Embedded System Design

By
Peter Marwedel

Springer

EMBEDDED SYSTEM DESIGN

Embedded System Design

by

PETER MARWEDEL
University of Dortmund, Germany

Springer

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 1-4020-7690-8 (HB)
ISBN-13 978-1-4020-7690-9 (HB)
ISBN-10 0-387-29237-3 (PB)
ISBN-13 978-0-387-29237-3 (PB)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springeronline.com

Printed on acid-free paper

Printed in the Netherlands.

This book is dedicated
to my family.

Contents

Preface

Importance of embedded systems

Embedded systems can be defined as information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment. Such systems come with a large number of common characteristics, including real-time constraints, and dependability as well as efficiency requirements. Embedded system technology is essential for providing ubiquitous information, one of the key goals of modern information technology (IT).

Following the success of IT for office and workflow applications, embedded systems are considered to be **the** most important application area of information technology during the coming years. Due to this expectation, the term **post-PC era** was coined. This term denotes the fact that in the future, standard-PCs will be a less dominant kind of hardware. Processors and software will be used in much smaller systems and will in many cases even be invisible (this led to the term **the disappearing computer**). It is obvious that many technical products have to be technologically advanced to find customers' interest. Cars, cameras, TV sets, mobile phones etc. can hardly be sold any more unless they come with smart software. The number of processors in embedded systems already exceeds the number of processors in PCs, and this trend is expected to continue. According to forecasts, the size of embedded software will also increase at a large rate. Another kind of Moore's law was predicted: *For many products in the area of consumer electronics the amount of code is doubling every two years* [Vaandrager, 1998].

This importance of embedded systems is so far not well reflected in many of the current curricula. This book is intended as an aid for changing this situation. It provides the material for a first course on embedded systems, but can also be used by non-student readers.

Audience for this book

This book intended for the following audience:

- Computer science, computer engineering and electrical engineering students who would like to specialize in embedded systems. The book should be appropriate for third year students who do have a basic knowledge of computer hardware and software. This book is intended to pave the way for more advanced topics that should be covered in a follow-up course.
- Engineers who have so far worked on systems hardware and who have to move more towards software of embedded systems. This book should provide enough background to understand the relevant technical publications.
- Professors designing a new curriculum for embedded systems.

Curriculum integration of embedded systems

The book assumes a basic understanding in the following areas (see fig. 0.1):

- electrical networks at the high-school level (e.g. Kirchhoff's laws),
- operational amplifiers (optional),
- computer hardware, for example at the level of the introductory book by J.L. Hennessy and D.A. Patterson [Hennessy and Patterson, 1995],
- fundamental digital circuits such as gates and registers,
- computer programming,
- finite state machines,
- fundamental mathematical concepts such as tuples, integrals, and linear equations,
- algorithms (graph algorithms and optimization algorithms such as branch and bound),
- the concept of NP-completeness.

A key goal of this book is to provide an overview of embedded system design and to relate the most important topics in embedded system design to each other. It should help to motivate students and teachers to look at more details. While the book covers a number of topics in detail, others are covered only briefly. These brief sections have been included in order to put a number of

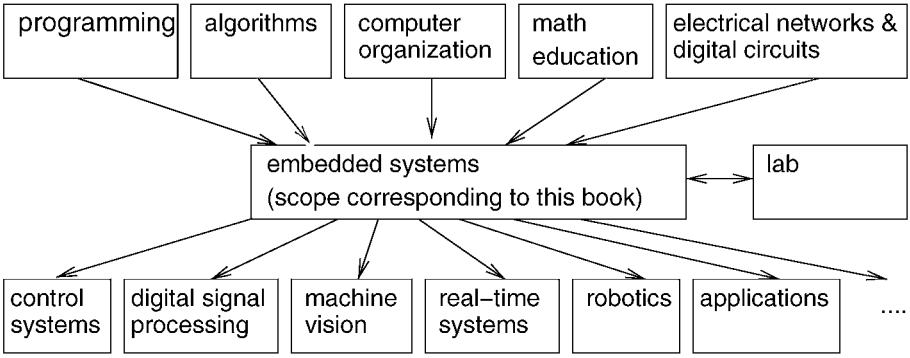

Figure 0.1. Positioning of the topics of this book

related issues into perspective. Furthermore, this approach allows lecturers to have appropriate links in the book for adding complementary material of their choice. The book should be complemented by follow-up courses providing a more specialized knowledge in some of the following areas:

- digital signal processing,
- robotics,
- machine vision,
- sensors and actors,
- real-time systems, real-time operating systems, and scheduling,
- control systems,
- specification languages for embedded systems,
- computer-aided design tools for application-specific hardware,
- formal verification of hardware systems,
- testing of hardware and software systems,
- performance evaluation of computer systems,
- low-power design techniques,
- security and dependability of computer systems,
- ubiquitous computing,
- application areas such as telecom, automotive, medical equipment, and smart homes,

- impact of embedded systems.

A course using this book should be complemented by an exiting lab, using, for example, small robots, such as Lego Mindstorm^{TM} or similar robots. Another option is to let students gain some practical experience with StateCharts-based tools.

Additional information related to the book can be obtained from the following web page:

http://ls12-www.cs.uni-dortmund.de/~marwedel/kluwer-es-book.

This page includes links to slides, exercises, hints for running labs, references to selected recent publications and error corrections. Readers who discover errors or who would like to make comments on how to improve the book should send an e-mail to peter.marwedel@udo.edu.

Assignments could also use the information in complementary books [Ganssle, 1992], [Ball, 1996], [Ball, 1998], [Barr, 1999], [Ganssle, 2000], [Wolf, 2001], [Buttazzo, 2002].

The use of names in this book without any reference to copyrights or trademark rights does not imply that these names are not protected by these.

Please enjoy reading the book!

Dortmund (Germany), September 2003

P. Marwedel

Welcome to the current updated version of this book! The merger of Kluwer and Springer publishers makes it possible to publish this version of the book less than two years after the initial 2003 version. In the current version, all typos and errors found in the original version have been corrected. Moreover, all Internet references have been checked and updated. Apart from these changes, the content of the book has not been modified. A list of the errors corrected is available at the web page listed above.

Please enjoy reading this updated book.

Dortmund (Germany), August 2005

P. Marwedel

Acknowledgments

My PhD students, in particular Lars Wehmeyer, did an excellent job in proofreading a preliminary version of this book. Also, the students attending my course "Introduction to Embedded Systems" of the summer of 2003 (in particular Lars Bensmann) provided valuable help. In addition, the following colleagues and students gave comments or hints which were incorporated into this book: W. Müller, F. Rammig (U. Paderborn), W. Rosenstiel (U. Tübingen), R. Dömer (UC Irvine), and W. Kluge (U. Kiel). Material from the following persons was used to prepare this book: G. C. Buttazzo, D. Gajski, R. Gupta, J. P. Hayes, H. Kopetz, R. Leupers, R. Niemann, W. Rosenstiel, and H. Takada. Corrections to the 2003 hardcopy version of the book were proposed by David Hec, Thomas Wiederkehr, and Thorsten Wilmer. Of course, the author is responsible for all remaining errors and mistakes.

Acknowledgments also go to all those who have patiently accepted the author's additional workload during the writing of this book and his resulting reduced availability for professional as well as personal partners.

Finally, it should be mentioned that Kluwer Academic Publishers (now Springer) has supported the publication of the book from its initial conception. Their support has been stimulating during the work on this book.

Chapter 1

INTRODUCTION

1.1 Terms and scope

Until the late eighties, information processing was associated with large mainframe computers and huge tape drives. During the nineties, this shifted towards information processing being associated with personal computers, PCs. The trend towards miniaturization continues and the majority of information processing devices will be small portable computers integrated into larger products. Their presence in these larger products, such as telecommunication equipment will be less obvious than for the PC. Hence, the new trend has also been called the **disappearing computer**. However, with this new trend, the computer will actually not disappear, it will be everywhere. This new type of information technology applications has also been called **ubiquitous computing** [Weiser, 2003], **pervasive computing** [Hansmann, 2001], [Burkhardt, 2001], and **ambient intelligence** [Koninklijke Philips Electronics N.V., 2003], [Marzano and Aarts, 2003]. These three terms focus on only slightly different aspects of future information technology. Ubiquitous computing focuses on the long term goal of providing "information anytime, anywhere", whereas pervasive computing focuses a somewhat more on practical aspects and the exploitation of already available technology. For ambient intelligence, there is some emphasis on communication technology in future homes and smart buildings. Embedded systems are one of the origins of these three areas and they provide a major part of the necessary technology. **Embedded systems are information processing systems that are embedded into a larger product** and that are normally not directly visible to the user. Examples of embedded systems include information processing systems in telecommunication equipment, in transportation systems, in fabrication equipment and in consumer electronics. Common characteristics of these systems are the following:

- Frequently, embedded systems are connected to the physical environment through **sensors** collecting information about that environment and **actuators**[1] controlling that environment.

- Embedded systems have to be **dependable**.

  Many embedded systems are safety-critical and therefore have to be dependable. Nuclear power plants are an example of extremely safety-critical systems that are at least partially controlled by software. Dependability is, however, also important in other systems, such as cars, trains, airplanes etc. A key reason for being safety-critical is that these systems are directly connected to the environment and have an immediate impact on the environment.

  Dependability encompasses the following aspects of a system:

  1 **Reliability:** Reliability is the probability that a system will not fail.

  2 **Maintainability:** Maintainability is the probability that a failing system can be repaired within a certain time-frame.

  3 **Availability:** Availability is the probability that the system is available. Both the reliability and the maintainability must be high in order to achieve a high availability.

  4 **Safety:** This term describes the property that a failing system will not cause any harm.

  5 **Security:** This term describes the property that confidential data remains confidential and that authentic communication is guaranteed.

- Embedded systems have to be **efficient**. The following metrics can be used for evaluating the efficiency of embedded systems:

  1 **Energy:** Many embedded systems are mobile systems obtaining their energy through batteries. According to forecasts [SEMATECH, 2003], battery technology will improve only at a very slow rate. However, computational requirements are increasing at a rapid rate (especially for multimedia applications) and customers are expecting long run-times from their batteries. Therefore, the available electrical energy must be used very efficiently.

  2 **Code-size:** All the code to be run on an embedded system has to be stored with the system. Typically, there are no hard discs on which code can be stored. Dynamically adding additional code is still an exception and limited to cases such as Java-phones and set-top boxes.

[1] In this context, actuators are devices converting numerical values into physical effects.

Due to all the other constraints, this means that the code-size should be as small as possible for the intended application. This is especially true for **systems on a chip** (SoCs), systems for which all the information processing circuits are included on a single chip. If the instruction memory is to be integrated onto this chip, it should be used very efficiently.

3 **Run-time efficiency:** The minimum amount of resources should be used for implementing the required functionality. We should be able to meet time constraints using the least amount of hardware resources and energy. In order to reduce the energy consumption, clock frequencies and supply voltages should be as small as possible. Also, only the necessary hardware components should be present. Components which do not improve the worst case execution time (such as many caches or memory management units) can be omitted.

4 **Weight:** All portable systems must be of low weight. Low weight is frequently an important argument for buying a certain system.

5 **Cost:** For high-volume embedded systems, especially in consumer electronics, competitiveness on the market is an extremely crucial issue, and efficient use of hardware components and the software development budget are required.

- These systems are **dedicated towards a certain application**.

  For example, processors running control software in a car or a train will always run that software, and there will be no attempt to run a computer game or spreadsheet program on the same processor. There are mainly two reasons for this:

  1 Running additional programs would make those systems less dependable.

  2 Running additional programs is only feasible if resources such as memory are unused. No unused resources should be present in an efficient system.

- Most embedded systems do not use keyboards, mice and large computer monitors for their user-interface. Instead, there is a **dedicated user-interface** consisting of push-buttons, steering wheels, pedals etc. Because of this, the user hardly recognizes that information processing is involved. Due to this, the new era of computing has also been characterized by the disappearing computer.

- Many embedded systems must meet **real-time constraints**. Not completing computations within a given time-frame can result in a serious loss of

the quality provided by the system (for example, if the audio or video quality is affected) or may cause harm to the user (for example, if cars, trains or planes do not operate in the predicted way). *A time-constraint is called **hard** if not meeting that constraint could result in a catastrophe* [Kopetz, 1997]. All other time constraints are called **soft**.

Many of today's information processing systems are using techniques for speeding-up information processing *on the average*. For example, caches improve the average performance of a system. In other cases, reliable communication is achieved by repeating certain transmissions. For example, Internet protocols typically rely on resending messages in case the original messages have been lost. On the average, such repetitions result in a (hopefully only) small loss of performance, even though for a certain message the communication delay can be orders of magnitude larger than the normal delay. In the context of real-time systems, arguments about the average performance or delay cannot be accepted. **A guaranteed system response has to be explained without statistical arguments** [Kopetz, 1997].

- Many embedded systems are **hybrid systems** in the sense that they include analog and digital parts. Analog parts use continuous signal values in continuous time, whereas digital parts use discrete signal values in discrete time.

- Typically, embedded systems are **reactive systems**. They can be defined as follows: *A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment* [Bergé et al., 1995]. Reactive systems can be thought of as being in a certain state, waiting for an input. For each input, they perform some computation and generate an output and a new state. Therefore, automata are very good models of such systems. Mathematical functions, which describe the problems solved by most algorithms, would be an inappropriate model.

- Embedded systems are **under-represented in teaching and in public discussions**. *Embedded chips aren't hyped in TV and magazine ads ...*[Ryan, 1995]. One of the problems in teaching embedded system design is the equipment which is needed to make the topic interesting and practical. Also, real embedded systems are very complex and hence difficult to teach.

Due to this set of common characteristics (except for the last one), it does make sense to analyze common approaches for designing embedded systems, instead of looking at the different application areas only in isolation.

Actually, not every embedded system will have all the above characteristics. We can define the term "embedded system" also in the following way: *Information processing systems meeting most of the characteristics listed above are*

called ***embedded systems***. This definition includes some fuzziness. However, it seems to be neither necessary nor possible to remove this fuzziness.

Most of the characteristics of embedded systems can also be found in a recently introduced type of computing: ubiquitous or pervasive computing, also called ambient intelligence. The key goal of this type of computing is to make information available *anytime, anywhere*. It does therefore comprise communication technology. Fig. 1.1 shows a graphical representation of how ubiquitous computing is influenced by embedded systems and by communication technology.

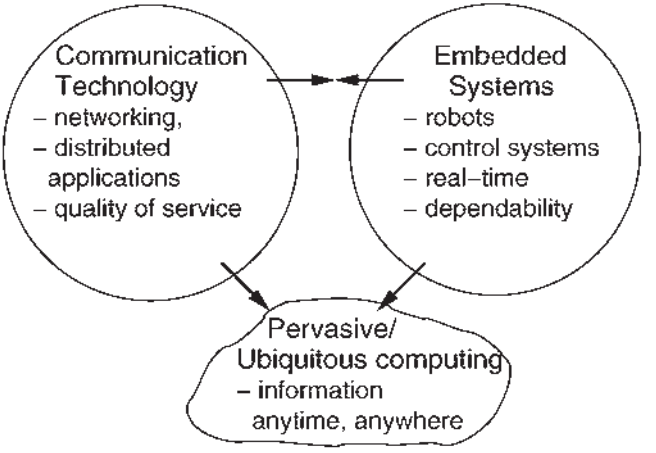

Figure 1.1. Influence of embedded systems on ubiquitous computing

For example, ubiquitous computing has to meet real-time and dependability requirements of embedded systems while using fundamental techniques of communication technology, such as networking.

