non-existent or very low-bandwidth traffic. However, to obtain a worst case timing constraint, we here assume a situation in which two of the router interfaces are each providing incoming datagrams at a constant speed of 1 Gbps for routing and forwarding through the other two interfaces. This kind of a situation is not realistic, since it requires a second assumption: routing decisions for the incoming datagrams must be evenly distributed among the

    two outbound interfaces. If this weren't the case, the underlying Ethernet would not be able to handle the traffic: one of the interfaces would require more than 1 Gbps of outbound bandwidth or the network segment would be severely congested. These assumptions require the router to be able to process incoming IPv6 datagrams with the speed of 2 Gbps. With such a target processing speed, the possibilites for the router to be congested when used on top of 1 Gbps Ethernets should be minimal.

  At 2 Gbps throughput and $1526 \cdot 8$ bits per datagram for the reasons explained above, we get an IPv6 datagram rate of about 164 000 datagrams per second. This means that a new datagram may arrive for processing every 6 μs.

  However, since a 200 MHz IXP1200 processor was used to implement the target routing functions in [120], the TACO architecture instances evaluated in this case study were also targeted to the 200 MHz clock frequency: this is necessary to warrant any kind of performance comparison between the two architectures. To achieve 2 Gbps throughput at 200 MHz, no more than 1200 cycles per IPv6 datagram may be used for processing.

- The PDU I/O interface was implemented as described in Chapter 3.

The previous discussion suggested that IPv6 routing is a rather complex protocol processing application. However, it can reasonably easily be split into smaller tasks that, in terms of software, would become subroutines in the final application code. This characteristic of the application was exploited also in this case study: instead of developing the entire application from specification to sequential assembler code for a virtual processor (as described in Chapter 4), the specification was split into smaller parts that were individually refined towards the virtual processor assembler level. Thus, the application analysis was completed in a situation in which the functional unit types needed for IPv6 routing had been determined, and in which virtual processor assembler code existed for the key tasks (or subroutines) of IPv6 routing. In addition to clarifying the refinement process, this approach was chosen also to ensure implementational accuracy to the IXP1200 implementations of the routing functions.

In the iterative design space exploration phase of the TACO design flow (see Chapter 4) we decided not to simulate and estimate the virtual processor, but to directly add more processing elements to the architecture; this decision seemed warranted due to the complexity of the target application as a whole. Also, a decision was made to use the same functional units in each architecture instance to be explored. The number and type of FUs selected

	Header validation	Route lookup	Internet checksum	Maximum needed
Checksum FUs	-	-	1	1
Comparators	1	1	-	1
Counters	-	2	-	2
Matchers	3	-	-	3
Maskers	1	1	1	1
Shifters	1	1	1	1
Routing table FU	-	1	-	1
Local Info FU	-	1	-	1
ICMPv6 FU	1	-	-	1

Table 5.3: Number of TACO functional units needed in each target IPv6 routing function to execute the function without unnecessary operand transports in most critical execution paths.

for the instances were obtained by reviewing the virtual assembler code for each target function. This process involved examining the code to find operations of the same kind needed consequtively or within a sequence of a few instructions in the most performance critical parts of each function (e.g. main loops or main repetitive code sequences). In these parts of the functions, several data transports and thus also several clock cycles may be unnecessarily spent for transporting operands to the FUs between calculations. For example, a Matcher unit requires two operands and a trigger data value for each computation (see Chapter 3 for more details). Now, if a function contains three consequtive Match operations, 9 data transports are needed to carry out the operations using just one Matcher unit. Consider a situation in which these three Match operations are used inside a loop that needs to be repeated several, for example, 10 times: 90 transports would be required to complete the operations. On the other hand, by using three Matcher FUs that each hold static operands during the loop, only 36 transports would be needed (six transports to store the operands in the FUs, and 30 transports for the 30 Match operations).

The code examination procedure was carried out for each of the target routing functions as outlined above. As a result, Table 5.3 lists the number and type of TACO FUs needed to execute each function without the occurrence of unnecessary operand transports as described above. The values listed in the *Maximum needed* column of Table 5.3 were decided to be the functional unit configuration for all TACO architecture instances explored in this case study: this way all target functions could be executed without unnecessary operand transports. Also, the resulting TACO architecture instances would

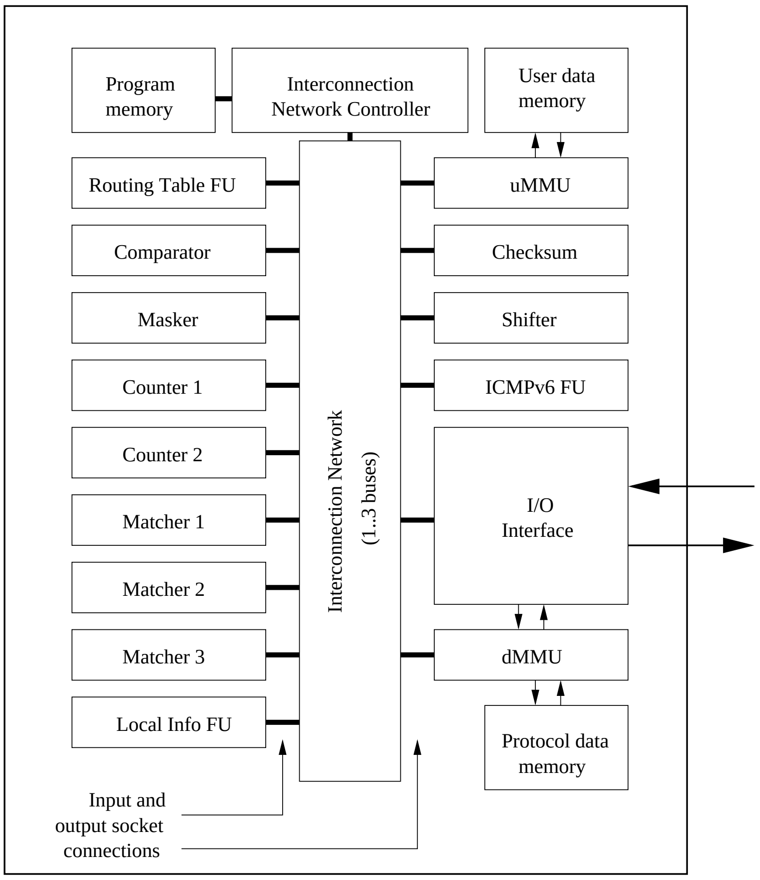


Figure 5.6: Functional view of the TACO IPv6 router processor architectures examined in this case study.

be relatively small and simple, which would better warrant a comparison to IXP1200 microengine implementations of the functions.

Thus, all TACO instances explored in this case study have the same functional unit configuration but a different interconnection bus count. One, two and three bus configurations were chosen for exploration; based on earlier experiments such as the ATM case study presented in the previous section, a three bus configuration was expected to provide sufficient data transport capacity for all the target functions.

Figure 5.6 shows a functional view of the TACO architecture instances explored in this case study. In the rest of this section, we will distinguish these architectures from one another using a notation that specifies the number of interconnection buses in the architecture: *TACO-1*, *TACO-2* and *TACO-3*.