1.2 Application areas

The following list comprises key areas in which embedded systems are used:

- **Automotive electronics:** Modern cars can be sold only if they contain a significant amount of electronics. These include air bag control systems, engine control systems, anti-braking systems (ABS), air-conditioning, GPS-systems, safety features, and many more.
- **Aircraft electronics:** A significant amount of the total value of airplanes is due to the information processing equipment, including flight control systems, anti-collision systems, pilot information systems, and others. Dependability is of utmost importance.

- **Trains:** For trains, the situation is similar to the one discussed for cars and airplanes. Again, safety features contribute significantly to the total value of trains, and dependability is extremely important.

- **Telecommunication:** Mobile phones have been one of the fastest growing markets in the recent years. For mobile phones, radio frequency (RF) design, digital signal processing and low power design are key aspects.

- **Medical systems:** There is a huge potential for improving the medical service by taking advantage of information processing taking place within medical equipment.

- **Military applications:** Information processing has been used in military equipment for many years. In fact, some of the very first computers analyzed military radar signals.

- **Authentication systems:** Embedded systems can be used for authentication purposes.

  For example, advanced payment systems can provide more security than classical systems. The SMARTpen® [IMEC, 1997] is an example of such an advanced payment system (see fig. 1.2).

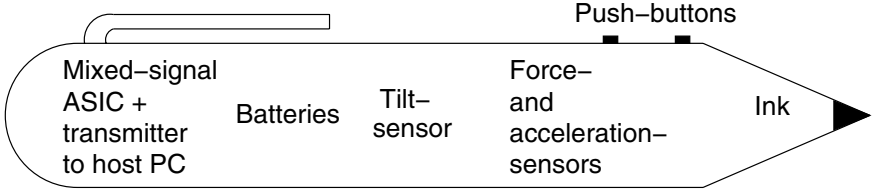

Figure 1.2. SMARTpen

  The SMARTpen is a pen-like instrument analyzing physical parameters while its user is signing. Physical parameters include the tilt, force and acceleration. These values are transmitted to a host PC and compared with information available about the user. As a result, it can be checked if both the image of the signature as well as the way it has been produced coincide with the stored information.

  Other authentication systems include finger print sensors or face recognition systems.

- **Consumer electronics:** Video and audio equipment is a very important sector of the electronics industry. The information processing integrated into such equipment is steadily growing. New services and better quality are implemented using advanced digital signal processing techniques.

Many TV sets, multimedia phones, and game consoles comprise high-performance processors and memory systems. They represent special cases of embedded systems.

- **Fabrication equipment:** Fabrication equipment is a very traditional area in which embedded systems have been employed for decades. Safety is very important for such systems, the energy consumption is less a problem. As an example, fig. 1.3 (taken from Kopetz [Kopetz, 1997]) shows a container connected to a pipe. The pipe includes a valve and a sensor. Using the readout from the sensor, a computer may have to control the amount of liquid leaving the pipe.

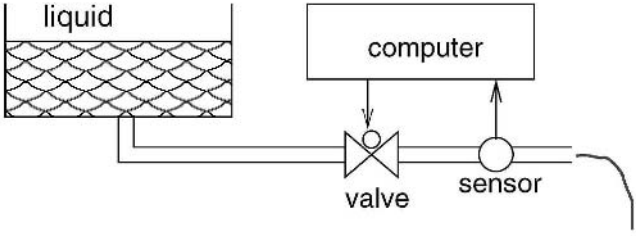

Figure 1.3. Controlling a valve

The valve is an example of an actuator (see definition on page 2).

- **Smart buildings:** Information processing can be used to increase the comfort level in buildings, can reduce the energy consumption within buildings, and can improve safety and security. Subsystems which traditionally were unrelated have to be connected for this purpose. There is a trend towards integrating air-conditioning, lighting, access control, accounting and distribution of information into a single system. For example, energy can be saved on cooling, heating and lighting of rooms which are empty. Available rooms can be displayed at appropriate places, simplifying ad-hoc meetings and cleaning. Air condition noise can be reduced to a level required for the actual operating conditions. Intelligent usage of blinds can optimize lighting and air-conditioning. Tolerance levels of air conditioning subsystems can be increased for empty rooms, and the lighting can be automatically reduced. Lists of non-empty rooms can be displayed at the entrance of the building in emergency situations (provided the required power is still available).

  Initially, such systems will mostly be present only in high-tech office buildings.

- **Robotics:** Robotics is also a traditional area in which embedded systems have been used. Mechanical aspects are very important for robots. Most of

the characteristics described above also apply to robotics. Recently, some new kinds of robots, modeled after animals or human beings, have been designed. Fig. 1.4 shows such a robot.

Figure 1.4. Robot "Johnnie" (courtesy H. Ulbrich, F. Pfeiffer, Lehrstuhl für Angewandte Mechanik, TU München), ©TU München

This set of examples demonstrates the huge variety of embedded systems. Why does it make sense to consider all these types of embedded systems in one book? It makes sense because information processing in these systems has many common characteristics, despite being physically so different.

1.3 Growing importance of embedded systems

The size of the embedded system market can be analyzed from a variety of perspectives. Looking at the number of processors that are currently used, it has been estimated that about 79% of all the processors are used in embedded systems[2]. Many of the embedded processors are 8-bit processors, but despite this, 75% of all 32-bit processors are integrated into embedded systems [Stiller, 2000]. Already in 1996, it was estimated that the average American came into contact with 60 microprocessors per day [Camposano and Wolf, 1996]. Some

[2]Source: Electronic design.

high-end cars contain more than 100 processors[3]. These numbers are much larger than what is typically expected, since most people do not realize that they are using processors. The importance of embedded systems was also stated by journalist Mary Ryan [Ryan, 1995]:

... embedded chips form the backbone of the electronics driven world in which we live. ... they are part of almost everything that runs on electricity.

According to quite a number of forecasts, the embedded system market will soon be much larger than the market for PC-like systems. Also, the amount of software used in embedded systems is expected to increase. According to Vaandrager, *for many products in the area of consumer electronics the amount of code is **doubling every two years*** [Vaandrager, 1998].

Embedded systems form the basis of the so-called **post-PC era**, in which information processing is more and more moving away from just PCs to embedded systems.

The growing number of applications results in the need for design technologies supporting the design of embedded systems. Currently available technologies and tools still have important limitations. For example, there is still a need for better specification languages, tools generating implementations from specifications, timing verifiers, real-time operating systems, low-power design techniques, and design techniques for dependable systems. This book should help teaching the essential issues and should be a stepping stone for starting more research in the area.

1.4 Structure of this book

Traditionally, the focus of a number of books on embedded systems is on explaining the use of micro-controllers, including their memory, I/O and interrupt structure. There are many such books [Ganssle, 1992], [Ball, 1996], [Ball, 1998], [Barr, 1999], [Ganssle, 2000].

Due to this increasing complexity of embedded systems, this focus has to be extended to include at least the different specification languages, hardware/software codesign, compiler techniques, scheduling and validation techniques. In the current book, we will be covering all these areas. The goal is to provide students with an introduction to embedded systems, enabling students to put the different areas into perspective.

For further details, we recommend a number of sources (some of which have also been used in preparing this book):

[3] According to personal communication.

- There is a large number of sources of information on specification languages. These include earlier books by Young [Young, 1982], Burns and Wellings [Burns and Wellings, 1990] and Bergé [Bergé et al., 1995]. There is a huge amount of information on new languages such as SystemC [Müller et al., 2003], SpecC [Gajski et al., 2000], Java etc.
- Approaches for designing and using real-time operating systems (RTOSes) are presented in a book by Kopetz [Kopetz, 1997].
- Real-time scheduling is covered comprehensively in the books by Buttazzo [Buttazzo, 2002] and by Krishna and Shin [Krishna and Shin, 1997].
- Lecture manuscripts of Rajiv Gupta [Gupta, 1998] provide a survey of embedded systems.
- Robotics is an area that is closely linked with embedded systems. We recommend the book by Fu, Gonzalez and Lee [Fu et al., 1987] for information on robotics.
- Additional information is provided by the ARTIST roadmap [Bouyssounouse and Sifakis, 2005] and a book by Vahid [Vahid, 2002].

The structure of this book corresponds to that of a simplified design information flow for embedded systems, shown in figure 1.5.

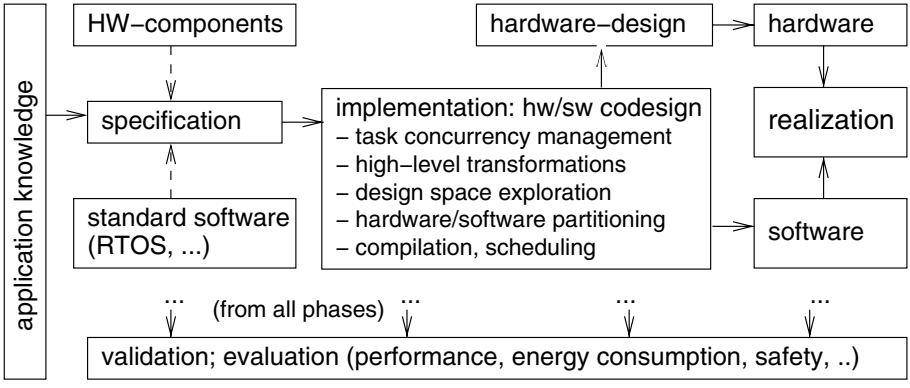

Figure 1.5. Simplified design information flow

The design information flow starts with ideas in people's heads. These ideas must be captured in a design specification. In addition, standard hardware and software components are typically available and should be reused whenever possible.

Design activities start from the specification. Typically, they involve a consideration of both hardware and software, since both have to be taken into

account for embedded system design. Design activities comprise mapping operations to concurrent tasks, high-level transformations (such as advanced loop transformations), mapping of operations to either hardware or software (called hardware/software partitioning), design space exploration, compilation, and scheduling. It may be necessary to design special purpose hardware or to optimize processor architectures for a given application. However, hardware design is not covered in this book. Standard compilers can be used for the compilation. However, they are frequently not optimized for embedded processors. Therefore, we will also briefly cover compiler techniques that should be applied in order to obtain the required efficiency. Once binary code has been obtained for each task, it can be scheduled precisely. Final software and hardware descriptions can be merged, resulting in a complete description of the design and providing input for fabrication.

At the current state of the art, none of the design steps can be guaranteed to be correct. Therefore, it is necessary to validate the design. Validation consists of checking intermediate or final design descriptions against other descriptions. Evaluation is another activity that is required during various phases of the design. Various properties can be evaluated, including performance, dependability, energy consumption, manufacturability etc.

Note that fig. 1.5 represents the flow of **information about the design object**. The sequence of design **activities** has to be consistent with that flow. This does not mean, however, that design activities correspond to a simple path from ideas to the final product. In practice, some design activities have to be repeated. For example, it may become necessary to return to the specification or to obtain additional application knowledge. It may also become necessary to consider additional standard operating systems if the initially considered operating system cannot be used for performance reasons.

Consistent with the design information flow, this book is structured as follows: in chapter 2, we will discuss specification languages. Key hardware components of embedded systems will be presented in chapter 3. Chapter 4 is devoted towards the description of real-time operating systems, other types of such *middleware*, and standard scheduling techniques. Standard design techniques for implementing embedded systems - including compilation issues - will be discussed in chapter 5. Finally, validation will be covered in the last chapter.

Chapter 2

SPECIFICATIONS

2.1 Requirements

Consistent with the simplified design flow (see fig. 1.5), we will now describe requirements and approaches for specifying embedded systems.

There may still be cases in which the specification of embedded systems is captured in a natural language, such as English. However, this approach is absolutely inappropriate since it lacks key requirements for specification techniques: it is necessary to check specifications for completeness, absence of contradictions and it should be possible to derive implementations from the specification in a systematic way. Therefore, specifications should be captured in machine readable, formal languages. Specification languages for embedded systems should be capable of representing the following features[1]:

- **Hierarchy:** Human beings are generally not capable of comprehending systems that contain many objects (states, components) having complex relations with each other. The description of all real-life systems needs more objects than human beings can understand. Hierarchy is the only mechanism that helps to solve this dilemma. Hierarchies can be introduced such that humans need to handle only a small number of objects at any time.

  There are two kinds of hierarchies:

[1] Information from the books of Burns et al. [Burns and Wellings, 1990], Bergé et al. [Bergé et al., 1995] and Gajski et al. [Gajski et al., 1994] is used in this list.

  - **Behavioral hierarchies:** Behavioral hierarchies are hierarchies containing objects necessary to describe the system behavior. States, events and output signals are examples of such objects.
  - **Structural hierarchies:** Structural hierarchies describe how systems are composed of physical components.

    For example, embedded systems can be comprised of processors, memories, actors and sensors. Processors, in turn, include registers, multiplexers and adders. Multiplexers are composed of gates.

- **Timing-behavior:** Since explicit timing requirements are one of the characteristics of embedded systems, timing requirements must be captured in the specification.

- **State-oriented behavior:** It was already mentioned in chapter 1 that automata provide a good mechanism for modeling reactive systems. Therefore, the state-oriented behavior provided by automata should be easy to describe. However, classical automata models are insufficient, since they cannot model timing and since hierarchy is not supported.

- **Event-handling:** Due to the reactive nature of embedded systems, mechanisms for describing events must exist. Such events may be external events (caused by the environment) or internal events (caused by components of the system).

- **No obstacles to the generation of efficient implementations:** Since embedded systems have to be efficient, no obstacles prohibiting the generation of efficient realizations should be present in the specification.

- **Support for the design of dependable systems:** Specification techniques should provide support for designing dependable systems. For example, specification languages should have unambiguous semantics, facilitate formal verification and be capable of describing security and safety requirements.

- **Exception-oriented behavior:** In many practical, systems exceptions do occur. In order to design dependable systems, it must be possible to describe actions to handle exceptions easily. It is not acceptable that exceptions have to be indicated for each and every state (like in the case of classical state diagrams). Example: In fig. 2.1, input k might correspond to an exception.

  Specifying this exception at each state makes the diagram very complex. The situation would get worse for larger state diagrams with many transitions. We will later show, how all the transitions can be replaced by a single one.

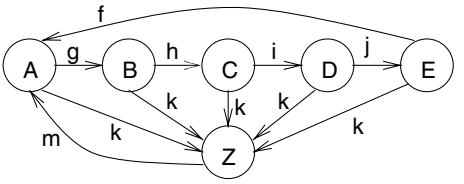

Figure 2.1. State diagram with exception k

- **Concurrency:** Real-life systems are distributed, concurrent systems. It is therefore necessary to be able to specify concurrency conveniently.
- **Synchronization and communication:** Concurrent actions have to be able to communicate and it must be possible to agree on the use of resources. For example, it is necessary to express mutual exclusion.
- **Presence of programming elements:** Usual programming languages have proven to be a convenient means of expressing computations that have to be performed. Hence, programming language elements should be available in the specification technique used. Classical state diagrams do not meet this requirement.
- **Executability:** Specifications are not automatically consistent with the ideas in people's heads. Executing the specification is a means of plausibility checking. Specifications using programming languages have a clear advantage in this context.
- **Support for the design of large systems:** There is a trend towards large and complex embedded software programs. Software technology has found mechanisms for designing such large systems. For example, object-orientation is one such mechanism. It should be available in the specification methodology.
- **Domain-specific support:** It would of course be nice if the same specification technique could be applied to all the different types of embedded systems, since this would minimize the effort for developing specification techniques and tool support. However, due to the wide range of application domains, there is little hope that one language can be used to efficiently represent specifications in all domains. For example, control-dominated, data-dominated, centralized and distributed applications-domains can all benefit from language features dedicated towards those domains.
- **Readability:** Of course, specifications have to be readable by human beings. Preferably, they should also be machine-readable into order to process them in a computer.

- **Portability and flexibility:** Specifications should be independent of specific hardware platforms so that they can be easily used for a variety of target platforms. They should be flexible such that small changes of the system's features should also require only small changes to the specification.
- **Termination:** It should be feasible to identify processes that will terminate from the specification.
- **Support for non-standard I/O-devices:** Many embedded systems use I/O-devices other than those typically found on a PC. It should be possible to describe inputs and outputs for those devices conveniently.
- **Non-functional properties:** Actual systems have to exhibit a number of non-functional properties, such as fault-tolerance, size, extendibility, expected lifetime, power consumption, weight, disposability, user friendliness, electromagnetic compatibility (EMC) etc. There is no hope that all these properties can be defined in a formal way.
- **Appropriate model of computation:** In order to describe computations, computational models are required. Such models will be described in the next section.

From the list of requirements, it is already obvious that there will not be any formal language capable of meeting all these requirements. Therefore, in practice, we have to live with compromises. The choice of the language used for an actual design will depend on the application domain and the environment in which the design has to be performed. In the following, we will present a survey of languages that can be used for actual designs.

2.2 Models of computation

Applications of information technology have so far very much relied on the von Neumann paradigm of sequential computing. This paradigm is not appropriate for embedded systems, in particular those with real-time requirements, since there is no notion of time in von Neumann computing. Other models of computation are more adequate. **Models of computation** can be described as follows [Lee, 1999]:

- Models of computation define **components**. Procedures, processes, functions, finite state machines are possible components.
- Models of computation define **communication protocols**. These protocols constrain the mechanism by which components can interact. Asynchronous

message passing and rendez-vous based communication are examples of communication protocols.

- Models of computation possibly also define what components know about each other (the information which components share). For example, they might share information about global variables.

Models of computation do not define the vocabulary of the interaction of the components.

Examples of models of computation capable of describing concurrency include the following [Lee, 1999], [Janka, 2002], [Jantsch, 2003]:

- **Communicating finite state machines** (CFSMs): CFSMs are collections of finite state machines communicating with each other. Methods for communication must be defined. This model of computation is used, for example, for StateCharts (see next section), the StateChart variant StateFlow, and SDL (see page 30).
- **Discrete event model:** In this model, there are events carrying a totally ordered time stamp, indicating the time at which the event occurs. Discrete event simulators typically contain a global event queue sorted by time. The disadvantage is that it relies on a global notion of one or more event queues, making it difficult to map the semantic model onto specific implementations. Examples include VHDL (see page 59), Verilog (see page 75), and Simulink from MathWorks (see page 79).
- **Differential equations:** Differential equations are capable to model analog circuits and physical systems. Hence, they can find applications in embedded system modeling.
- **Asynchronous message passing:** In asynchronous message passing, processes communicate by sending messages through channels that can buffer the messages. The sender does not need to wait for the receiver to be ready to receive the message. In real life, this corresponds to sending a letter. A potential problem is the fact that messages may have to be stored and that message buffers can overflow.

  There are several variations of this scheme, including Kahn process networks (see page 53) and dataflow models.

  A **dataflow program** is specified by a directed graph where the nodes (vertices), called "actors", represent computations and the arcs represent first-in first-out (FIFO) channels. The computation performed by each actor is assumed to be functional, that is, based on the input values only. Each process in a dataflow graph is decomposed into a sequence of firings, which are atomic actions. Each firing produces and consumes tokens.

Of particular interest is **synchronous dataflow** (SDF), which requires processes to consume and produce a fixed number of tokens each firing. SDF can be statically scheduled, which makes implementations very efficient.

- **Synchronous message passing:** In synchronous message passing, the components are processes. They communicate in atomic, instantaneous actions called **rendez-vous**. The process reaching the point of communication first has to wait until the partner has also reached its point of communication. There is no risk of overflows, but the performance may suffer.

  Examples of languages following this model of computation include CSP (see page 55) and ADA (see page 55).

Different applications may require the use of different models. While some of the actual languages implement only one of the models, others allow a mix of models.

2.3 StateCharts

The first actual language which will be presented is StateCharts. StateCharts was introduced in 1987 by David Harel [Harel, 1987] and later described more precisely [Drusinsky and Harel, 1989]. StateCharts describes communicating finite state machines. It based on the shared memory concept of communication. According to Harel, the name was chosen since it was *the only unused combination of "flow" or "state" with "diagram" or "chart"*.

We mentioned in the previous section that we need to describe state-oriented behavior. State diagrams are a classical method of doing this. Fig. 2.2 (the same as fig. 2.1) shows an example of a classical state diagram, representing a **finite state machine (FSM)**.

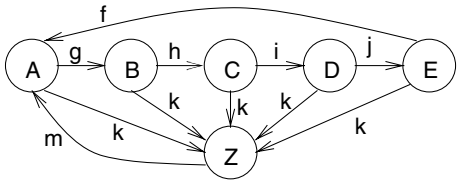

Figure 2.2. State diagram

Circles denote states. At any time, **deterministic** FSMs which we will consider here, can only be in one of their states. Edges denote state transitions. Edge labels represent events. If an event happens, the FSM will change its state as indicated by the edge. FSMs may also generate output (not shown in fig. 2.2). For more information about classical FSMs refer to, for example, Kohavi [Kohavi, 1987].

2.3.1 Modeling of hierarchy

StateCharts describe extended FSMs. Due to this, they can be used for modeling state-oriented behavior. The key extension is **hierarchy**. Hierarchy is introduced by means of **super-states**.

Definitions:

- States comprising other states are called **super-states**.
- States included in super-states are called **sub-states** of the super-states.

Fig. 2.3 shows a StateCharts example. It is a hierarchical version of fig. 2.2.

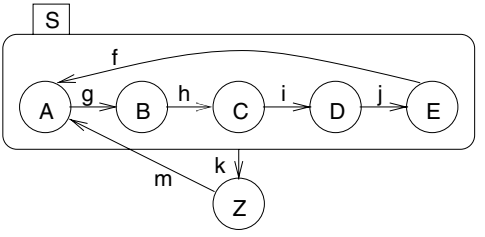

Figure 2.3. Hierarchical state diagram

Super-state S includes states A, B, C, D and E. Suppose the FSM is in state Z (we will also call Z to be an **active state**). Now, if input m is applied to the FSM, then A will be the new state. If the FSM is in S and input k is applied, then Z will be the new state, regardless of whether the FSM is in sub-states A, B, C, D or E of S. In this example, all states contained in S are non-hierarchical states. In general, sub-states of S could again be super-states consisting of sub-states themselves.

Definitions:

- Each state which is not composed of other states is called a **basic state**.
- For each basic state *s*, the super states containing *s* are called **ancestor states**.

The FSM of fig. 2.3 can only be in one of the sub-states of super-state S at any time. Super states of this type are called **OR-super-states**[2].

[2]More precisely, they should be called XOR-super-states, since the FSM is in **either** A, B, D or E. However, this name is not commonly used in the literature.

In fig. 2.3, k might correspond to an exception for which state S has to be left. The example already shows that the hierarchy introduced in StateCharts enables a compact representation of exceptions.

StateCharts allows hierarchical descriptions of systems in which a system description comprises descriptions of subsystems which, in turn, may contain descriptions of subsystems. The description of the entire system can be thought of as a **tree**. The root of the tree corresponds to the system as a whole, and all inner nodes correspond to hierarchical descriptions (in the case of StateCharts called super-nodes). The **leaves of the hierarchy** are non-hierarchical descriptions (in the case of StateCharts called basic states).

So far, we have used explicit, direct edges to basic states to indicate the next state. The disadvantage of that approach is that the internal structure of super-states cannot be hidden from the environment. However, in a true hierarchical environment, we should be able to hide the internal structure so that it can be described later or changed later without affecting the environment. This is possible with other mechanisms for describing the next state.

The first additional mechanism is the **default state mechanism**. It can be used in super-states to indicate the particular sub-states that will be entered if the super-states are entered. In diagrams, default states are identified by edges starting at small filled circles. Fig. 2.4 shows a state diagram using the default state mechanism. It is equivalent to the diagram in fig. 2.3. Note that the filled circle does not constitute a state itself.

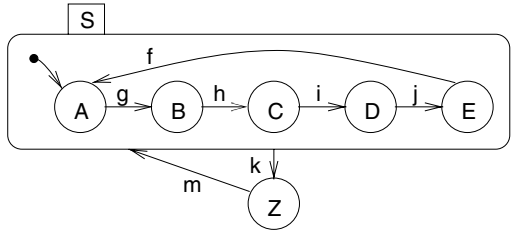

Figure 2.4. State diagram using the default state mechanism

Another mechanism for specifying next states is the **history mechanism**. With this mechanism, it is possible to return to the last sub-state that was active before a super-state was left. The history mechanism is symbolized by a circle containing the letter H. In order to define the next state for the very initial transition into a super-state, the history mechanism is frequently combined with the default mechanism. Fig. 2.5 shows an example.

The behavior of the FSM is now somewhat different. If we input m while the system is in Z, then the FSM will enter A if this is the very first time we enter S, and otherwise it will enter the last state that we were in. This mechanism

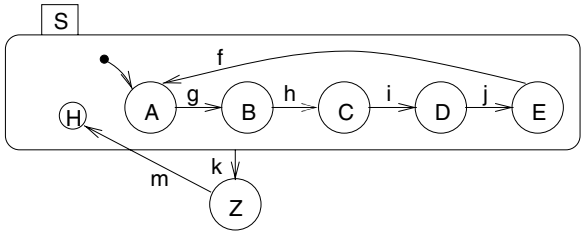

Figure 2.5. State diagram using the history and the default state mechanism

has many applications. For example, if k denotes an exception, we could use input m to return to the state we were in before the exception. States A, B, C, D and E could also call Z like a procedure. After completing "procedure" Z, we would return to the calling state.

Fig. 2.5 can also be redrawn as shown if fig. 2.6. In this case, the symbols for the default and the history mechanism are combined.

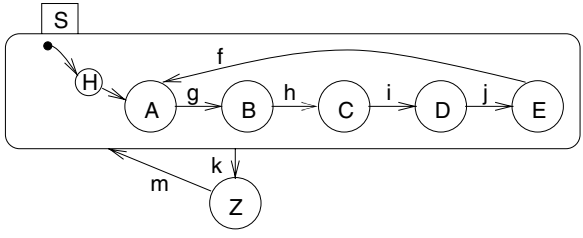

Figure 2.6. Combining the symbols for the history and the default state mechanism

Specification techniques must also be able to describe concurrency conveniently. Towards this end, StateCharts provides a second class of super-states, so called **AND**-states.

Definition: Super-states S are called **AND-super-states** if the system containing S will be in all of the sub-states of S whenever it is in S.

An AND-super-state is included in the answering machine shown in fig. 2.7.

An answering machine normally performs two tasks concurrently: it is monitoring the line for incoming calls and the keys for user input. In fig. 2.7, the corresponding states are called Lwait and Kwait. Incoming calls are processed in state Lproc while the response to pressed keys is generated in state Kproc. For the time being, we assume that the on/off switch (generating events key-off and key-on) is decoded separately and pushing it does not result in entering Kproc. If this switch is pushed, the line monitoring state as well as the key monitoring state are left and re-entered only if the machine is switched on. At that time, default states Lwait and Kwait are entered. While switched on,

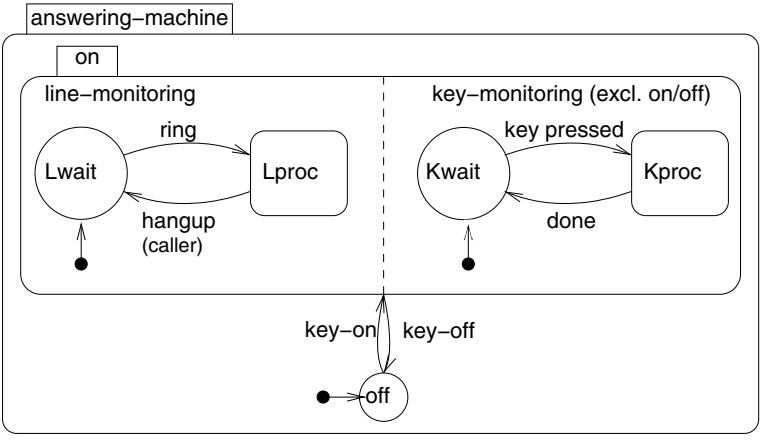

Figure 2.7. Answering machine

the machine will always be in the line monitoring state as well as in the key monitoring state.

For AND-super-states, the sub-states entered as a result of some event can be defined independently. There can be any combination of history, default and explicit transitions. It is crucial to understand that **all** sub-states will always be entered, even if there is just one explicit transition to one of the sub-states. Accordingly, transitions out of an AND-super-state will always result in leaving **all** the sub-states.

For example, let us modify our answering machine such that the on/off switch, like all other switches, is decoded in state Kproc (see fig. 2.8).

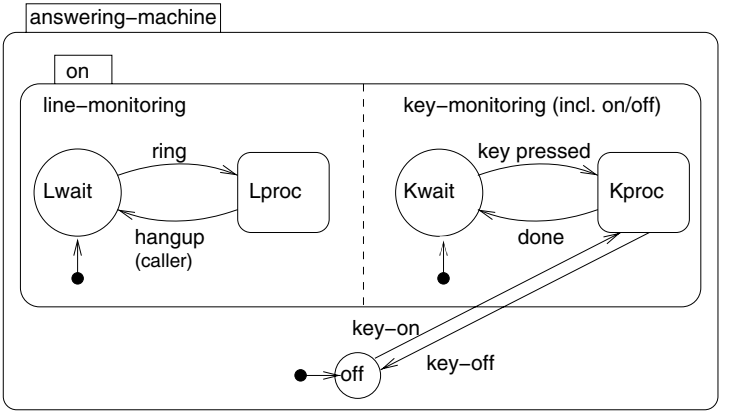

Figure 2.8. Answering machine with modified on/off switch processing

If pushing that key is detected in Kproc, a transition is made to the off state. This transition results in leaving the line-monitoring state as well. Switching the machine on again results in also entering the line-monitoring state.

Summarizing, we can state the following: **States in StateCharts diagrams are either AND-states, OR-states or basic states.**

2.3.2 Timers

Due to the requirement to model time in embedded systems, StateCharts also provides timers. Timers are denoted by the symbol shown in fig. 2.9 (left).

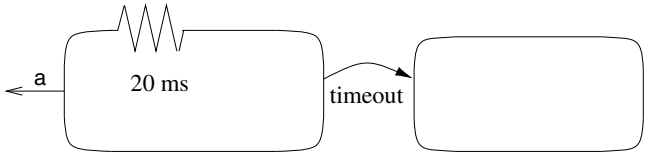

Figure 2.9. Timer in StateCharts

After the system has been in the state containing the timer for the specified period, a time-out will occur and the system will leave the specified state. Timers can be used hierarchically.

Timers can be used, for example, at the next lower level of the hierarchy of the answering machine in order to describe the behavior of state Lproc. Fig. 2.10 shows a possible behavior for that state.

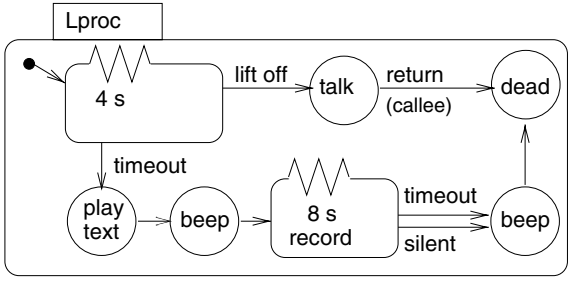

Figure 2.10. Servicing the incoming line in Lproc

Due to the exception-like transition for hangups by the caller in fig. 2.7, state Lproc is terminated whenever the caller hangs up. For hangups (returns) by the callee, the design of state Lproc results in an inconvenience: If the callee hangs up the phone first, the telephone will be dead (and quiet) until the caller has also hung up the phone.