SystemC and Matlab top level files were again generated for the instances

TACO instance	Routing function	Exec. cycles	Exec. time [ns]	Move slots	Unused slots	Bus util.
TACO-1	Header validation	32	160	32	0	100 %
	Route lookup (5)	63	315	63	3	95 %
	Route lookup (50)	510	2 550	510	0	100 %
	Checksum (64)	41	205	41	0	100 %
	Checksum (1500)	755	3 775	755	0	100 %
TACO-2	Header validation	20	100	40	7	83 %
	Route lookup (5)	38	190	76	17	78 %
	Route lookup (50)	305	1 525	710	101	86 %
	Checksum (64)	24	120	48	3	94 %
	Checksum (1500)	381	1 905	762	3	100 %
TACO-3	Header validation	13	65	39	7	82 %
	Route lookup (5)	32	160	96	36	63 %
	Route lookup (50)	254	1 270	762	252	67 %
	Checksum (64)	21	105	63	13	79 %
	Checksum (1500)	378	1 890	1 134	372	67 %

Table 5.4: Simulation results for the three architecture instances explored in the IPv6 router case study. Execution times have been calculated assuming a 200 MHz clock frequency. *Route lookup (50)* indicates a routing table with 50 entries, and *Checksum (1500)* indicates an Internet checksum calculation for 1500 bytes of data.

as described in the previous section and in Chapter 4. Then, the virtual assembler code for each function was prepared first for the *TACO-1* architecture; this meant modifying the virtual code to take advantage of the additional FUs in the architecture (and resulted in the unnecessary operand transports to be removed from the code). Once the code had been prepared for the *TACO-1* architecture, modifying it for use in the more complex architectures was quite straight-forward; the code was improved to take advantage of the increased data transport capacity in the *TACO-2* and *TACO-3* architectures while maintaining data dependencies. Once the application code for each function had been prepared for execution on each of the architecture instances, the SystemC and Matlab models were used to simulate and estimate the instances.

Table 5.4 shows the results of SystemC simulations for each function executed on each architecture instance. For *Route lookup* and *Checksum calculation*, two results are given for each instance. The execution time of the *Route lookup* function depends on the number of routing table entries to be analyzed before a routing decision is made. *Route lookup (5)* indicates five entries, and *Route lookup (50)* indicates 50. For the checksum calculations, a similar effect on execution time is caused by the length of the data for which the checksum is calculated. *Checksum calculation (64)* indicates a calculation for 64-byte data, and *Checksum calculation (1500)* for 1500-byte data. The IXP1200 implementations of the functions were tested in [120] using 5 entries for route lookup and 64-byte data for checksum calculation; the same values were used here to facilitate a performance comparison. The more demanding values of 50 entries for *Route lookup* and 1500 bytes of data for *Checksum calculation* were also simulated to obtain better estimates for processor performance in the entire routing process (discussed later in this section). The term *slot* is used in the simulation results as defined in the previous section (the ATM AIS case study).

It is interesting to notice that the IPv6 routing functions are not able to use the interconnection buses as efficiently as the ATM AIS processing routine examined in the previous section. Possible reasons behind this are the different PDU sizes used in the two protocols, and the fact that no table lookups needed to be implemented in the ATM case study. The IPv6 router function most closely resembling the ATM AIS processing routine is the *Header validation* function; in both of these tasks PDU header fields are analyzed to find out certain specific properties of the PDU in question. As it turns out, the IPv6 *Header validation* function is also best able to take advantage of increases in data transport capacity (see Table 5.4, *Header validation* bus utilizations for *TACO-2* and *TACO-3*). On the other hand, in the *Checksum calculation* function, increasing data transport capacity beyond two buses provides practically no performance improvement in this case study. For example, see Table 5.4 for the *Checksum calculation (1500)* results of the *TACO-3* instance: there are almost as many unused transport slots as there are clock cycles used for executing the function, which means that one bus is virtually unused during the execution. In the other two functions, the *TACO-3* instance is clearly faster than the other instances due to its higher data transport capacity.

Table 5.5 shows the results of Matlab estimations and VHDL synthesis for each of the explored TACO instances. The results in this table do not include the area and power costs produced by protocol data, user data and program memory blocks nor the small memory blocks associated with the routing table FU. The power estimates assume a worst case situation in which all FUs and sockets are active at all times. Thus, the power estimates are highly exaggerated; in typical application execution in TACO processors $n \cdot 2$ functional units and $n \cdot 2$ sockets (n is the number of interconnection buses) and the interconnection network controller are active while the rest of the FUs and sockets are more or less idle.

Architecture instance	Worst case power estimate	Logic area estimated	Logic area synthesized	Processor area (estimated)
TACO-1	155 mW	0.48 mm^2	0.44 mm^2	0.53 mm^2
TACO-2	165 mW	0.50 mm^2	0.47 mm^2	0.66 mm^2
TACO-3	177 mW	0.52 mm^2	0.52 mm^2	0.79 mm^2

Table 5.5: Estimation and synthesis results using 0.18 μm technology for the three TACO architecture instances explored in the IPv6 router case study. The results do not include memory blocks.

As the results in Table 5.5 show, the Matlab model is quite accurate in estimating the logic area; some additional pessimism in the model might be appropriate (and was later implemented into the model) to ensure that the estimates will not produce a result smaller than actually produced by gate-level synthesis (see the next case study for further discussion on this topic). The *processor area* estimates also include the area required by the interconnection buses and any other wiring in addition to the reported logic area. The increases in logic areas and worst-case power consumption of the architectures with more than one bus are produced by an increase in the sizes of the sockets and the interconnection network controller due to increasing interconnection network complexity.

We recall that in the target 2 Gbps throughput speed range each PDU needs to be processed in less than 6 μs or 6 000 ns. The fast path of the entire IPv6 routing process more or less consists of receiving a datagram, validating it (and decreasing its *hop limit* header value), making a routing decision for it, and sending the datagram. Sending and receiving the datagrams are not actual protocol processing tasks and the maximum speed at which these operations can be done depends on the LAN co-circuitry used. However, assuming a worst-case scenario in which the processor would be idle during the receive and send operations (which is not normally the case in TACO processors), altogether 750 TACO cycles[2], or 3 750 nanoseconds, would be consumed for the receive and send operations. As seen earlier in this case study, routing table size has a major effect on the overall PDU processing time especially when a sequential access scheme is used. It is easily seen from Table 5.4 that for a routing table of only five entries all TACO architectures are easily able to function at the target 2 Gbps speed range. However, for 50 entries, the *TACO-1* architecture consumes too many cycles if the processor is idle during the receive and send processes. *TACO-2* and *TACO-3* would be able to function in the target speed range even in this worst case scenario. Thus, we

[2]Assuming a 1500-octet, or 375-word, datagram.

Function	TACO-1 [ns]	TACO-2 [ns]	TACO-3 [ns]	IXP1200 μE [ns]
Header validation	160	100	65	84
Route lookup (5)	315	190	160	478
Checksum (64)	205	120	105	1 983

Table 5.6: Required processing time per PDU in IPv6 routing functions for the explored TACO instances and for one IXP1200 microengine at 200 MHz.

conclude that when synthesized at 200 MHz, the *TACO-2* and *TACO-3* fulfill the design constraints for this target application. The *TACO-1* instance is able to fulfill the constraints if the routing table is sufficiently small (no more than 30 entries). Naturally, routing table access speed can be dramatically improved by resorting to more intelligent table lookup schemes and hardware accelerators. We have analyzed the use of different IPv6 routing table access solutions in the context of TACO processors in [73].

As a conclusion to this case study, Table 5.6 shows the processing times of the target IPv6 routing functions for the TACO instances examined in this case study, and for one of the six (identical) microengines of an Intel IXP1200 processor as analyzed in [120]. The IXP1200 results assume that four PDUs are processed at the same time using the four microengine processing threads. For example, the *Header validation* actually consumes 335 ns but produces a validation result for four PDUs simultaneously.