StateCharts do include a number of other language elements. For a full description refer to Harel [Harel, 1987]. A more detailed description of the semantics of the StateMate implementation of StateCharts is described by Drusinsky and Harel [Drusinsky and Harel, 1989].

2.3.3 Edge labels and StateCharts semantics

Until now, we have not considered outputs generated by our extended FSMs. Generated outputs can be specified using edge labels. The general form of an edge label is "event [condition] / reaction". All three label parts are optional. The *reaction*-part describes the reaction of the FSM to a state transition. Possible reactions include the generation of events and assignments to variables. The *condition*-part implies a test of the values of variables or a test of the current state of the system. The *event*-part refers to a test of current events. Events can be generated either internally or externally. Internal events are generated as a result of some transition and are described in *reaction*-parts. External events are usually described in the model environment.

Examples:

- on-key / on:=1 (Event-test and variable assignment),
- [on=1] (Condition test for a variable value),
- off-key [not in Lproc] / on:=0 (Event-test, condition test for a state, variable assignment. The assignment is performed if the event has occurred and the condition is true).

The semantics of edge labels can only be explained in the context of the semantics of StateCharts. According to the semantics of the StateMate implementation of StateCharts [Drusinsky and Harel, 1989], a step-based execution of StateCharts-descriptions is assumed. Each step consists of three phases:

1 In the first phase, the effect of external changes on conditions and events is evaluated. This includes the evaluation of functions which depend on external events. This phase does not include any state changes. In our simple examples, this phase is not actually needed.

2 The next phase is to calculate the set of transitions that should be made in the current step. Variable assignments are evaluated, but the new values are only assigned to temporary variables.

3 In the third phase, state transitions become effective and variables obtain their new values.

The separation into phases 2 and 3 is especially important in order to guarantee a deterministic and reproducible behavior of StateCharts models. Consider the StateCharts model of fig. 2.11.

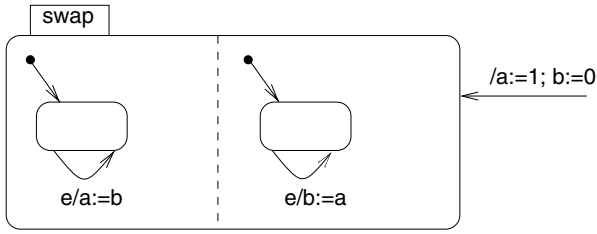

Figure 2.11. Mutually dependent assignments

Due to the separation into two phases, new values for a and b are stored in temporary variables, say a' and b'. In the final phase, temporary variables are copied into the used-defined variables:

phase 2: a':=b; b':=a;

phase 3: a:=a'; b:=b'

As a result, the two variables will be swapped each time an event e happens. This behavior corresponds to that of two cross-coupled registers (one for each variable) connected to the same clock (see fig. 2.11) and reflects the operation of a clocked finite state machine including those two registers.

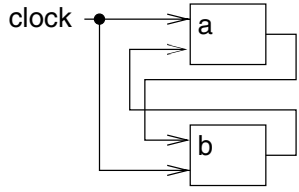

Figure 2.12. Cross-coupled registers

Without the separation into two phases, the result would depend on the sequence in which the assignments are performed. In any case, the same value would be assigned to both variables. This separation into (at least) two phases is quite typical for languages that try to reflect the operation of synchronous hardware. We will find the same separation in VHDL (see page 73).

The three phases are assumed to be executed for each **step**. Steps are assumed to be executed each time events or variables have changed. The execution of a StateChart model consists of the execution of a sequence of steps (see fig. 2.13), each step consisting of three phases.

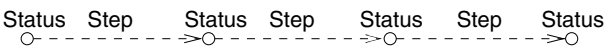

Figure 2.13. Steps during the execution of a StateCharts model

The set of all values of variables, together with the set of events generated (and the current time) is defined as the **status**[3] of a StateCharts model. After executing the third phase, a new status is obtained. The notion of steps allows us to more precisely define the semantics of **events**. Events are generated, as mentioned, either internally or externally. **The visibility of events is limited to the step following the one in which they are generated.** Thus, events behave like single bit values which are stored in permanently enabled registers at one clock transition and have an effect on the values stored at the next clock transition. They do not live forever.

Variables, in contrast, retain their values, until they are reassigned. According to StateCharts semantics, new values of variables are visible to all parts of the model from the step following the step in which the assignment was made onwards. That means, StateCharts semantics implies that new values of variables are propagated to all parts of a model between two steps. StateCharts implicitly assumes a **broadcast mechanism for updates on variables**. For distributed systems, it will be very difficult to update all variables between two steps. Due to this broadcast mechanism, StateCharts is not an appropriate language for modeling distributed systems.

2.3.4 Evaluation and extensions

StateCharts' main application domain is that of local, control-dominated systems. The capability of nesting hierarchies at arbitrary levels, with a free choice of AND- and OR-states, is a key advantage of StateCharts. Another advantage is that the semantics of StateCharts is defined at a sufficient level of detail [Drusinsky and Harel, 1989]. Furthermore, there are quite a number of commercial tools based on StateCharts. StateMate (see http://www.ilogix.com), StateFlow (see http://www.mathworks.com/products/stateflow) and BetterState (see http://www.windriver.com/products/betterstate/index.html) are examples of commercial tools based on StateCharts. Many of these are capable of translating StateCharts into equivalent descriptions in C or VHDL (see page 59). From VHDL, hardware can be generated using synthesis tools. Therefore, StateCharts-based tools provide a complete path from StateCharts-based spec-

[3]We would normally use the term "state" instead of "status". However, the term "state" has a different meaning in StateCharts.

ifications down to hardware. Generated C programs can be compiled and executed. Hence, a path to software-based realizations also exists.

Unfortunately, the efficiency of the automatic translation is frequently a concern. For example, sub-states of AND-states may be mapped to UNIX-processes. This concept is acceptable for simulating StateCharts, but will hardly lead to efficient implementations on small processors. The productivity gain from object-oriented programming is not available in StateCharts, since it is not object-oriented. Furthermore, the broadcast mechanism makes it less appropriate for distributed systems. StateCharts do not comprise program constructs for describing complex computation and cannot describe hardware structures or non-functional behavior.

Commercial implementations of StateCharts typically provide some mechanisms for removing the limitations of StateCharts. For example, C-code can be used to represent program constructs and **module charts** of StateMate can represent hardware structures.

2.4 General language characteristics

The previous section provides us with some first examples of properties of specification languages. These examples help us to understand a more general discussion of language properties in this section before we will discuss more languages in the sections that will follow. There are several characteristics by which we can compare the properties of languages. The first property is related to the distinction between deterministic and non-deterministic models already touched in our discussion of StateCharts.

2.4.1 Synchronous and asynchronous languages

A problem which exists for some languages based on general communicating finite state machines and sets of processes described in ADA or Java is that they are non-deterministic, since the order in which executable processes are executed is not specified. The order of execution may well affect the result. This effect can have a number of negative consequences, such as, for example, problems with verifying a certain design. The *non-determinism* is avoided with so-called synchronous languages. Synchronous languages describe concurrently operating automata. ... *when automata are composed in parallel, a transition of the product is made of the "simultaneous" transitions of all of them* [Halbwachs, 1998]. This means: we do not have to consider all the different sequences of state changes of the automata that would be possible if each of them had its own clock. Instead, we can assume the presence of a single global clock. Each clock tick, all inputs are considered, new outputs

and states are calculated and then the transitions are made. This requires a fast broadcast mechanism for all parts of the model. This idealistic view of concurrency has the advantage of guaranteeing **deterministic behavior**. This is a restriction if compared to the general CFSM model, in which each FSM can have its own clock. Synchronous languages reflect the principles of operation in synchronous hardware and also the semantics found in control languages such as IEC 60848 and STEP 7 (see page 78). Guaranteeing a deterministic behavior for all language features has been a design goal for the synchronous languages Esterel (see page 79) [Esterel, 2002] and Lustre [Halbwachs et al., 1991]. Due to the three simulation phases in StateCharts, StateCharts is also a synchronous language (and it is deterministic). Just like StateCharts, synchronous languages are difficult to use in a distributed environment, where the concept of a single clock results in difficulties.

2.4.2 Process concepts

There are several criteria by which we can compare the process concepts in different programming languages:

- The **number of processes** can be either **static** or **dynamic**. A static number of processes simplifies the implementation and is sufficient if each process models a piece of hardware and if we do not consider "hot-plugging" (dynamically changing the hardware architecture). Otherwise, dynamic process creation (and death) should be supported.
- Processes can either be statically **nested** or all declared at the same level. For example, StateCharts allows nested process declarations while SDL (see page 30) does not. Nesting provides encapsulation of concerns.
- Different techniques for **process creation** exist. Process creation can result from an elaboration of the process declaration in the source code, through the fork and join mechanism (supported for example in Unix), and also through explicit process creation calls.

StateCharts is limited to a static number of processes. Processes can be nested. Process creation results from an elaboration of the source code.

2.4.3 Synchronization and communication

There are essentially two communication paradigms: **shared memory** and **message passing**.

For shared memory, all variables can be accessed from all processes. Access to shared memory should be protected, unless access is totally restricted to

reads. If writes are involved, exclusive access to the memory must be guaranteed while processes are accessing shared memories. Segments of code, for which exclusive access must be guaranteed, are called **critical sections**. Several mechanisms for guaranteeing exclusive access to resources have been proposed. These include semaphores, conditional critical regions and monitors. Refer to books on operating systems for a description of the different techniques. Shared memory-based communication can be very fast, but is difficult to implement in multiprocessor systems if no common memory is physically available.

For message passing, messages are sent and received just like mails are sent on the Internet. Message passing can be implemented easily even if no common memory is available. However, message passing is generally slower than shared memory based communication. For this kind of communication, we can distinguish between the following three techniques:

- **asynchronous message passing**, also called **non-blocking communication** (see page 17),
- **synchronous message passing** or **blocking communication, rendez-vous based communication** (see page 18),
- **extended rendez-vous, remote invocation:** the transmitter is allowed to continue only after an acknowledgment has been received from the receiver. The recipient does not have to send this acknowledgment immediately after receiving the message, but can do some preliminary checking before actually sending the acknowledgment.

StateCharts allows global variables and hence uses the shared memory model.

2.4.4 Specifying timing

Burns and Wellings [Burns and Wellings, 1990] define the following four requirements for specification languages:

- Access to a timer, which provides a means to **measure elapsed time**:

  CSP, for example, meets this requirement by providing channels which are actually timers. Read operations to such a channel return the current time.

- Means for **delaying of processes** for a specified time:

  Typically, real-time languages provide some delay construct. In VHDL, the wait for-statement (see page 69) can be used.

- Possibility to specify **timeouts**:

Real-time languages usually also provide some timeout construct.

- Methods for specifying **deadlines** and **schedules**:

  Unfortunately, most languages do not allow to specify timing constraints. If they can be specified at all, they have to be specified in separate control files, pop-up menus etc.

StateCharts allows timeouts. There is no straightforward way of specifying other timing requirements.

2.4.5 Using non-standard I/O devices

Some languages include special language features enabling direct control over I/O devices. For example, ADA allows variables to be mapped to specific memory addresses. These may be the addresses of memory mapped I/O devices. This way, all I/O operations can be programmed in ADA. ADA also allows procedures to be bound to interrupt addresses.

No direct support for I/O is available in standard StateCharts, but commercial implementations can support I/O programming.

2.5 SDL

Because of the use of shared memory and the broadcast mechanism, StateCharts cannot be used for distributed applications. We will now turn our attention towards a second language, one which is applicable for modeling distributed systems, namely SDL. SDL was designed for distributed applications and is based on asynchronous message passing. It dates back to the early seventies. Formal semantics have been available since the late eighties. The language was standardized by the ITU (International Telecommunication Union). The first standards document is the Z.100 Recommendation published in 1980, with updates in 1984, 1988, 1992 (SDL-92), 1996 and 1999. Relevant versions of the standard include SDL-88, SDL-92 and SDL-2000 [SDL Forum Society, 2003a].

Many users prefer graphical specification languages while others prefer textual specification languages. SDL pleases both types of users since it provides textual as well as graphical formats. Processes are the basic elements of SDL. Processes represent extended finite state machines. Extensions include operations on data. Fig. 2.14 shows the graphical symbols used in the graphical representation of SDL.

As an example, we will consider how the state diagram in fig. 2.15 can be represented in SDL. Fig. 2.15 is the same as fig. 2.4, except that output has been added, state Z deleted, and the effect of signal k changed. Fig. 2.16 contains the

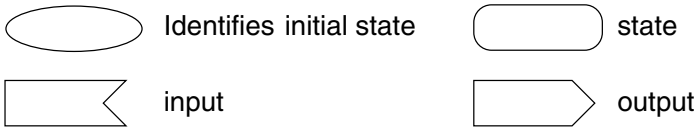

Figure 2.14. Symbols used in the graphical form of SDL

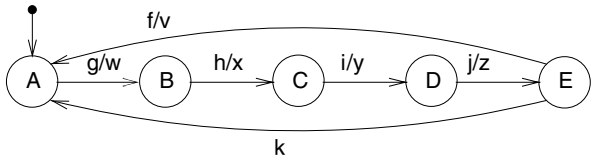

Figure 2.15. FSM described in SDL

corresponding graphical SDL representation. Obviously, the representation is equivalent to the state diagram of fig. 2.15.

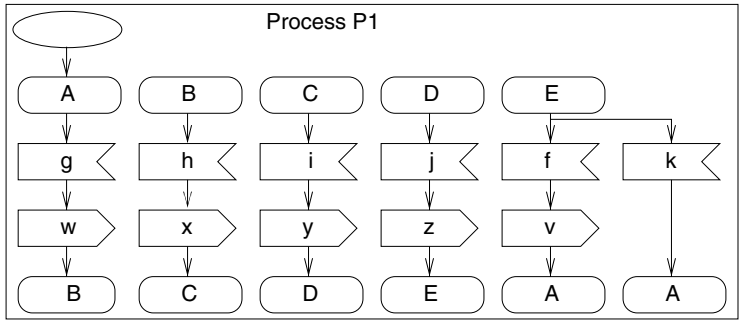

Figure 2.16. SDL-representation of fig. 2.15

As an extension to FSMs, SDL processes can perform operations on data. Variables can be declared locally for processes. Their type can either be pre-defined or defined in the SDL description itself. SDL supports abstract data types (ADTs). The syntax for declarations and operations is similar to that in other languages. Fig. 2.17 shows how declarations, assignments and decisions can be represented in SDL.

SDL also contains programming language elements such as procedures. Procedure calls can also be represented graphically. Object-oriented features became available with version SDL-1992 of the language and were extended with SDL-2000.

Extended FSMs are just the basic elements of SDL descriptions. In general, SDL descriptions will consist of a set of interacting processes, or FSMs. Pro-

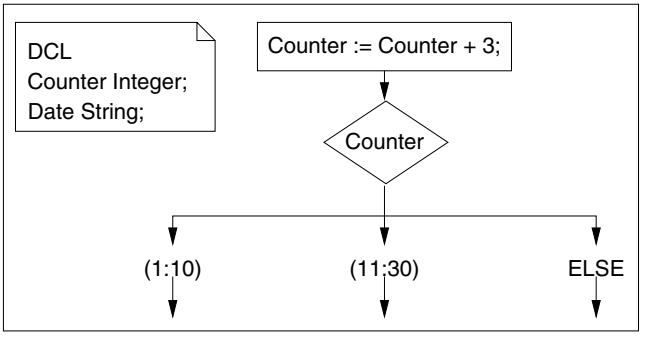

Figure 2.17. Declarations, assignments and decisions in SDL

cesses can send signals to other processes. Semantics of interprocess communication in SDL is based on *first-in first-out* (FIFO) *queues* associated with each process. Signals sent to a particular process will be placed into the corresponding FIFO-queue (see fig. 2.18). Therefore, SDL is based on asynchronous message passing.