In general, the TACO instances seem to provide somewhat better execution times in all the implemented routing functions. The only exception is the *Header validation* function, in which the *TACO-1* and *TACO-2* are slightly slower (in absolute processing speed) than the IXP1200 microengine. This is due to lacking data transport capacity in the *TACO-1* and *TACO-2* architectures; the *Header validation* function consists mostly of relatively straightforward Boolean comparisons that can be executed in parallel if enough transport capacity is available. However, in the *TACO-1* and *TACO-2* architectures the available data transport capacity is limited (1 and 2 data buses, respectively). Perhaps the most surprising results are obtained for the *Checksum* function, in which the TACO instances are roughly 10-20 times faster than the IXP1200 microengine. This is due to the fact that TACO processors have native hardware support for calculating checksums: it is only necessary to transport all data words to be checksummed to the Checksum FU. Contrary to this, in the IXP1200 microengine implementation described in this case study the checksum calculation is carried out as a full software implementation for tasks like data word splitting and one's complement summing (see Chapter 2 for more information on Internet checksum calculation, and

Chapter 3 for a specification of the TACO checksum FU). Also, the IXP1200 implementation has not been written by an experienced IXP1200 programmer but by a student working on his M.Sc. project, and on the other hand the TACO implementation has been programmed by an experienced TACO programmer. Still, both implementations follow the block diagram given in Figure 5.5 *b)* and thus the TACO architecture can be determined to provide better support for IP checksum calculation than an IXP1200 microengine.

It is important to remember that an IXP1200 processor is formed of six microengines and a general purpose processor core; thus, it is quite obvious that its processing performance as a whole is far better than what is provided by any of the TACO instances analyzed in this case study. In terms of power consumption we refer to the IXP1200 datasheet [56], according to which the typical power consumption of an IXP1200 processor at 200 MHz is 4.5 W and the maximum consumption is 6.6 W. Of these values, the embedded SA-1100 equivalent on-board StrongARM core has a maximum power consumption of about 500 mW at the target clock frequency according to the SA-1100 datasheet [55]. The rest of the (maximum) power, about 6 W, is consumed by the six microengines, the internal memories and other on-chip functional blocks (e.g. Intel IXP Bus controller). Unfortunately the datasheets cited here do not reveal more detailed information on power consumption of individual functional blocks in an IXP1200 processor; therefore we can only state (based on the values given in Table 5.5 and in the cited Intel datasheets) that the power consumption of the TACO architectures analyzed in this case study can be assumed to be in the same order of magnitude as the power consumed by one IXP1200 microengine.

Also in terms of area comparisons the available information for the IXP1200 microengines is very limited. According to [77], the die size for an IXP1200 processor is 126 mm^2. On the other hand, [74] reports a die size of 75 mm^2 for an SA-1100 (including memory and interface controllers). If the memory and interface controllers on-board an IXP1200 are expected to occupy about the same area as the ones used in an SA-1100 (i.e. the die size of an SA-1100 is assumed to be about the same as the IXP1200's on-board StrongARM core plus the IXP1200's on-board memory and interface controllers), 51 mm^2 would be left for the six microengines and the on-chip IX Bus controller. Further, if we assume that the IX bus controller occupies roughly the area of four microengines (based on a physical layout image presented in [77]), we reach a coarse area estimate of 5.1 mm^2 for each microengine. This value (which can under no circumstances be deemed accurate) is higher than the area occupied by any of the TACO instances; however, according to [77], the IXP1200 is manufactured using a 0.25 μm process. The results for the TACO instances of this case study were obtained for 0.18 μm. The conclusion for the area comparisons between the TACO instances and an IXP1200 microengine thus is the same as for the power comparison: the values can be assumed to be in the same order of magnitude, especially if the same manufacturing technology had been used for both architectures.

An interesting continuation for this case study could be to implement a Network-on-Chip (NoC) [111] device that would incorporate a general purpose processor core and several TACO processors, and to compare its performance to the IXP1200 or some other member of the IXP network processor family. The first step to this direction is taken in the next section, in which a TACO IPv6 client processor is integrated onto a multimedia processing NoC platform.

5.3 IPv6 Client for a Network-on-Chip Platform

As the third case study of this thesis we examine the component reuse and Network-on-Chip (NoC) [111] integration capabilities of the TACO framework. Like the case study presented in the previous section, the target application for this case study also deals with processing the IPv6 protocol. However, the target application considered in this case study is simpler and requires a lower processing speed; hence, the design process is reduced to reusing both the hardware modules and the application code from the previous case study, and to implementing a Network-on-Chip interface for the resulting processor core. Architectural and implementational details of the NoC interface for TACO processors have already been discussed in Chapter 3, and that discussion is not repeated here.

Soon after the completion of the IPv6 router processor project the TACO research group was invited to participate in a NoC design project within a national research program in Finland. The goal in the project was to develop a NoC platform for networked multimedia processing. The platform was required to receive, decrypt, and decode a compressed video stream. Such a platform could be used for example in a PocketPC-type device for applications like confidential mobile videoconferencing. The video stream was assumed to have been compressed using the MPEG-2 [58] algorithm and encrypted using the RSA [96] algorithm prior to sending it over a wireless connection. The MPEG-2 compression parameters were chosen to meet the screen size and processing capacity of modern high-end hand-held devices with built-in wireless LAN (WLAN) support. The compression parameters are shown in Table 5.7.

The networking environment in this case study is a small office with a wireless LAN. The encrypted compressed video data is streamed from a server equipped with wireless access point functionality to hand-held devices in the WLAN segment. The network layer protocol (OSI layer 3) used in this case study is IPv6. Due to this arrangement, the server can be configured to send

Video	**Audio**	**Frame Size**	**Frame Rate**	**Bits/Pixel**
900 kbps (CBR)	64 kbps (joint stereo)	320 x 240 px	25 fps	24

Table 5.7: Compression parameters for the case study. CBR = constant bit rate.

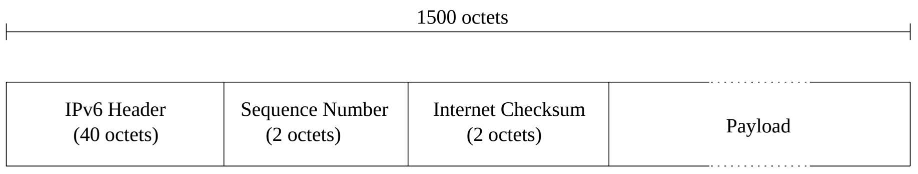


Figure 5.7: Structure of the PDU used in the case study.

all datagrams with a *hop limit* value of 1. This way the possibility of confidential IPv6 datagrams traversing outside the target operating environment is reduced: IPv6 routers discard such datagrams.

To improve the reliability of the wireless transmission, a simple proprietary transport layer (OSI layer 4) protocol was defined to be used on top of IPv6 in this case study. The proprietary protocol basically adds sequence numbering and an Internet checksum of the entire datagram to the basic IPv6 header. Also, no IPv6 extension headers (see Chapter 2) were neither needed nor used in this case study. The structure of the resulting PDU is shown in Figure 5.7. The payload of the PDU carries the RSA-encrypted MPEG-2 video stream. With 44 header octets and a total PDU size of 1500 octets, the transmission overhead produced by the transport and network layer protocols used in this case study is about 3 %.

The NoC platform was built of IP (intellectual property) blocks developed in the research units participating in the co-operation. The main components were a TACO processor core for IPv6 client operation, an RSA unit[3] [95] for decryption, and a programmable general purpose RISC processor core called COFFEE[3] for decompression. These IP blocks were interconnected using an on-chip packet-switching network called PROTEO[3] [100]. Each IP block was required to provide Virtual Component Interface (VCI) [118] compliancy towards the PROTEO network.