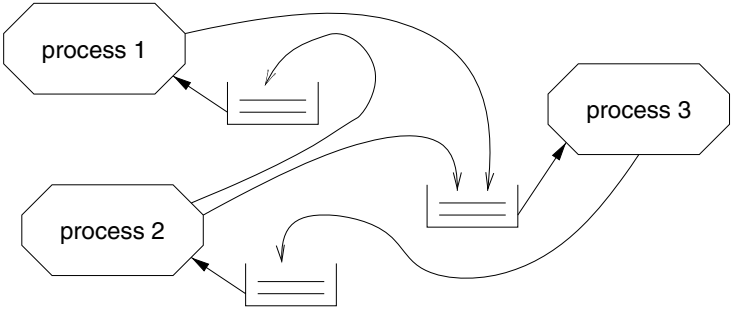

Figure 2.18. SDL interprocess communication

Each process is assumed to fetch the next available entry from the FIFO queue and check whether it matches one of the inputs described for the current state. If it does, the corresponding state transition takes place and output is generated. The entry from the FIFO-queue is ignored if it does not match any of the listed inputs (unless the so-called SAVE-mechanism is used). FIFO-queues are conceptually thought of as being of infinite length. This means: in the description of the semantics of SDL models, FIFO-overflow is never considered. In actual systems, however, FIFO-queues must be of finite length. This is one of the problems of SDL: in order to derive realizations from specifications, safe upper bounds on the length of the FIFO-queues must be proven.

Process interaction diagrams can be used for visualizing which of the processes are communicating with each other. Process interaction diagrams include **channels** used for sending and receiving signals. In the case of SDL, the term "signal" denotes inputs and outputs of modeled automata. Process interaction diagrams are special cases of **block diagrams** (see below).

Example: Fig. 2.19 shows a process interaction diagram B1 with channels Sw1 and Sw2. Brackets include the names of signals propagated along a certain channel.

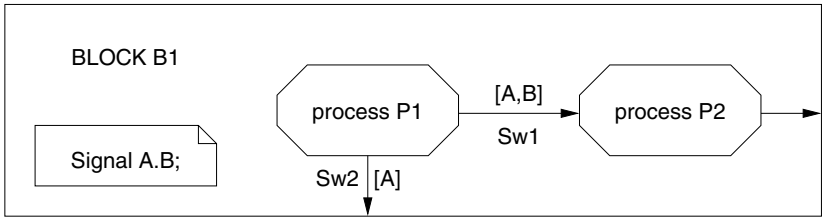

Figure 2.19. Process interaction diagram

There are three ways of indicating the recipient of signals:

1 **Through process identifiers:** by using identifiers of recipient processes in the graphical output symbol (see fig. 2.20 (left)).

Figure 2.20. Describing signal recipients

Actually, the number of processes does not even need to be fixed at compile time, since processes can be generated dynamically at run-time. OFFSPRING represents identifiers of child processes generated dynamically by a process.

2 **Explicitly:** by indicating the channel name (see fig. 2.20 (right)). Sw1 is the name of a channel.

3 **Implicitly:** if signal names imply the channel names, those channels are used. Example: for fig. 2.19, signal B will implicitly always be communicated via channel Sw1.

No process can be defined within any other (processes cannot be nested). However, they can be grouped hierarchically into so-called **blocks**. Blocks at the highest hierarchy level are called **systems**, blocks at the lowest hierarchy level

are called **process interaction diagrams**. B1 can be used within intermediate level blocks (such as within B in fig. 2.21).

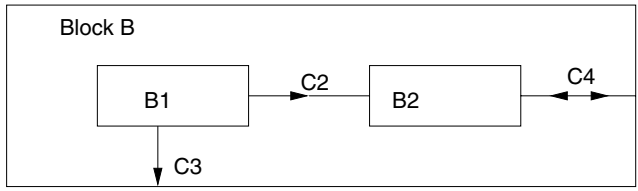

Figure 2.21. SDL block

At the highest level in the hierarchy, we have the system (see fig. 2.22). A system will not have any channels at its boundary if the environment is also modeled as a block.

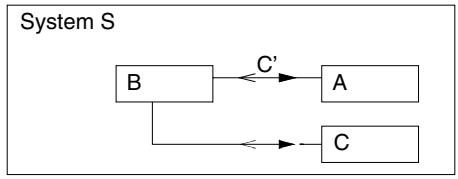

Figure 2.22. SDL system

Fig. 2.23 shows the hierarchy modeled by block diagrams 2.19, 2.21 and 2.22. Process interaction diagrams are next to the *leaves* of the hierarchical description, system descriptions their *root*. Some of the restrictions of modeling hierarchy are removed in version SDL-2000 of the language. With SDL-2000, the descriptive power of blocks and processes is harmonized and replaced by a general *agent* concept.

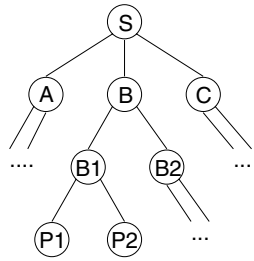

Figure 2.23. SDL hierarchy

In order to support the modeling of time, SDL includes **timers**. Timers can be declared locally for processes. They can be set and reset using SET and

RESET primitives, respectively. Fig. 2.24 shows the use of a timer T. The diagram corresponds to that of fig. 2.16, with the exceptions that timer T is set to the current time plus p during the transition from state D to E. For the transition from E to A we now have a timeout of p time units. If these time units have elapsed before signal f has arrived, a transition to state A is taken without generating output signal v.

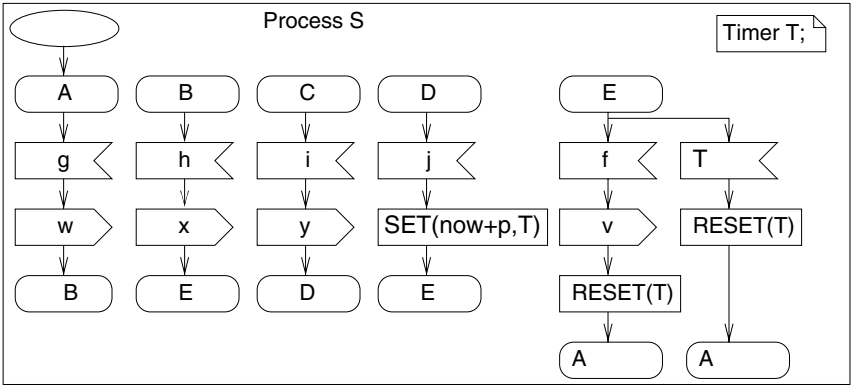

Figure 2.24. Using timer T

SDL can be used, for example, to describe protocol stacks found in computer networks. Fig. 2.25 shows three processors connected through a router. Communication between processors and the router is based on FIFOs.

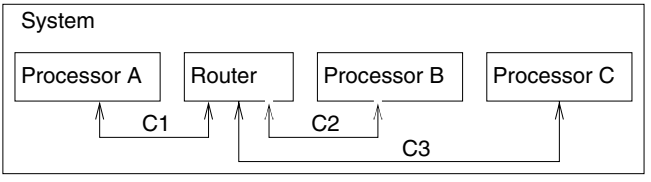

Figure 2.25. Small computer network described in SDL

The processors as well as the router implement layered protocols (see fig. 2.26).

Each layer describes communication at a more abstract level. The behavior of each layer is typically modeled as a finite state machine. The detailed description of these FSMs depends on the network protocol and can be quite complex. Typically, this behavior includes checking and handling error conditions, and sorting and forwarding of information packages.

Currently (2003) available tools for SDL include interfaces to UML (see page 45), MSCs (see page 44), and CHILL (see page 78) from companies such as

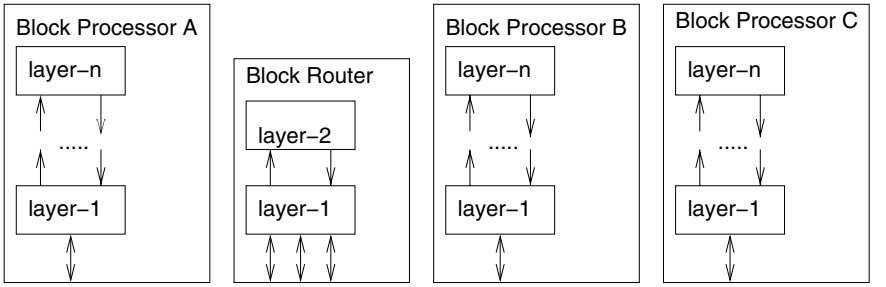

Figure 2.26. Protocol stacks represented in SDL

Telelogic [Telelogic AB, 2003], Cinderella [Cinderella ApS, 2003] and SINTEF. A comprehensive list of tools is available from the SDL forum [SDL Forum Society, 2003b].

SDL is excellent for distributed applications and was used, for example, for specifying ISDN. Commercial tools for SDL are available (see, for example, http://www.telelogic.com). SDL is not necessarily deterministic (the order, in which signals arriving at some FIFO at the same time are processed, is not specified). Reliable implementations require the knowledge of a upper bound on the length of the FIFOs. This upper bound may be difficult to compute. The timer concept is sufficient for soft deadlines, but not for hard ones. Hierarchies are not supported in the same way as in StateCharts. There is no full programming support (but recent revisions of the standard have started to change this) and no description of non-functional properties.

2.6 Petri nets

2.6.1 Introduction

In 1962, Carl Adam Petri published his method for modeling causal dependencies, which became known as Petri nets. The key strength of Petri nets is this focus on causal dependencies. Petri nets do not assume any global synchronization and are therefore especially suited for modeling distributed systems.

Conditions, **events** and a **flow relation** are the key elements of Petri nets. Conditions are either satisfied or not satisfied. Events can happen. The flow relation describes the conditions that must be met before events can happen and it also describes the conditions that become true if events happen.

Graphical notations for Petri nets typically use circles to denote conditions and boxes to denote events. Arrows represent flow relations. Fig. 2.27 shows a first example.

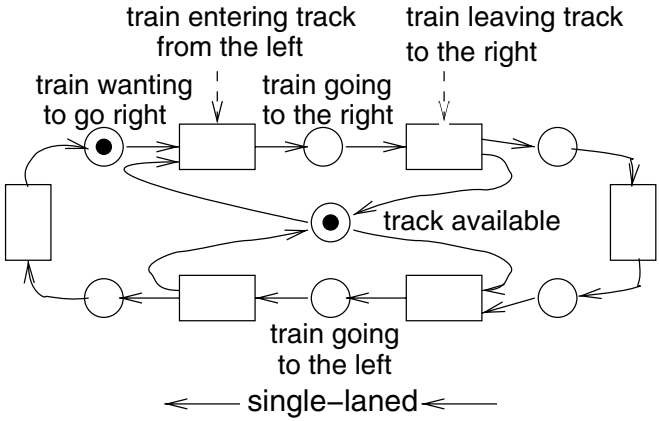

Figure 2.27. Single track railroad segment

This example describes mutual exclusion for trains at a railroad track that must be used in both directions. A token is used to prevent collisions of trains going into opposite directions. In the Petri net representation, that token is symbolized by a condition in the center of the model. A filled circle denotes the situation in which the condition is met (this means: the track is available). When a train wants to go to the right (also denoted by a filled circle in fig. 2.27), the two conditions that are necessary for the event "train entering track from the left" are met. We call these two conditions **preconditions**. If the preconditions of an event are met, it can happen. As a result of that event happening, the token is no longer available and there is no train waiting to enter the track. Hence, the preconditions are no longer met and the filled circles disappear (see fig. 2.28).

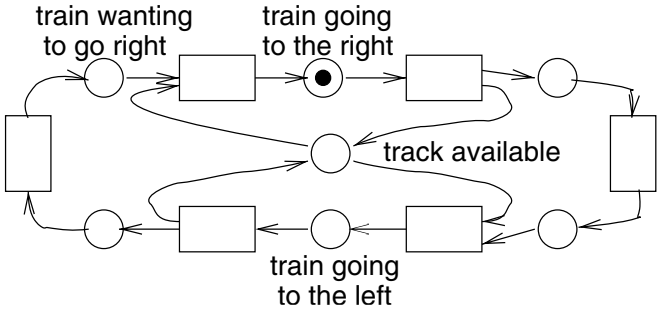

Figure 2.28. Using resource "track"

However, there is now a train going on that track from the left to the right and thus the corresponding condition is met (see fig. 2.28). A condition which is

met after an event happened is called a **postcondition**. In general, an event can happen only if all its preconditions are true (or met). If it happens, the preconditions are no longer met and the postconditions become valid. Arrows identify those conditions which are preconditions of an event and those that are postconditions of an event. Continuing with our example, we see that a train leaving the track will return the token to the condition at the center of the model (see fig. 2.29).

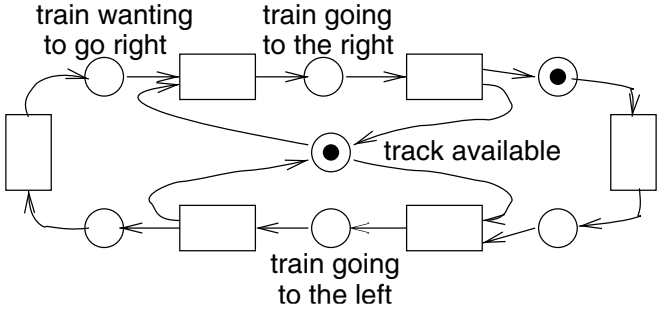

Figure 2.29. Freeing resource "track"

If there are two trains competing for the single-track segment (see fig. 2.30), only one of them can enter.

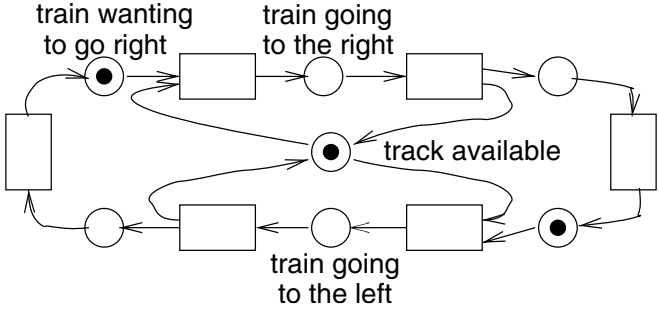

Figure 2.30. Conflict for resource "track"

Let us now consider a larger example:

We are again considering the synchronization of trains. In particular, we are trying to model high-speed Thalys trains traveling between Amsterdam, Cologne, Brussels and Paris. Segments of the train run independently from Amsterdam and Cologne to Brussels. There, the segments get connected and then they run to Paris. On the way back from Paris, they get disconnected at Brussels again. We assume that Thalys trains have to synchronize with some other train at Paris. The corresponding Petri net is shown in fig. 2.31.

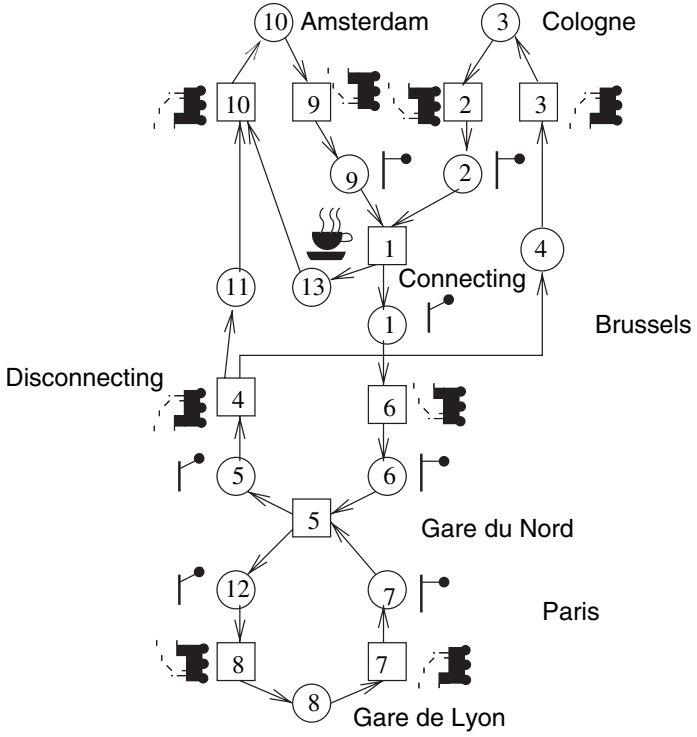

Figure 2.31. Model of Thalys trains running between Amsterdam, Cologne, Brussels, and Paris

Places 3 and 10 model trains waiting at Amsterdam and Cologne, respectively. Transitions 9 and 2 model trains driving from these cities to Brussels. After their arrival at Brussels, places 9 and 2 contain tokens. Transition 1 denotes connecting the two trains. The cup symbolizes the driver of one of the trains, who will have a break at Brussels while the other driver is continuing on to Paris. Transition 5 models synchronization with other trains at the Gare du Nord station of Paris. These other trains connect Gare du Nord with some other station (we have used Gare de Lyon as an example, even though the situation at Paris is somewhat more complex). Of course, Thalys trains do not use steam engines; they are just easier to visualize than modern high speed trains.

A key advantage of Petri nets is that they can be the basis for formal proofs about system properties and that there are standardized ways of generating such proofs. In order to enable such proofs, we need a more formal definition of Petri nets.

2.6.2 Condition/event nets

Condition/event nets are the first class of Petri nets that we will define more formally.

Definition: $N = (C, E, F)$ is called a **net**, iff the following holds:

1 C and E are disjoint sets.

2 $F \subseteq (E \times C) \cup (C \times E)$ is a binary relation, called flow relation.

The set C is called conditions and the set E is called events.

Def.: Let N be a net and let $x \in (C \cup E)$. ${}^\bullet x := \{y | yFx\}$ is called the set of preconditions of x and $x^\bullet := \{y | xFy\}$ is called the set of postconditions of x.

This definition is mostly used for the case of $x \in E$, but it applies also to the case of $x \in C$.

Def.: Let $(c, e) \in C \times E$.

1 (c, e) is called a **loop**, if $cFe \wedge eFc$.

2 N is called **pure**, if F does not contain any loops (see fig. 2.32, left).

Def.: A net is called simple, if no two transitions t_1 and t_2 have the same set of pre- and postconditions.

Figure 2.32. Nets which are not pure (left) and not simple (right)

Simple nets with no isolated elements meeting some additional restrictions are called **condition/event nets**. Condition/event nets are a special case of bipartite graphs (graphs with two disjoint sets of nodes). We will not discuss those additional restrictions in detail since we will consider more general classes of nets in the following.

2.6.3 Place/transition nets

For condition/event nets, there is at most one token per condition. For many applications, it is useful to remove this restriction and to allow more tokens per conditions. Nets allowing more than one token per condition are called place/transition nets. Places correspond to what we so far called conditions and

transitions correspond to what we so far called events. The number of tokens per place is called a **marking**. Mathematically, a marking is a mapping from the set of places to the set of natural numbers extended by a special symbol ω denoting infinity.

Let $\mathbb{N}_0$ denote the natural numbers including 0. Then, formally speaking, place/transition nets can be defined as follows:

Def.: (P,T,F,K,W,M_0) is called a place/transition net $\Longleftrightarrow$

1 $N = (P,T,F)$ is a net with places $p \in P$ and transitions $t \in T$.

2 Mapping $K : P \to (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}$ denotes the capacity of places (ω symbolizes infinite capacity).

3 Mapping $W : F \to (\mathbb{N}_0 \setminus \{0\})$ denotes the weight of graph edges.

4 Mapping $M_0 : P \to \mathbb{N}_0 \cup \{\omega\}$ represents the initial marking of places.

Edge weights affect the number of tokens that are required before transitions can happen and also identify the number of tokens that are generated if a certain transition takes place. Let $M(p)$ denote a current marking of place $p \in P$ and let $M'(p)$ denote a marking after some transition $t \in T$ took place. The weight of edges belonging to preconditions represents the number of tokens that are removed from places in the precondition set. Accordingly, the weight of edges belonging to the postconditions represents the number of tokens that are added to the places in the postcondition set. Formally, marking M' is computed as follows:

$$M'(p) = \begin{cases} M(p) - W(p,t), & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + W(t,p), & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) - W(p,t) + W(t,p), & \text{if } p \in {}^\bullet t \cap t^\bullet \\ M(p) & \text{otherwise} \end{cases}$$

Fig. 2.33 shows an example of how transition t_j affects the current marking.

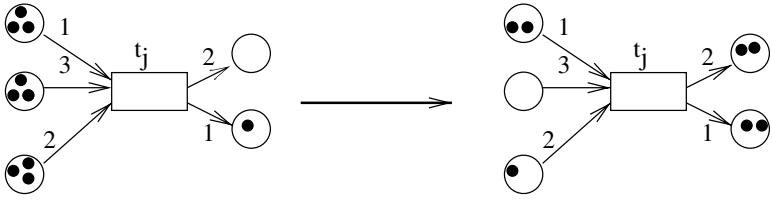

Figure 2.33. Generation of a new marking

By default, unlabeled edges are considered to have a weight of 1 and unlabeled places are considered to have unlimited capacity ω.

We now need to explain the two conditions that must be met before a transition $t \in T$ can take place:

- for all places p in the precondition set, the number of tokens must at least be equal to the weight of the edge from p to t and
- for all places p in the postcondition set, the capacity must be large enough to accommodate the new tokens which t will generate.

Transitions meeting these two conditions are called **M-activated**. Formally, this can be defined as follows:

Def.: Transition $t \in T$ is said to be M-activated $\Longleftrightarrow$

$$(\forall p \in {}^\bullet t : M(p) \geq W(p,t)) \wedge (\forall p \in t^\bullet : M(p) + W(t,p) \leq K(p))$$

Activated transitions can happen, but they do not need to. If several transitions are activated, the sequence in which they happen is not deterministically defined.

For place/transition nets, there are standard techniques for generating proofs of system properties. For example, there may be subsets of places for which the total number of tokens does not change, no matter which transition fires [Reisig, 1985]. Such subsets of places are called place invariants. For example, the number of trains commuting between Cologne and Paris does not change in our railway example. The same is true for the trains traveling between Amsterdam and Paris. Computing such invariants can be the standard point for verifying required system properties such as mutual exclusion.

2.6.4 Predicate/transition nets

Condition/event nets as well as place/transition nets can quickly become very large for large examples. A reduction of the size of the nets is frequently possible with predicate/transition nets. We will demonstrate this, using the so-called "dining philosophers problem" as an example. The problem is based on the assumption that a set of philosophers is dining at a round table. In front of each philosopher, there is a plate containing spaghetti. Between each of the plates, there is just one fork (see fig. 2.34). Each philosopher is either eating or thinking. Eating philosophers need their two adjacent forks for that, so they can only eat if their neighbors are not eating.

This situation can be modeled as a condition/event net, as shown in fig. 2.35. Conditions t_j correspond to the thinking states, conditions e_j correspond to the eating states, and conditions f_j represent available forks.

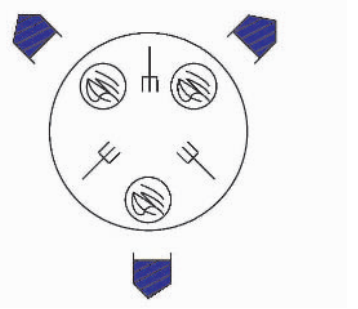

Figure 2.34. The dining philosophers problem

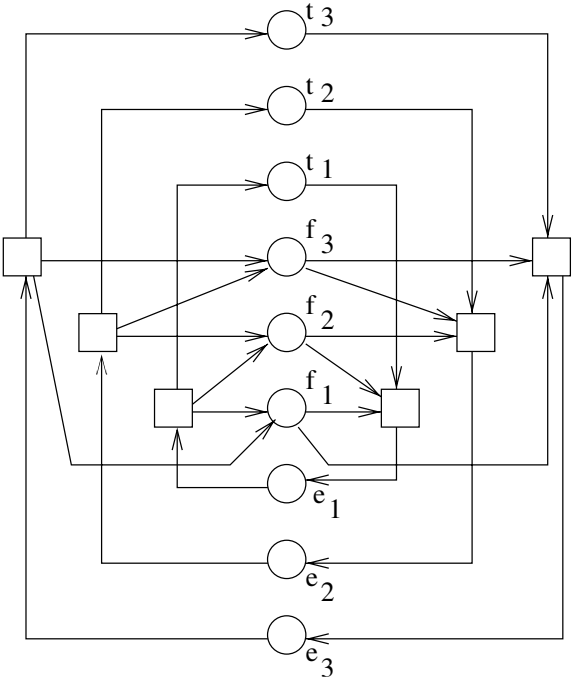

Figure 2.35. Place/transition net model of the dining philosophers problem

Considering the small size of the problem, this net is already very large. The size of this net can be reduced by using predicate/transition nets. Fig. 2.36 is a model of the same problem as a predicate/transition net.

With predicate/transition nets, tokens have an identity and can be distinguished. We use this in fig. 2.36 in order to distinguish between the three different philosophers p_1 to p_3 and to identify fork f_3. Furthermore, edges can be labeled with variables and functions. In the example, we use variables to represent the

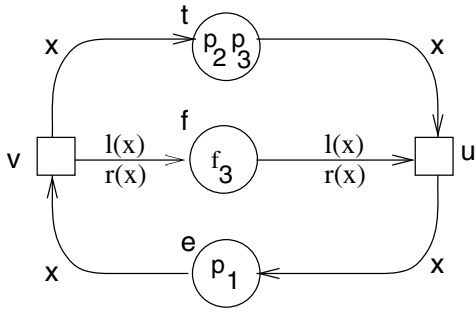

Figure 2.36. Predicate/transition net model of the dining philosophers problem

identity of philosophers and functions l(x) and r(x) to denote the left and right forks of philosopher x, respectively. These two forks are required as a precondition for transition u and returned as a postcondition by transition v. Note that this model can be easily extended to the case of $n > 3$ philosophers. We just need to add more tokens. In contrast to the net in fig. 2.35, the structure of the net does not have to be changed.

2.6.5 Evaluation

The key advantage of Petri nets is their power for modeling causal dependencies. Standard Petri nets have no notion of time and all decisions can be taken locally, by just analyzing transitions and their pre- and postconditions. Therefore, they can be used for modeling geographically distributed systems. Furthermore, there is a strong theoretical foundation for Petri nets, simplifying formal proofs of systems properties.

In certain contexts, their strength is also their weakness. If time is to be explicitely modeled, standard Petri nets cannot be used. Furthermore, standard Petri nets have no notion of hierarchy and no programming language elements, let alone object oriented features. In general, it is difficult to represent data.

There are extended versions of Petri nets avoiding the mentioned weaknesses. However, there is no universal extended version of Petri nets meeting all requirements mentioned at the beginning of this chapter. Nevertheless, due to the increasing amount of distributed computing, Petri nets became more popular than they were initially.

2.7 Message Sequence Charts

Message sequence charts (MSCs) provide a graphical means for representing schedules. MSCs use one dimension (typically the vertical dimension) for

representing time, and the other dimension for representing geographical distribution.

MSCs provide the right means for visualizing schedules of trains or busses. Fig. 2.37 is an example. This example also refers to trains between Amsterdam, Cologne, Brussels and Paris. Aachen is included as an intermediate stop between Cologne and Brussels. Vertical segments correspond to times spent at stations. For one of the trains, there is a timing overlap between the trains coming from Cologne and Amsterdam at Brussels. There is a second train which travels between Paris and Cologne which is not related to an Amsterdam train.

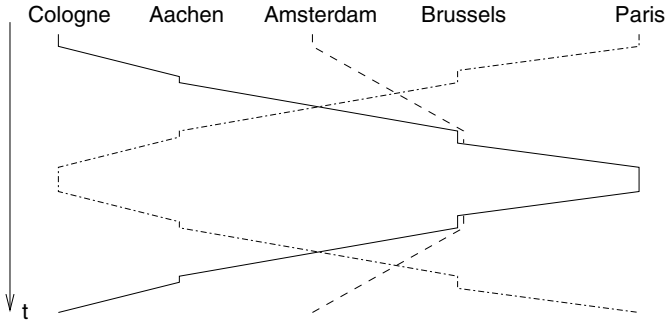

Figure 2.37. Message sequence diagram

A more realistic example is shown in fig. 2.38. This example [Huerlimann, 2003] describes simulated Swiss railway traffic in the Lötschberg area. Slow and fast trains can be distinguished by their slope in the graph. The figure includes information about the time of the day. In this context, the diagram is called time/distance diagram.

MSCs are appropriate means for representing typical schedules. However, they fail to provide information about necessary synchronization. For example, in the presented example it is not known whether the timing overlap at Brussels happens coincidentally or whether some real synchronization for connecting trains is required. Furthermore, permissible deviations from the presented schedule (min/max timing behavior) can hardly be included in these charts.

2.8 UML

All the languages presented so far require a rather precise knowledge about the behavior of the system to be specified. Frequently, and especially during the early specification phases, such knowledge is not available. Very first ideas about systems are frequently sketched on “napkins” or “envelopes”. Support for a more systematic approach to these first phases in a design process is the goal of the major so-called UML standardization effort. UML [OMG, 2005],

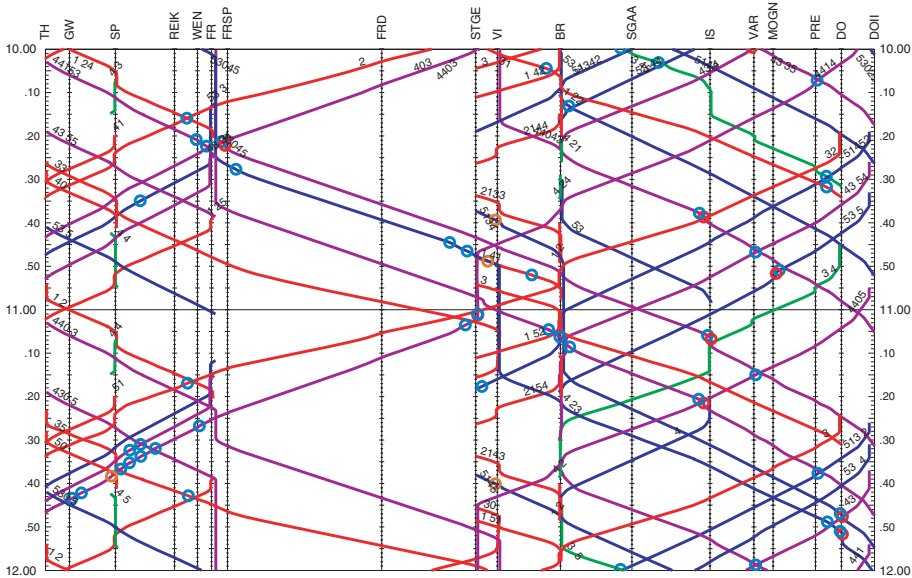

Figure 2.38. Railway traffic displayed by a message sequence diagram (courtesy H. Brändli, IVT, ETH Zürich), ©ETH Zürich

[Fowler and Scott, 1998] stands for "Unified Modeling Language". UML was designed by leading software technology experts and is supported by commercial tools. UML primarily aims at the support of the software design process. UML contains a large number of diagram types and it is, by itself, a complex graphical language. Fortunately, most of the diagram types are variants of those graphical languages which we have already introduced in this book.