The NoC platform itself is presented in [3]. In this case study we focus on constructing a VCI-compliant IPv6 client core needed for the platform using the tools and techniques of the TACO framework. Analysis of the tasks needed to be performed by the IPv6 client processor as part of the NoC platform produced the following conclusions:

[3]Developed at the Tampere University of Technology, Finland.

- The data word length of 32 bits should be used, and 0.18 μm CMOS technology should be used.

- IPv6 client operation requires the following protocol processing functional unit types: Comparator, Matcher, Counter, Internet Checksum, Masker and Shifter. After the previous case study in IPv6 routing, all these functional units already existed as 0.18 μm implementations in TACO libraries (see Chapter 3 for more details on the FU types).

- Current standard WLANs operate at maximum speeds of less than 100 Mbps. With 1500-octet PDUs and a 100 Mbps peak transmission rate the target NoC must be able to receive and process at least 8 300 IPv6 datagrams per second. This is about 20 times less than the datagram processing rate required from the TACO IPv6 router architectures explored in the previous case study, which suggests that a relatively simple TACO architecture could be used for this kind of IPv6 client operation. Also, a 100 Mbps networking environment clearly provides enough data transport capacity to carry the MPEG-2 stream defined in Table 5.7.

- The IPv6 client receives incoming datagrams from an on-chip I/O buffer through a VCI compliant NoC interface. The I/O interface was described in Chapter 3.

- Each received IPv6 datagram needs to be validated using a procedure very similar to the *Validate* function in the previous case study. Specifically, the header *Version* field has to indicate IPv6, the *Next Header* value has to indicate the previously defined proprietary upper layer protocol, the *Payload Length* field must indicate a value that puts the total datagram size to that defined in Figure 5.7, and the *Hop Limit* value may not be larger than 1.

- Each received IPv6 datagram is verified for integrity by computing its checksum as defined in RFC 1071 [11].

- Once validated and verified, the header information needs to be removed from each datagram, and the payload should then be transported to the on-chip RSA decryption block through the VCI-compliant on-chip network.

In comparison to the IPv6 router case study, the IPv6 client application requires a significantly lower PDU processing speed. For this reason, a decision was made to simplify the *TACO-1* router architecture instance and to adapt its *validate* and *checksum* functions for this case study. Thus, the application specification did not need to be refined into assembler code for

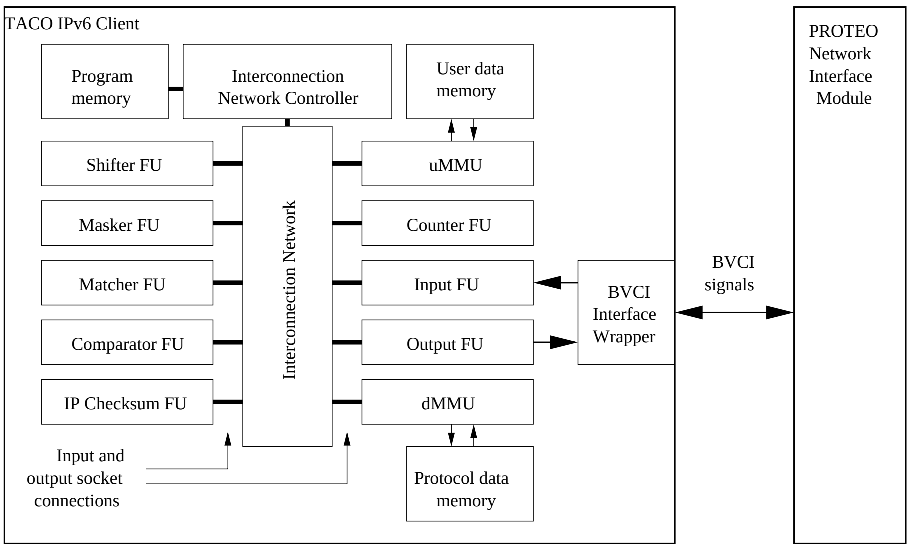


Figure 5.8: The TACO IPv6 Client and its BVCI connection to the on-chip network.

a virtual processor as usually done in TACO design projects. Instead, an architecture with only one FU of each required type and only one interconnection bus was specified. All protocol processing FUs were reused from the IPv6 router case study. However, to connect the IPv6 client to the on-chip network, a BVCI [118] compliant interface needed to be designed and implemented. The interface was implemented as a bridge between the TACO input and output FUs and the on-chip PROTEO network as described in Chapter 3. The resulting IPv6 client architecture is shown in Figure 5.8. Since the TACO NoC interface was already presented in Chapter 3, we will not discuss its details further in this section.

The application code for the IPv6 client was constructed using modified versions of the IPv6 routing function implementations from the previous case study as a starting point. Thus, previously existing modules were extensively reused for both the hardware and the software parts of the IPv6 client processor.

Top-level SystemC and Matlab files for the architecture were again generated using the TACO design tool discussed in Chapter 4. SystemC simulations suggested that the target application requires 1920 clock cycles per PDU when executed on the TACO architecture of Figure 5.8. The original application specification required the client to be able to operate in LAN speeds of up to 100 Mbps (or up to 8 300 PDUs per second). At this speed, the clock speed requirement for the TACO IPv6 client is 17 MHz. However, we decided to target the Matlab estimations and VHDL synthesis to 200 MHz, at which

	Estimated worst-case power [mW]	Estimated logic area [μm^2]	Synthesized logic area [μm^2]
Matcher	7.1	13 785	12 851
Shifter	14.6	28 160	24 351
Comparator	9.3	18 036	17 197
Masker	6.4	12 346	11 335
Checksum	10.5	20 324	19 836
Counter	6.8	13 161	11 782
Input FU	15.9	30 820	26 064
Output FU	9.4	18 207	17 002
dMMU	6.1	11 844	11 721
uMMU	4.0	7701	7342
Sockets	21.6	41 122	31 720
Network Controller	7.0	28 346	21 169
Total (IPv6 client part)	118.7	243 852	212 370
BVCI Interface wrapper	15.6	30 342	23 137
Total (client + wrapper)	134.3	274 194	235 507

Table 5.8: Estimation and synthesis results in the IPv6 Client case study. Estimations and synthesis have been targeted to 200 MHz clock frequency. The results do not include memory blocks.

speed the TACO client should be able to provide adequate performance for operation at network speeds of up to 1 Gbps.

Results of the Matlab estimations and VHDL synthesis for 0.18 μm CMOS technology and 200 MHz target clock speed are shown in Table 5.8. The Matlab estimates suggested a logic area of 0.27 mm^2 (core size without memories) and a worst case power consumption of 134 mW. These values were acceptable for the target multimedia processing NoC platform, and thus we proceeded to synthesize the architecture. Synthesis resulted in a logic area of 0.24 mm^2. When comparing this to the estimated value, it can be stated that the accuracy of the area estimates was $A_{synth} = 0.86\ A_{estim}$. This accuracy is sufficient at the system level, where slight pessimism in estimates is always preferable: if the estimations would suggest a smaller area than what is obtainable through synthesis, original design constraints might not be met. It is important to notice that Matlab area estimates for are consistently slightly larger than the values obtained through synthesis for every module in the processor, hence ensuring that design constraints are always met also after synthesis. The power consumption estimate reflects the situation of operating at 200 MHz (i.e. at about 1 Gbps network speed). The network speed for the target application is considerably lower, which reduces the power consumption respectively. Also, the estimate expects all functional units to be active at all times, which is obviously a worst-case scenario as explained in the previous case study (the application software and the complexity of the interconnection network determine FU activity).