Version 1.4 of UML was not designed for embedded systems. Therefore, it lacks a number of features required for modeling embedded systems (see page 13). In particular, the following features are missing [McLaughlin and Moore, 1998]:

- the partitioning of software into tasks and processes cannot be modeled,
- timing behavior cannot be described at all,
- the presence of essential hardware components cannot be described.

Due to the increasing amount of software in embedded systems, UML is gaining importance for embedded systems as well. Hence, several proposals for UML extensions to support real-time applications have been made [McLaughlin and Moore, 1998], [Douglass, 2000]. These extensions have been consid-

ered during the design of UML 2.0. UML 2.0 includes 13 diagram types (up from nine in UML 1.4) [Ambler, 2005]:

- **Sequence diagrams:** Sequence diagrams are variants of message sequence charts. Fig. 2.39 shows an example (based on an example from Gentleware AG [Poseidon, 2003]).

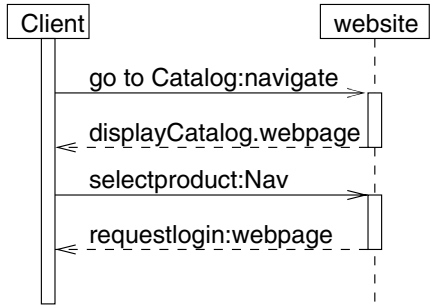

Figure 2.39. Segment from an UML sequence diagram

  One of the key distinction between the type of diagrams shown in figs. 2.38 and 2.39 is that fig. 2.39 does not include any reference to real time. UML version 1.4 was not designed for real-time applications. Some of the restrictions of UML 1.4 have been removed in UML 2.0.

- **State machine diagrams** (called **State Diagrams** in version 1 of UML): UML includes a variation of StateCharts and hence allows modeling state machines.

- **Activity diagrams**: In essence, activity diagrams are extended Petri nets. Extensions include symbols denoting decisions (just like in ordinary flow charts). The placement of symbols is somewhat similar to SDL. Fig. 2.40 shows an example.

  The example shows the procedure to be followed during a standardization process. Forks and joins of control correspond to transitions in Petri nets and they use the symbols (horizontal bars) that were initially used for Petri nets as well. The diamond at the bottom shows the symbol used for decisions. Activities can be organized into "swim-lanes" (areas between vertical dotted lines) such that the different responsibilities and the documents exchanged can be visualized.

- **Deployment diagram**: These diagrams are important for embedded systems: they describe the "execution architecture" of systems (hardware or software nodes).

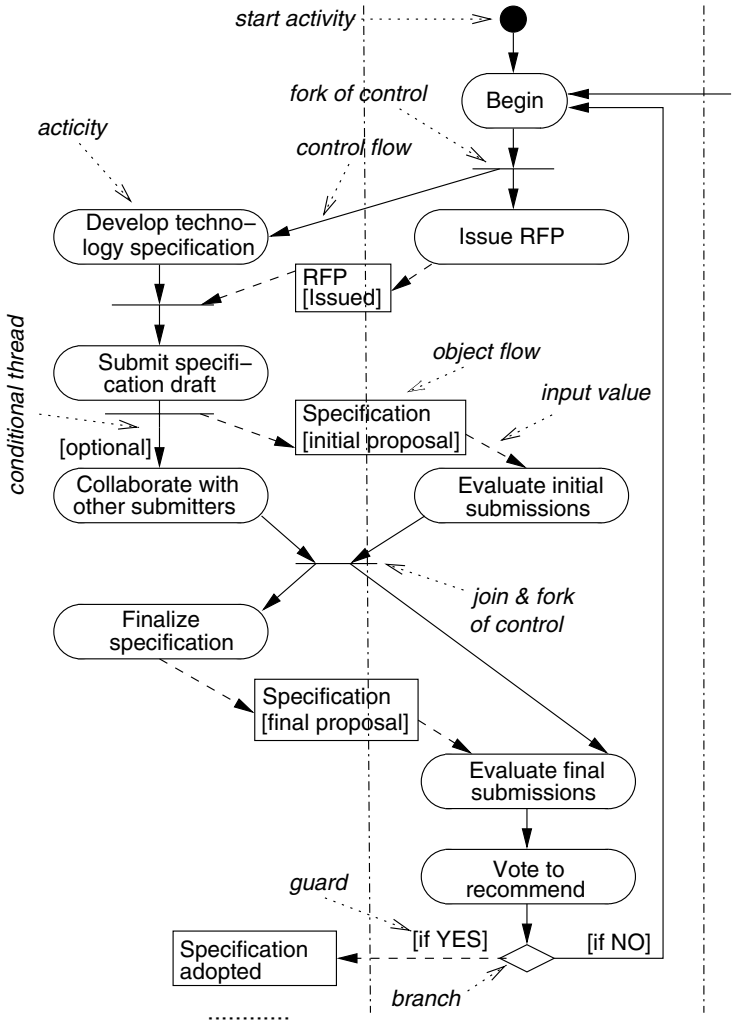

Figure 2.40. Activity diagram [Kobryn, 2001]

- **Package diagrams**: Package diagrams represent the partitioning of software into software packages. They are similar to module charts in StateMate.

- **Use case diagrams**: These diagrams capture typical application scenarios of the system to be designed. For example, fig. 2.41 [Ambler, 2005] shows scenarios for customers of some bank.

- **Class diagrams**: These diagrams describe inheritance relations of object classes.

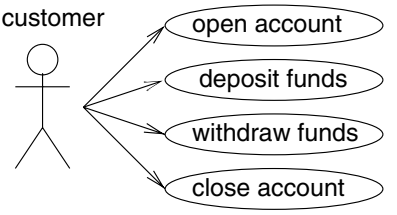

Figure 2.41. Use case example

- **Timing diagrams**: They can be used to show the change of the state of an object over time.

- **Communication diagram** (called **Collaboration diagrams** in UML 1.x): These graphs represent classes, relations between classes, and messages that are exchanged between them.

- **Component diagrams:** They represent the components used in applications or systems.

- **Object diagrams**, **interaction overview diagrams**, **composite structure diagrams**: This list consists of three types of diagrams which are less frequently used. Some of them may actually be special cases of other types of diagrams.

Currently available tools, for example from ilogix (see http://www.ilogix.com), provide some consistency checking between the different diagram types. Complete checking, however, seems to be impossible. One reason for this is that the semantics of UML initially was left undefined. It has been argued that this was done intentionally, since one does not like to bother about the precise semantics during the early phases of the design. As a consequence, precise, executable specifications can only be obtained if UML is combined with some other, executable language. Available design tools have combined UML with SDL [Telelogic, 1999] and C++. There are, however, also some first attempts to define the semantics of UML.

In this book, we will not discuss UML in further detail, since all the relevant diagram types have already been described. Nevertheless, it is interesting to note how a technique like Petri nets was initially certainly not a mainstream technique. Decades after its invention, it has become a frequently applied technique due to its inclusion in UML.

2.9 Process networks

2.9.1 Task graphs

Process networks have already been mentioned in the context of computational models. Process networks are modeled with graphs. We will use the names **task graphs** and **process networks** interchangeably, even though these terms were created by different communities. Nodes in the graph represent processes performing operations. Processes map input data streams to output data streams. Processes are often implemented in high-level programming languages. Typical processes contain (possibly non-terminating) iterations. In each cycle of the iteration, they consume data from their inputs, processes the data received, and generate data on their output streams. Edges represent relations between processes. We will now introduce these graphs at a more detailed level.

The most obvious relation between processes is their causal dependence: Many processes can only be executed after other processes have terminated. This dependence is typically captured in **dependence graphs**. Fig. 2.42 shows a dependence graph for a set of processes or tasks.

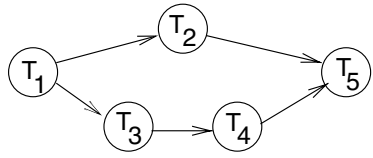

Figure 2.42. Dependence graph

Def.: A dependence graph is a directed graph $G = (V, E)$ in which $E \subseteq V \times V$ is a partial order. If $(v_1, v_2) \in E$, then v_1 is called an **immediate predecessor** of v_2 and v_2 is called an **immediate successor** of v_1. Suppose E^* is the transitive closure of E. If $(v_1, v_2) \in E^*$, then v_1 is called a **predecessor** of v_2 and v_2 is called a **successor** of v_1.

Such dependence graphs form a special case of **task graphs**. Task graphs represent relations between a set of processes. Task graphs may contain more information than modeled in the dependence graph in fig. 2.42. For example, task graphs may include the following extensions of dependence graphs:

1 **Timing information:** Tasks may have arrival times, deadlines, periods, and execution times. In order to take these into account while scheduling tasks, it may be useful to include this information in the task graphs. Adopting the notation used in the book by Liu [Liu, 2000], we include possible execution intervals in fig. 2.43. Tasks T_1 to T_3 are assumed to be independent.

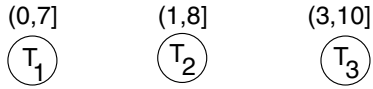

Figure 2.43. Task graphs including timing information

Significantly more complex combinations of timing and dependence relations can exist.

2 **Distinction between different types of relations** between tasks: Precedence relations just model constraints for possible execution sequences. At a more detailed level, it may be useful to distinguish between constraints for scheduling and communication between tasks. Communication can again be described by edges, but additional information may be available for each of the edges, such as the time of the communication and the amount of information exchanged. Precedence edges may be kept as a separate type of edges, since there could be situations in which processes have to execute sequentially even though they do not exchange information.

In fig. 2.42, input and output (I/O) is not explicitly described. Implicitly it is assumed that tasks without any predecessor in the task graph might be receiving input at some time. Also, it is assumed that they generate output for the successor task and that this output is available only after the task has terminated. It is often useful to describe input and output more explicitly. In order to do this, another kind of relation is required. Using the same symbols as Thoen [Thoen and Catthoor, 2000], we use partially filled circles for denoting input and output. In fig. 2.44, filled circles identify I/O edges.

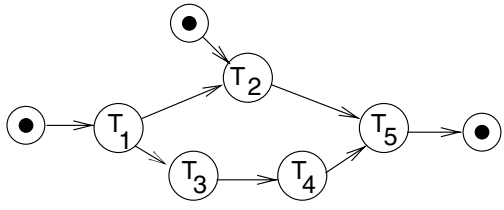

Figure 2.44. Task graphs including I/O-nodes and edges

3 **Exclusive access to resources:** Tasks may be requesting exclusive access to some resource, for example to some input/output device or some communication area in memory. Information about necessary exclusive access should be taken into account during scheduling. Exploiting this information might, for example, be used to avoid the priority inversion problem

(see page 141). Information concerning exclusive access to resources can be included in task graphs.

4 **Periodic schedules:** Many tasks, especially in digital signal processing, are periodic. This means that we have to distinguish more carefully between a task and its execution (the latter is frequently called a **job** [Liu, 2000]). Task graphs for such schedules are infinite. Fig. 2.45 shows a task graph including jobs J_{n-1} to J_{n+1} of a periodic task.

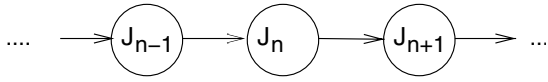

Figure 2.45. Task graph including jobs of a periodic task

5 **Hierarchical graph nodes:** The complexity of the computations denoted by graph nodes may be quite different. On one hand, specified programs may be quite large and contain thousands of lines of code. On the other hand, programs can be split into small pieces of code so that in the extreme case, each of the nodes corresponds only to a single operation. The level of complexity of graph nodes is also called their **granularity**. Which granularity should be used? There is no universal answer to this. For some purposes, the granularity should be as large as possible. For example, if we consider each of the nodes as one process to be scheduled by the RTOS, it may be wise to work with large nodes in order to minimize context-switches between different processes. For other purposes, it may be better to work with nodes modeling just a single operation. For example, nodes will have to be mapped to hardware or to software. If a certain operation (like the frequently used Discrete Cosine Transform, or DCT) can be mapped to special purpose hardware, then it should not be buried in a complex node that contains many other operations. It should rather be modeled as its own node. In order to avoid frequent changes of the granularity, hierarchical graph nodes are very useful. For example, at a high hierarchical level, the nodes may denote complex tasks, at a lower level basic blocks and at an even lower level individual arithmetic operations. Fig. 2.46 shows a hierarchical version of the dependence graph in fig. 2.42, using a rectangle to denote a hierarchical node.

A very comprehensive task graph model, called **multi-thread graph** (MTG), was proposed by Thoen [Thoen and Catthoor, 2000]. MTGs are defined as follows:

Def.: A multi-thread graph M is defined as an 11-tuple $(O, E, V, D, \vartheta, \iota, \Lambda, \mathcal{E}^{lat}, \mathcal{E}^{resp}, \nabla^{i}, \nabla^{av})$ where

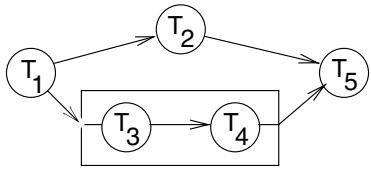

Figure 2.46. Hierarchical task graph

- O is the set of operation nodes. They can be of different types, including *thread, hierarchical thread, or, event, synchro, semaphore, source* and *sink*. MTGs have single sources and sinks of type *source* and *sink*, respectively. Nodes of type *or* allow modeling situations in which only one of a set of tasks is required in order to start the next task. *Events* model external input. *Semaphores* can be used to model mutual exclusion. *Synchro* nodes provide acknowledgments to the environment.
- E is the set of control edges. Attributes of control edges include timing information like production and consumption rate.
- V, D, and ϑ refer to the access of variables, not discussed in detail in this text.
- ι is the set of input/output nodes,
- Λ associates execution latency intervals with all threads,
- $\mathcal{E}^{lat}$, $\mathcal{E}^{resp}$, ∇^{i} and ∇^{av} are timing constraints.

As can be seen from the definition, almost all of the presented extensions of simple precedence graphs are included in MTGs. MTGs are used for the work described starting at page 191.

2.9.2 Asynchronous message passing

For asynchronous message passing, communication between processes is buffered. Typically, buffers are assumed to be FIFOs of theoretically unbounded length.

2.9.2.1 Kahn process networks

Kahn process networks (KPN) [Kahn, 1974] are a special case of such process networks. For KPN, writes are non-blocking, whereas reads block whenever an attempt is made to read from an empty FIFO queue. There is no other way for communication between processes except through FIFO-queues. Only a

single process is allowed to read from a certain queue. So, if output data has to be sent to more than a single process, duplication of data must be done inside processes. In general, Kahn processes require scheduling at run-time, since it is difficult to predict their precise behavior over time. The question of whether or not all finite-length FIFOs are sufficient for an actual KPN model is undecidable in the general case. Nevertheless, practically useful algorithms exist [Kienhuis et al., 2000].

2.9.2.2 Synchronous data flow

The synchronous data flow (SDF) model [Lee and Messerschmitt, 1987] can best be introduced by referring to its graphical notation. Fig. 2.47 (left) shows a synchronous data flow graph. The graph is a directed graph, nodes A and B denote computations * and +. SDF graphs, like all data flow graphs, show computations to be performed and their dependence, but not the order in which the computations have to be performed (in contrast to specifications in sequential languages such as C). Inputs to SDF graphs are assumed to consist of an infinite stream of samples. Nodes can start their computations when their inputs are available. Edges must be used whenever there is a data dependency between any two nodes.