The results presented in Table 5.8 revealed that the BVCI wrapper module did not considerably increase the size or power consumption of the IPv6 client processor. In fact, the increase in both size and power is about the same as the cost of adding one functional unit into the architecture. For the IPv6 client processor discussed above, the synthesized area increases by 11 % and the estimated average power consumption by 13 %. Naturally, the relative increase in area and power consumption produced by the BVCI wrapper is reduced in more complex applications in which more functional units and interconnection network buses are used.

The design of the TACO IPv6 client did not involve any modifications to the TACO SystemC, Matlab and VHDL protocol processing FU descriptions; instead, the work was done by choosing which components from the component library are needed using the TACO design tool. Then the tool was used to generate top level instantiation files for simulation, estimation and synthesis. Naturally, since no new protocol processing functional units were needed into the libraries for the target application, and most of the application code was taken from earlier work, the IPv6 client design project was carried out in a very short period of time (a matter of days). More time was needed for specifying and implementing the BVCI interface wrapper. However, after this case study the BVCI interface can now also be incorporated into any new TACO processor design project as a library component, thus enabling convenient Network-on-Chip integration in future design projects.

Clearly, the component reuse capabilities of the TACO framework were evident and proved to be quite useful in designing the IPv6 client processor. A silicon implementation of the target multimedia processing NoC platform (nowadays called CoffeeTable) with its TACO IPv6 client core is currently under development as a continuation to the research program that originated the NoC project. At the time of writing this thesis, the project timetable expects the chip to be manufactured in 2005. At that time, the chip will be extremely valuable in further performance studies of TACO processors as well as in examining TACO use as an intellectual property block in Network-on-Chip devices.

5.4 Chapter Summary

This chapter discussed three case studies in which the tools and techniques of the TACO framework were applied to three different protocol processing applications used in different kinds of networking environments and conditions. The goal in the case studies was to show that the assumptions and arguments made in Chapter 4 were valid.

The first case study, ATM AIS processing, demonstrated the rapid designer driven system level design space exploration capabilities of the TACO framework and verified the accuracy of system-level simulations and estimations. The second case study, IPv6 routing/packet forwarding, focused on determining how well the protocol processing performance of TACO processors scales up to meet the performance provided by typical processing elements found in modern commercial network multiprocessor architectures. As a result, TACO architectures were seen to provide similar execution times for IPv6 routing functions as an Intel IXP1200 microengine. Interestingly, the TACO architectures considered in the case study were found to be considerably faster than an IXP1200 microengine in calculating the Internet checksum. The third case study, IPv6 client operation, demonstrated the component reuse and Network-on-Chip integration capabilities in the TACO framework. A processor architecture for IPv6 client operation was constructed entirely from pre-existing protocol processing functional units, and most of the application code was also obtained from the earlier IPv6 router case study. The processing speed required for the IPv6 client application was roughly 20 times less than the speed required in the IPv6 router case study. Also, the IPv6 client application was considerably simpler than the routing application. Thus, no design space exploration was necessary: it was enough to simplify the most basic one of the IPv6 router architectures analyzed in the second case study. Finally, in the third case study it was seen that a TACO processor can rather conveniently be integrated into a Network-on-Chip platform by using an industry-standard interface. It was also seen that the area and power costs of including the NoC interface wrapper into a TACO architecture are about the same as the costs from including an average functional unit. We consider these costs to be quite reasonable for gaining NoC support to any given TACO architecture. A silicon implementation of the NoC platform is currently under development, and the chip is expected to be manufactured in 2005. The chip will provide a valuable testing and analysis environment for further performance studies of TACO processors and their use as intellectual property blocks in Network-on-Chip devices. Also, we are currently working towards an FPGA implementation of a more complex TACO architecture with NoC support. The FPGA implementation is expected to be ready during fall 2004.

We conclude this chapter by stating that the three case studies reached the goals that had been set for them. Thus, they validated the fundamental assumptions and arguments on which the TACO framework and its design process are based on. The next chapter summarizes and concludes this thesis.

Chapter 6

Conclusion

In this thesis we have proposed a framework for rapidly designing and evaluating programmable protocol processors with application-optimized hardware execution units. The proposed framework, called the *TACO framework*, is built of a hardware platform optimized for protocol processing, and a documented design methodology for rapidly specifying, simulating, evaluating and synthesizing protocol processors based on the hardware platform. Using the tools and methods provided by the TACO framework an optimized hardware architecture, optimized application code and a synthesizable processor model are reached using an application specification as a starting point. In the introduction of this thesis we described the potential benefits to be gained by developing protocol processor architectures and domain-specific design methodologies, and postulated the key research issues that need to be addressed in order to create a design framework for these purposes. In this chapter we summarize the solutions to these issues that have been proposed in earlier chapters of this thesis, and outline directions for future work within the TACO research project.

We started unfolding the research issues by arguing that different protocols require similar operations in their processing, and thus implementing hardware support for such operations is beneficial in terms of processing performance. To support this argument we analyzed six commonly and abundantly used communication protocols to find common characteristic functionality in their processing. The findings made in this analysis clearly showed foreseeable advantages to be gained by designing processors with optimized hardware for protocol processing tasks. From these findings we proceeded to specifying a hardware platform with native support for protocol processing operations. The TTA-based hardware platform, which we called the *TACO architecture*, is a modular and scalable family of programmable architectures optimized for protocol processing. We argued that the TACO architecture can be used for automated component library based hardware design. To our knowledge, the

TACO hardware platform is the first and so far the only approach in which the TTA paradigm is applied to protocol processing. Similarly, to our knowledge the memory organization and access scheme of the TACO hardware platform is unique in comparison to existing TTA implementations.

After specifying the hardware platform we proceeded to developing a rapid protocol processor design methodology around it. The work required desiging and implementing processor models for simulation, estimation and synthesis of TACO architectures. A key issue in the development of the processor models was that they needed to provide a reasonably simple application programming interface through which models for any given TACO architecture could be generated. This was deemed especially important for system level simulations of architectures: in addition to functional verification, simulations need to provide hardware-accurate results like cycle-by-cycle simulation and register transfer statistics. Whilst providing such precise information, the simulations were also required to execute very fast to facilitate rapid architectural exploration at the system level. The TACO SystemC simulation model was found to meet these requirements through use of object oriented programming techniques and a heterogeneous level of abstraction in the implementation; it was seen that typically a TACO simulator is able to simulate thousands of clock cycles per second on a standard PC.

For the Matlab system-level estimation model the speed requirements did not cause potential performance bottlenecks since the estimations are carried out as one-time calculations of pre-defined equations. The synthesis model on the other hand is relieved of such performance requirements since it is put to use only after a suitable architecture has been found through system-level simulations and estimations; the most important requirements for the synthesis model were that its API supports generating a synthesizable description of a given TACO architecture, and that the generated description can be reliably synthesized.

In addition to recognizing common characteristics in the processing of different protocols, we argued that attention also needs to be paid to identification of frequently needed operations within the particular target application. We defined a sequence of object oriented UML-based analysis and refinement methods for such application analysis. Finally, using all the building blocks outlined above, we constructed a design flow for rapidly and iteratively finding optimized combinations of hardware architecture and application code for a given target application. The most important characteristics of the entire design methodology were identified to be completeness, capability for system-level iteration, exploration and evaluation, short turn-around time, reliability of the simulation, estimation and synthesis results, and cost-efficiency.

To validate the assumptions and arguments made in this thesis, we applied the tools and methods of the TACO framework to three case studies in protocol processor design. The first case study verified the rapid designer driven system level design space exploration capabilities of the TACO framework and the accuracy of system-level simulations and estimations. The case study concentrated on exploring TACO architectures for processing a part of the ATM AIS (Alarm Indication Signal) function in a 622 Mbps ATM network, and was targeted at the 0.35 μm technology generation. Six different TACO architectures were explored. Of these, the fastest one was able to execute the target application at 53 MHz while the slowest one required a clock frequency of 178 MHz.

The second case study was targeted at the 0.18 μm technology generation and focused on determining how well the protocol processing performance of TACO processors scales up to meet the performance provided by typical processing elements found in modern commercial network multiprocessor architectures. As a result, the analyzed TACO architectures were seen to provide slightly better execution times for key IPv6 routing functions than an Intel IXP1200 microengine running at the same clock frequency. In Internet checksum calculation the TACO implementations were actually seen to be up to 20 times faster than the corresponding IXP1200 microengine implementation. A key factor for such a difference was determined to be the full hardware implementation in TACO architectures as compared to a full software implementation on the IXP1200. We also concluded that the less than 200 mW worst-case power consumption and the less than 1 mm^2 area of the analyzed TACO architectures can be expected to be in the same order of magnitude as the corresponding characteristics of an IXP1200 microengine, especially if the same manufacturing technology would have been used on both architectures.

The third case study verified the component reuse and Network-on-Chip integration capabilities of the TACO framework: an IPv6 client core was successfully constructed solely of previously existing TACO hardware and software components. It was also seen that the area and power costs of including support for Network-on-Chip connectivity into a TACO architecture are about the same as the costs from including an average functional unit (a 16 mW increase in power consumption and a 0.03 mm^2 increase in area in the 0.18 μm technology generation). A silicon implementation of the NoC platform for which the IPv6 client core was designed is currently under development, and the chip is expected to be manufactured in 2005. The chip will be extremely valuable as a test environment for further performance studies of TACO processors and their use as intellectual property blocks in Network-on-Chip devices. Also, we are currently working towards an FPGA implementation of a more complex TACO architecture with NoC support. The FPGA implementation is expected to be ready during fall 2004. All in all, the three case studies reached the goals that had been set for them and thus validated the fundamental assumptions and arguments on which the TACO framework and its building blocks are based on.

There are still several tasks in the TACO design process that could be automated further. For example, the application analysis procedure described in this thesis is performed manually by the designer. As this is the case, automated methods for application analysis are currently investigated in another line of research within the TACO project. These methods are aimed at relieving the designer from manually refining application specifications, and at automatically finding domain-specific operations needed by the application. A related topic is designing a compiler that would automatically compile and optimize program code written in a high-level language to a given TACO architecture. The current TACO compiler operates on assembler and requires the designer to manually optimize the code for the target architecture. Recent advances in TACO application analysis research suggest that it may be possible to incorporate optimizing compiler functionality into the automatized application analysis process. However, substantially more research is still needed in this area of TACO research.

An emerging research direction in the TACO project is mapping the TACO framework to other application domains. Our current focus in this direction is video compression. This can in many senses be seen as a related, although mathematically more intensive, application domain. In terms of the TACO architecture, the future focus will be on optimizing the Input/Output interface further and in developing alternate solutions for the current basic TACO I/O interface. Finally, we intend to further improve the simulation and estimation models to provide even more detailed results, especially in terms of power consumption analysis. Among other things, this involves gathering and analyzing very detailed information on the characteristics of the data being moved on the interconnection buses (e.g. quantity and position information of one and zero bits in each transferred data word during a simulation).

Bibliography

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, August 2002. http://developer.intel.com/technology/itj/2002/volume06issue03/ (verified 2004-08-24).

[2] T. Ahonen, T. Nurmi, J. Nurmi, and J. Isoaho. Block-wise extraction of Rent's exponents for an extensible processor. In *IEEE Computer Society Annual Symposium on VLSI*, pages 193–199, February 2003.

[3] T. Ahonen, S. Virtanen, J. Kylliäinen, D. Truscan, T. Kasanko, D. Sigüenza-Tortosa, T. Ristimäki, J. Paakkulainen, T. Nurmi, I. Saastamoinen, H. Isännäinen, J. Lilius, J. Nurmi, and J. Isoaho. *A Brunch from the Coffee Table - Case Study in NoC Platform Design. Chapter 16 in J. Nurmi, H. Tenhunen, J. Isoaho and A. Jantsch (Eds): Interconnect-Centric Design for Advanced SoC and NoC.* Kluwer Academic Publishers, Boston, MA, U.S.A., April 2004.

[4] M. Alanen, J. Lilius, I. Porres, and D. Truscan. Realizing a model driven engineering process. Technical Report 565, Turku Centre for Computer Science, Turku, Finland, November 2003.

[5] J. Allen, B. Bass, C. Basso, R. Boivie, J. Calvignac, G. Davis, L. Freléchoux, M. Heddes, A. Herkersdorf, A. Kind, J. Logan, M. Peyravian, M. Rinaldi, R. Sabhikhi, M. Siegel, and M. Waldvogel. IBM PowerNP network processor: Hardware, software and applications. *IBM Journal of Research and Development*, 47(2/3):177–194, March 2003. http://www.research.ibm.com/journal/rd47-23.html (verified 2004-08-24).

[6] American National Standardization Institute. *ANSI T1.105-2001, Synchronous Optical Network (SONET) - Basic Description including Multiplex Structure, Rates and Formats*, 2001.

[7] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software*

Codesign (CODES'01), pages 61–66, Copenhagen, Denmark, April 2001.

[8] M. Attia and I. Verbauwhede. Programmable Gigabit Ethernet packet processor design methodology. In *Proceedings of the European Conference on Circuit Theory and Design (ECCTD'01)*, pages III:177–180, Espoo, Finland, August 2001.

[9] H. B. Bakoglu. *Circuits, Interconnections and Packaging for VLSI*. Addison-Wesley Publishing Company, Inc., Reading, MA, U.S.A., 1990.

[10] A. Both, B. Biermann, R. Lerch, Y. Manoli, and K. Sievert. Hardware-software-codesign of application specific microcontrollers with the ASM environment. In *Proceedings of the Conference on European Design Automation*, pages 72–76, Grenoble, France, September 1994.

[11] R. Braden. Computing the Internet checksum. *RFC 1071*, Sept. 1988.

[12] S. Bradner and A. Mankin. IP: Next generation (IPng) white paper solicitation. *RFC 1550*, December 1993.

[13] C-Port Corporation (a Motorola Company), North Andover, MA, U.S.A. *C-5 Network Processor Architecture Guide*, May 2001. http://www.freescale.com/files/netcomm/doc/ref_manual/C5NPD0-AG.pdf (verified 2004-08-24).

[14] A. Caldwell, Y. Cao, A. B. Kahng, F. Koushanfar, H. Lu, I. L. Markov, M. Oliver, D. Stroobandt, and D. Sylvester. GTX: The MARCO GSRC technology extrapolation system. In *Proceedings of the 37th IEEE/ACM Design Automation Conference*, pages 693–698, Los Angeles, CA, U.S.A., June 2000.

[15] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[16] S. M. Cherry. Wi-Fi takes new turn with "Wireless-G". *IEEE Spectrum*, 40(8):12–13, August 2003.

[17] G. Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis. *Synchronous Transfer Architecture (STA)*, pages 343–352. LNCS 3133. Springer-Verlag, Berlin, Germany, 2004.

[18] R. Clauberg, P. Buchmann, A. Herkersdorf, and D. J. Webb. Design methodology for a large communication chip. *IEEE Design and Test of Computers*, pages 86–94, July-September 2000.

[19] D. E. Comer. *Network Systems Design Using Network Processors*. Pearson Prentice Hall, Upper Saddle River, NJ, U.S.A., 2004.

[20] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. John Wiley and Sons Ltd., Chichester, West Sussex, England, 1998.

[21] H. Corporaal and J. Hoogerbrugge. Cosynthesis with the MOVE framework. In *Proceedings of the IMACS-IEEE Multiconference on Computational Engineering in Systems Applications (CESA'96)*, pages 184–189, Lille, France, July 1996.

[22] M. Crawford. Transmission of IPv6 packets over Ethernet networks. *RFC 2464*, December 1998.

[23] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, editors. *Network Processor Design - Issues and Practices*, volume 1. Morgan Kauffmann Publishers, San Francisco, CA, U.S.A., 2003.

[24] The Data Display Debugger (DDD) web site.
http://www.gnu.org/software/ddd/ (verified 2004-08-24).

[25] J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71:1334–1340, December 1983.

[26] M. Decina and V. Trecordi. Convergence of telecommunications and computing to networking models for integrated services and applications. *Proceedings of the IEEE*, 85(12):1887–1914, December 1997.

[27] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. *RFC 1883*, December 1995.

[28] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. *RFC 2460*, December 1998.

[29] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 2000.

[30] The Easics on-line CRC Tool.
http://www.easics.com/webtools/crctool/ (verified 2004-08-24).

[31] J. C. Eble, V. K. De, and J. D. Meindl. A first generation generic system simulator (GENESYS) and its relation to the NTRS. In *Proceedings of the 11th Biennial University/Government/Industry Microelectronics Symposium*, pages 147–154, Austin, TX, U.S.A., May 1995.

[32] The *MOVE* project web site.
http://ce.et.tudelft.nl/MOVE/ (verified 2004-08-24).

[33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616*, June 1999.

[34] V. Fuller, T. Li, and K. Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. *RFC 1519*, Sept. 1993.

[35] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice-Hall, Inc., Englewood Cliffs, NJ, U.S.A., 1994.

[36] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and Z. Shuqing. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, Norwell, MA, U.S.A., 2000.

[37] J. Gerlach and W. Rosenstiel. System level design using the SystemC modeling platform. In *Proceedings of the 3rd Workshop on System Design Automation (SDA2000)*, Rathen, Germany, March 2000.

[38] B. Geuskens and K. Rose. *Modeling microprocessor performance.* Kluwer Academic Publishers, Boston, MA, U.S.A., 1998.

[39] E. Grimpe and F. Oppenheimer. Aspects of object-oriented hardware modelling with SystemCPlus. In *Proceedings of the 2001 Forum on Design Languages (FDL'01)*, Lyon, France, September 2001.

[40] T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. Processor evaluation in an embedded systems design environment. In *Proceedings of the 13th International Conferenence on VLSI Design*, pages 98–103, January 2000.

[41] J. Heikkinen, T. Rantanen, A. Cilio, J. Takala, and H. Corporaal. Evaluating template-based instruction compression on transport triggered architectures. In *Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, pages 192–195, Calgary, Canada, June 2003.

[42] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: a VLSI processor architecture. Technical Report 223, Stanford University, Computer Systems Laboratory, Stanford, CA, U.S.A., November 1981.

[43] T. Henriksson. *Hardware Architecture for Protocol Processing.* Licentiate thesis, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 2001.

[44] T. Henriksson. *Intra-Packet Data-Flow Protocol Processor*. PhD thesis, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 2003.

[45] T. Henriksson, U. Nordqvist, and D. Liu. Specification of a configurable general-purpose protocol processor. In *Proceedings of Second International Symposium on Communication Systems, Networks and Digital Signal Processing*, pages 284–289, Bournemouth, UK, July 2000.

[46] R. Hinden. Simple Internet Protocol Plus white paper. *RFC 1710*, October 1994.

[47] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 20(11):1338–1354, November 2001.

[48] C. Hornig. A standard for the transmission of IP datagrams over Ethernet networks. *RFC 894*, April 1985.

[49] Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A. *RFC 760: DOD Standard Internet protocol*, January 1980.

[50] Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A. *RFC 761: DOD Standard Transmission Control Protocol*, January 1980.

[51] Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A. *RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification*, September 1981.

[52] The Institute of Electrical and Electronics Engineers, Inc., New York, NY, U.S.A. *IEEE Std 802.2, 1998 Edition. Logical Link Control*, 1998.

[53] The Institute of Electrical and Electronics Engineers, Inc., New York, NY, U.S.A. *IEEE Std 802.11, 1999 Edition. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[54] The Institute of Electrical and Electronics Engineers, Inc., New York, NY, U.S.A. *IEEE Std 802.3-2002. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, 2002.

[55] Intel Corporation. *Intel StrongARM SA-1100 Microprocessor Specification Update (order number 278105-006)*, November 1998, U.S.A.

[56] Intel Corporation. *Intel IXP1200 Network Processor Datasheet (part number 278298-010)*, December 2001, U.S.A.

[57] Intel Corporation. *Intel IXP1200 Network Processor Family Hardware Reference Manual (part number 278303-008)*, August 2001, U.S.A.

[58] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 13818 family of standards for coding of moving pictures*, 1996-2000.
http://www.iso.org/iso/en/prods-services/popstds/mpeg.html (verified 2004-08-24).

[59] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation I.361: B-ISDN ATM Layer Specification*, 1993.

[60] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation I.610: B-ISDN Operation and Maintenance Principles and Functions*, 1993.

[61] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation G.708: Sub STM-0 network node interface for the synchronous digital hierarchy (SDH)*, 1999.

[62] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation G.707/Y.1322: Network node interface for the synchronous digital hierarchy (SDH)*, 2000.

[63] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation G.709: Interfaces for the Optical Transport Network OTN*, 2003.

[64] The International Telegraph and Telephone Consultative Committee. *CCITT Recommendation I.321: B-ISDN Protocol Reference Model and its Application*, 1991.

[65] Y. I. Ismail and E. G. Friedman. Effects of inductance on the propagation delay and repeater insertion in VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):195–206, April 2000.

[66] M. K. Jain, M. Balakrishnan, and A. Kumar. ASIP design methdologies: Survey and issues. In *Proceedings of the 14th International Conference on VLSI Design*, pages 76–81, Bangalore, India, January 2001.

[67] A. Jantsch, J. Öberg, and A. Hemani. Is there a niche for a general protocol processor core? In *Proceedings of the 16th IEEE Norchip Conference*, pages 93–100, Lund, Sweden, November 1998.

[68] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. A. Vissers. *A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach*, pages 18–37. LNCS 2268. Springer-Verlag, Berlin, Germany, 2002.

[69] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C20(12):1469–1479, December 1971.

[70] B. Leiner, R. Cole, J. Postel, and D. Mills. The DARPA Internet protocol suite. *IEEE Communications Magazine*, 23(3):29–34, March 1985.

[71] S. Y. Liao. Towards a new standard for system-level design. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, U.S.A., May 2000.

[72] J. Lilius and D. Truscan. UML-driven TTA-based protocol processor design. In *Proceedings of the 2002 Forum for Design and Specification Languages (FDL'02)*, Marseille, France, September 2002.

[73] J. Lilius, D. Truscan, and S. Virtanen. Fast Evaluation of Protocol Processing Architectures for IPv6 Routing. In *Proceedings of the 2003 Design, Automation and Test in Europe conference (DATE'03)*, Munich, Gemany, March 2003.

[74] T. Litch and J. Slaton. StrongARMing portable communications. *IEEE Micro*, 18(2):48–55, March 1998.

[75] Y. Ma, A. Jantsch, and H. Tenhunen. A programmable protocol processor architecture for high speed Internet protocol processing. In *Proceedings of the 18th IEEE Norchip Conference*, pages 212–216, Turku, Finland, November 2000.

[76] G. Malkin and R. Minnear. RIPng for IPv6. *RFC 2080*, January 1997.

[77] W. Mangione and G. Memik. Network processor technologies. http://cares.icsl.ucla.edu/cares/content/presentations/NPU_overview-UCRIVERSIDEandMINDSPEED_files/presentation.pdf (verified 2004-09-02).

[78] R. Metcalfe and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(5):395–404, July 1976.

[79] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison Wesley Longman, Inc., Reading, MA, U.S.A., 2nd edition, 1997.

[80] U. Nordqvist. *A Programmable Network Interface Accelerator.* Licentiate thesis, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 2002.

[81] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen. Physical modeling and system level performance characterization of a protocol processor architecture. In *Proceedings of the 18th IEEE Norchip Conference*, pages 294–301, Turku, Finland, November 2000.

[82] M. O'Connor and C. A. Gomez. The iFlow address processor. *IEEE Micro*, pages 16–23, March-April 2001.

[83] The ODETTE Project web site. http://odette.offis.de/ (verified 2004-08-24).

[84] R. O. Onvural. *Asynchronous Transfer Mode Networks: Performance Issues.* Artech House, Inc., Norwood, MA, U.S.A., 1994.

[85] The Open SystemC Initiative web site. http://www.systemc.org/ (verified 2004-08-24).

[86] D. A. Patterson. RISC watch. *ACM SIGARCH Computer Architecture News*, 12:11–19, March 1984.

[87] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28:8–21, January 1985.

[88] J. B. Postel. Simple mail transfer protocol. *RFC 821*, August 1982.

[89] J. V. Praet, G. Goossens, D. Lanneer, and H. D. Man. Instruction set definition and instruction selection for ASIP. In *Proceedings of the Seventh International Symposium on High-Level Synthesis*, pages 11–16, Niagara-on-the-lake, Canada, May 1994.

[90] D. J. d. S. Price. An ancient Greek computer. *Scientific American*, 200(6):60–67, June 1959.

[91] J. M. Rabaey, A. Chandrakasan, and B. Nikolić. *Digital Integrated Circuits - a Design Perspective.* Prentice Hall/Pearson Education International, Upper Saddle River, NJ, U.S.A., second edition, 2003.

[92] M. Radetzki and W. Nebel. Synthesizing hardware from object-oriented descriptions. In *Proceedings of the 2nd Forum on Design Languages (FDL'99)*, Lyon, France, August 1999.

[93] G. Radin. The 801 minicomputer. *IBM Journal of Research and Development*, 27(3):237–246, May 1983. http://www.research.ibm.com/journal /rd/273/ibmrd2703E.pdf (verified 2004-08-24).

[94] A. Rijsinghani. Computation of the Internet checksum via incremental update. *RFC 1624*, May 1994.

[95] T. Ristimäki and J. Nurmi. Implementation of a fast 1024-bit RSA encryption on platform FPGA. In *Proceedings of the 6th IEEE International Workshop on Design and Diagnostics of Electronics Circuits and Systems (DDECS'03)*, Poznan, Poland, April 2003.

[96] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[97] G. Schumacher and W. Nebel. Inheritance concept for signals in object-oriented extensions to VHDL. In *Proceedings of the EURO-DAC'95 Design Automation Conference with EURO-VHDL'95*, pages 428–435, Brighton, UK, September 1995.

[98] G. Schumacher and W. Nebel. Object-oriented hardware modelling – where to apply and what are the objects? In *Proceedings of the EURO-DAC'96 Design Automation Conference with EURO-VHDL'96*, pages 428–433, Geneva, Switzerland, September 1996.

[99] R. S. Shelar, S. Nath, and J. S. Nanaware. Parameterized reusable component library methodology. In *Proceedings of the 26th EUROMICRO Conference (EUROMICRO'00)*, Maastricht, The Netherlands, September 2000.

[100] D. Sigüenza-Tortosa and J. Nurmi. PROTEO: A new approach to Network-on-Chip. In *Proceedings of the IASTED International Conference on Communication Systems and Networks (CSN'02)*, Malaga, Spain, September 2002.

[101] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures - a Design Space Approach.* Addison-Wesley Longman Ltd., Harlow, Essex, England, 1998.

[102] R. J. Smith and M. Gibbs. *Navigating the Internet.* SAMS Publishing, Indianapolis, IN, U.S.A., 1994.

[103] P. Srisuresh. Traditional IP network address translator (traditional NAT). *RFC 3022*, January 2001.

[104] W. Stallings. *Computer Organization and Architecture - Designing for Performance.* Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., international edition, 1996.

[105] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, U.S.A., 3rd edition, 1997.

[106] D. Sylvester and K. Keutzer. System-level performance modeling with BACPAC - Berkeley Advanced Chip Performance Calculator. In *SLIP'99 - Workshop on System-Level Interconnect Prediction*, pages 109–114, April 1999.

[107] D. Tabak and G. J. Lipovski. MOVE architecture in digital controllers. *IEEE Transactions on Computers*, 29(2):180–190, February 1980.

[108] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., Upper Saddle River, NJ, U.S.A., third edition, 1996.

[109] A. S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., fourth edition, 1999.

[110] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., Upper Saddle River, NJ, U.S.A., fourth edition, 2003.

[111] H. Tenhunen and A. Jantsch, editors. *Networks on Chip*. Kluwer Academic Publishers, Dordrecht, Netherlands, 2003.

[112] L. Torvalds. *Open Sources: Voices from the Open Source Revolution*, chapter "The Linux Edge". O'Reilly and Associates, Inc., Sebastopol, CA, U.S.A., 1999.

[113] D. Truscan, J. M. Fernandes, and J. Lilius. Tool support for DFD-UML model-based transformations. In *Proceedings of the 11th International Conference and Workshop on the Engineering Of Computer-Based Systems (ECBS'04)*, Brno, Czech Republic, May 2004.

[114] J. Van Praet, D. Lanneer, W. Geurts, and G. Goossens. Processor modeling and code selection for retargetable compilation. *ACM Transactions on Design Automation of Electronic Systems*, 6(3):277–307, July 2001.

[115] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Computer Society Workshop on VLSI'99*, Orlando, FL, U.S.A., April 1999.

[116] S. Virtanen, J. Lilius, and T. Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proceedings of the 18th IEEE Norchip Conference*, pages 204–211, Turku, Finland, November 2000.

[117] S. Virtanen, T. Nurmi, J. Paakkulainen, and J. Lilius. A system-level framework for designing and evaluating protocol processor architectures. *International Journal of Embedded Systems (Special Issue on Hardware-Software Codesign for SoC)*, 1(1), 2004 (in press).

[118] VSI Alliance. *Virtual Component Interface Standard.* VSIA, April 2001.

[119] Xelerator X10q Network Processors: Product Brief. http://www.xelerated.com/file.aspx?file_id=3 (verified 2004-09-06).

[120] Z. Yang. IPv6 router design and routing functions implementation on IXP1200 network processor. Master's thesis, University of Turku, Finland, August 2002.

[121] L. R. Zheng, B. Li, and H. Tenhunen. Global interconnect design for high speed ULSI and system-on-package. In *Proceedings of the 12th Annual IEEE ASIC/SOC conference (ASIC/SOC'99)*, Washington DC, U.S.A., September 1999.

ISBN 952-12-1410-4
ISSN 1239-1883