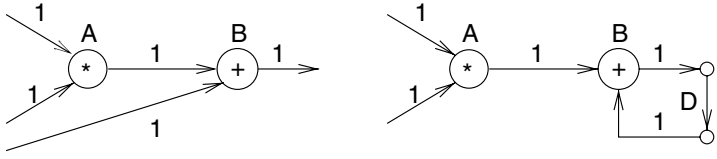

Figure 2.47. Graphical representations of synchronous data flow

For each execution, the computation in a node is called a firing. For each firing, a number of tokens, representing data, is consumed and produced. In synchronous data flow, the number of tokens produced or consumed in one firing is constant. Constant edge labels denote the corresponding numbers of tokens. These constants facilitate the modeling of multi-rate signal processing applications (applications for which certain signals are generated at frequencies that are multiples of other frequencies). The term **synchronous** data flow reflects the fact that tokens are consumed from the incoming arcs in a synchronous manner (all at the same instant in time). The term **asynchronous** message passing reflects the fact that tokens can be buffered using FIFOs. The property of producing and consuming a constant number of tokens makes it possible to determine execution order and memory requirements at compile time. Hence, complex run-time scheduling of executions is avoided. SDF graphs may include delays, denoted by the symbol D on an edge (see fig. 2.47

(right)). SDF graphs can be translated into periodic schedules for mono- as well as for multi-processor systems (see e.g. [Pino and Lee, 1995]]. A legal schedule for the simple example of fig. 2.47 would consist of the sequence (A, B) (repeated forever). A sequence (A, A, B) (A executed twice as many times as B) would be illegal, since it would accumulate an infinite number of tokens on the implicit FIFO buffer between A and B.

2.9.3 Synchronous message passing

2.9.3.1 CSP

CSP [Hoare, 1985] (*communicating sequential processes*) is one of the first languages comprising mechanisms for interprocess communication. Communication is based on channels.

Example:

```
process A                          process B
.....                              ......
var a ..                           var b ...
 a := 3;                            ...
 c!a; - - output to channel c      c?b; - - input from channel c
end;                               end;
```

Both processes will wait for the other process to arrive at the input or output statement. This form of communication is called **rendez-vous concept** or **blocking communication**.

CSP has laid the foundation for the OCCAM language that was proposed as a programming language of the **transputer** [Thiébaut, 1995].

2.9.3.2 ADA

During the eighties, the Department of Defense (DoD) in the US realized that the dependability and maintainability of the software in its military equipment could soon become a major source of problems, unless some strict policy was enforced. It was decided that all software should be written in the same real-time language. Requirements for such a language were formulated. No existing language met the requirements and, consequently, the design of a new one was started. The language which was finally accepted was based on PASCAL. It was called ADA (after Ada Lovelace, who can be considered being the first (female) programmer). ADA'95 [Kempe, 1995], [Burns and Wellings, 2001] is an object-oriented extension of the original standard.

One of the interesting features of ADA is the ability to have nested declarations of processes (called tasks in ADA). Tasks are started whenever control

passes into the scope in which they are declared. The following is an example (according to Burns et al. [Burns and Wellings, 1990]):

```
procedure example1 is
  task a;
  task b;
  task body a is
    -- local declarations for a
    begin
      -- statements for a
    end a;
  task body b is
    -- local declarations for b
    begin
      -- statements for b
    end b;
  begin
   -- Tasks a and b will start before the first
   -- statement of the body of example1
  end;
```

The communication concept of ADA is another key concept. It is based on the *rendez-vous* paradigm. Whenever two tasks want to exchange information, the task reaching the "meeting point" first has to wait until its partner has also reached a corresponding point of control. Syntactically, procedures are used for describing communication. Procedures which can be called from other tasks have to be identified by the keyword **entry**. Example [Burns and Wellings, 1990]:

```
task screen_out is
  entry call (val : character; x, y : integer);
end screen_out;
```

Task screen_out includes a procedure named call which can be called from other processes. Some other task can call this procedure by prefixing it with the name of the task:

```
screen_out.call('Z',10,20);
```

The calling task has to wait until the called task has reached a point of control, at which it accepts calls form other tasks. This point of control is indicated by the keyword **accept**:

```
task body screen_out is
  ...
   begin
    accept call (val : character; x, y : integer) do
   ...
   end call;
   ...
end screen_out;
```

Obviously, task screen_out may be waiting for several calls at the same time. The ADA **select**-statement provides this capability. Example:

```
task screen_output is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
end screen_out;
task body screen_output is
...
  select
   accept call_ch ...  do...
   end call_ch;
  or
   accept call_int ... do ..
   end call_int;
  end select; ...
```

In this case, task screen_out will be waiting until either call_ch or call_int are called.

ADA is the language of choice for almost all military equipment produced in the Western hemisphere.

Again, process networks are not explictly represented as graphs, but these graphs can be generated from the textual representation.

2.10 Java

Java was designed as a platform-independent language. It can be executed on any machine for which an interpreter of the internal byte-code representation of Java-programs is available. This byte-code representation is a very compact representation, which requires less memory space than a standard binary machine code representation. Obviously, this is a potential advantage in system-on-a-chip applications, where memory space is limited.

Also, Java was designed as a safe language. Many potentially dangerous features of C or C++ (like pointer arithmetic) are not available in Java. Hence, Java meets the safety requirements for specification languages for embedded systems. Java supports exception handling, simplifying recovery in case of run-time errors. There is no danger of memory leakages due to missing memory deallocation, since Java provides automatic garbage collection. This feature avoids potential problems in applications that have to run for months or even years without ever being restarted. Java also meets the requirement to support concurrency since it includes threads (light-weight processes).

In addition, Java applications can be implemented quite fast, since Java supports object orientation and since Java development systems come with powerful libraries.

However, standard Java is not really designed for real-time systems and a number of characteristics which would make it a real-time programming language are missing:

- The size of Java run-time libraries has to be added to the size of the application itself. These run-time libraries can be quite large. Consequently, only really large applications benefit from the compact representation of the application itself.

- For many embedded applications, direct control over I/O devices is necessary (see page 16). For safety reasons, no direct control over I/O devices is available in standard Java.

- Automatic garbage collection requires some computing time. In standard Java, the instance in time at which automatic garbage collection is started cannot be predicted. Hence, the worst case execution time is very difficult to predict. Only extremely conservative estimates can be made.

- Java does not specify the order in which threads are executed if several threads are ready to run. As a result, worst-case execution time estimates must be even more conservative.

First proposals for solving the problems were made by Nilsen. Proposals include hardware-supported garbage-collection, replacement of the run-time scheduler and tagging of some of the memory segments [Nilsen, 2004].

In 2003, relevant Java programming environments included the Java Enterprise Edition (J2EE), the Java Standard Edition (J2SE), the Java Micro Edition (J2ME), and CardJava. CardJava is a stripped-down version of Java with emphasis on security for SmartCard applications. J2ME is the relevant Java environment for all other types of embedded systems. Two library profiles have been defined for J2ME: CDC and CLDC. CLDC is used for mobile phones, using the so-called MIDP 1.0/2.0 as its standard for the application programming interface (API). CDC is used, for example, for TV sets and powerful mobile phones. The currently relevant real-time extension of Java is called "Real-time specification for Java (JSR-1)" [Java Community Process, 2002] and is supported by TimeSys [TimeSys Inc., 2003].

2.11 VHDL

2.11.1 Introduction

Languages for describing hardware, such as VHDL, are called **hardware description languages** (HDLs). Up to the eighties, most design systems used graphical HDLs. The most common building block was a gate. However, in addition to using graphical HDLs, we can also use textual HDLs. The strength of textual languages is that they can easily represent complex computations including variables, loops, function parameters and recursion. Accordingly, when digital systems became more complex in the eighties, textual HDLs almost completely replaced graphical HDLs. Textual HDLs were initially a research topic at Universities. See Mermet et al. [Mermet et al., 1998] for a survey of languages designed in Europe in the eighties. MIMOLA was one of these languages and the author of this book contributed to its design and applications [Marwedel and Schenk, 1993]. Textual languages became popular when VHDL and its competitor Verilog (see page 75) were introduced. VHDL was designed in the context of the VHSIC program of the Department of Defense (DoD) in the US. VHSIC stands for *very high speed integrated circuits*[4]. Initially, the design of VHDL (VHSIC hardware description language) was done by three companies: IBM, Intermetrics and Texas Instruments. A first version of VHDL was published in 1984. Later, VHDL became an IEEE standard, called IEEE 1076. The first IEEE version was standardized in 1987; updates were designed in 1992, in 1997 and in 2002.

[4]The design of the Internet was also part of the VHSIC program.

A key distinction between common software languages and hardware description languages is the need to describe concurrency among different hardware components. VHDL uses **processes** for doing this. Each process models one component of the potentially concurrent hardware. For simple hardware components, a single process may be sufficient. More complex components may need several processes for modeling their operation. Processes communicate through **signals**. Signals roughly correspond to physical connections (wires). Another distinction between software languages and HDLs comes from the need to model time. VHDL, like all other HDLs, includes the necessary support.

The design of VHDL used ADA as the starting point, since both languages were designed for the DoD. Since ADA is based on PASCAL, VHDL has some of the syntactical flavor of PASCAL. However, the syntax of VHDL is much more complex and it is necessary not to get distracted by the syntax. In the current book, we will just focus on some concepts of VHDL which are useful also in other languages. A full description of VHDL is beyond the scope of this book. The entire standard is available from IEEE (see [IEEE, 1992]).

2.11.2 Entities and architectures

In VHDL, each unit to be modeled is called a **design entity** or a **VHDL entity**. Design entities are composed of two types of ingredients: an **entity declaration** and one (or several) **architectures** (see fig. 2.48). For each entity, the most recently analyzed architecture will be used by default. Using other architectures can be specified.

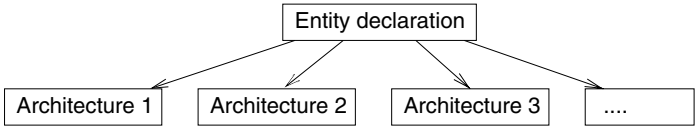

Figure 2.48. An entity consists of an entity declaration and architectures

We will consider a full adder as an example. Full adders have three input ports and two output ports (see fig. 2.49).

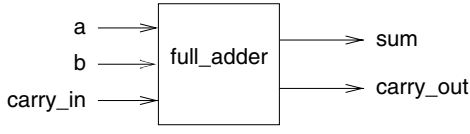

Figure 2.49. Full-adder and its interface signals

An entity declaration corresponding to fig. 2.49 is the following:

```
entity full_adder is              -- entity declaration
   port (a, b, carry_in: in Bit;  -- input ports
         sum, carry_out: out Bit); -- output ports
end full_adder;
```

Architectures consist of architecture headers and architectural bodies. We can distinguish between different styles of bodies, in particular between structural and behavioral bodies. We will show how the two are different using the full adder as an example. Behavioral bodies include just enough information to compute output signals from input signals and the local state (if any), including the timing behavior of the outputs. The following is an example of this (<= denotes assignments to signals):

```
architecture behavior of full_adder is  -- architecture
begin
   sum       <= (a xor b) xor carry_in after 10 Ns;
   carry_out <= (a and b) or (a and carry_in) or
                (b and carry_in) after 10 Ns;
end behavior;
```

VHDL-based simulators are capable of displaying output signal waveforms resulting from stimuli applied to the inputs of the full adder described above.

In contrast, structural bodies describe the way entities are composed of simpler entities. For example, the full adder can be modeled as an entity consisting of three components (see fig. 2.50). These components are called i1 to i3 and are of type half_adder or or_gate.

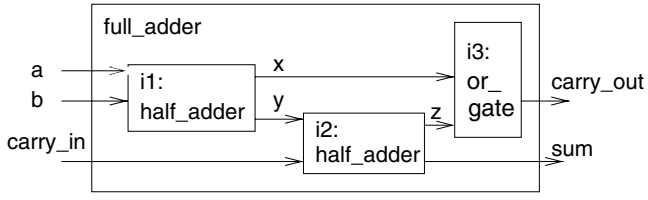

Figure 2.50. Schematic describing structural body of the full adder

In the 1987 version of VHDL, these components must be declared in a so-called component declaration. This declaration is very similar (and it serves the same purpose) as forward declarations in other languages. This declaration provides the necessary information about the component even if the full description of that component is not yet stored in the VHDL data base (this may

happen in the case of so-called top-down designs). From the 1992 version of VHDL onwards, such declarations are not required if the relevant components are already stored in the component data base.

Connections between local component and entity ports are described in **port maps**. The following VHDL code represents the structural body shown in fig. 2.50:

```
architecture structure of full_adder is  -- architecture head
  component half_adder
    port (in1, in2 : in Bit; carry  :out Bit; sum :out Bit);
  end component;
  component or_gate
    port(in1, in2:in Bit; o:out Bit);
  end component;
  signal x, y, z: Bit;        -- local signals
 begin                        -- port map section
  i1: half_adder              -- introduction of half_adder i1
     port map (a, b, x, y);   -- connections between ports
  i2: half_adder port map (y, carry_in, z, sum);
  i3: or_gate    port map (x, z, carry_out);
 end structure;
```

2.11.3 Multi-valued logic and IEEE 1164

In this book, we are restricting ourselves to embedded systems implemented with binary logic. Nevertheless, it may be advisable or necessary to use more than two values for modeling such systems. For example, our systems might contain electrical signals of different strengths and it may be necessary to compute the strength and the logic level resulting from a connection of two or more sources of electrical signals. In the following, we will therefore distinguish between the **level** and the **strength** of a **signal**. While the former is an abstraction of the signal voltage, the latter is an abstraction of the impedance (resistance) of the voltage source. We will be using discrete sets of signal values representing the signal level and the strength. Using discrete sets of strengths avoids the problems of having to solve Kirchhoff's equations and enables us to work with algebraic techniques. We will also model unknown electrical signals by special signal values.

In practice, electronic design systems use a variety of value sets. Some systems allow only two, while others allow 9 or 46. The overall goal of developing discrete value sets is to avoid the problems of solving network equations (e.g. Kirchoff's laws) and still model existing systems with sufficient precision. In the following, we will present a systematic technique for building up value sets and for relating these to each other. We will use the strength of electrical signals as the key parameter for distinguishing between various value sets. A systematic way of building up value sets, called CSA-theory, was presented by Hayes [Hayes, 1982]. We will later show how the standard value set used for most cases of VHDL-based modeling can be derived as a special case.

2.11.3.1 Two logic values (1 signal strength)

In the simplest case, we will start with just two logic values, called '0' and '1'. These two values are considered to be of the same strength. This means: if two wires connect values '0' and '1', we will not know anything about the resulting signal level.

A single signal strength may be sufficient if no two wires carrying values '0' and '1' are connected and no signals of different strength meet at a particular node of electronic circuits.

2.11.3.2 Three and four logic values (2 signal strengths)

In many circuits, there may be instances in which a certain electrical signal is not actively driven by any output. This may be the case, when a certain wire is not connected to ground, the supply voltage or any circuit node.

For example, systems may contain open-collector outputs (see fig. 2.51, left) or tristate outputs (see fig. 2.51, right). Using appropriate input signals, such outputs can be effectively disconnected from a wire[5].

Obviously, the signal strength of disconnected outputs is the smallest strength that we can think of. In particular, the signal strength of 'Z' is smaller than that of '0' and '1'. Furthermore, the signal level of such an output is unknown. This combination of signal strength and signal value is represented by a logic value called 'Z'. If a signal of value 'Z' is connected to another signal, that other signal will always dominate. For example, if two tristate outputs are connected to the same bus and if one output contributes a value of 'Z', the resulting value on the bus will always be the value contributed by the second output (see fig. 2.52).

[5] In practice, pull-up transistors may be depletion transistors and the tri-state outputs may be inverting.

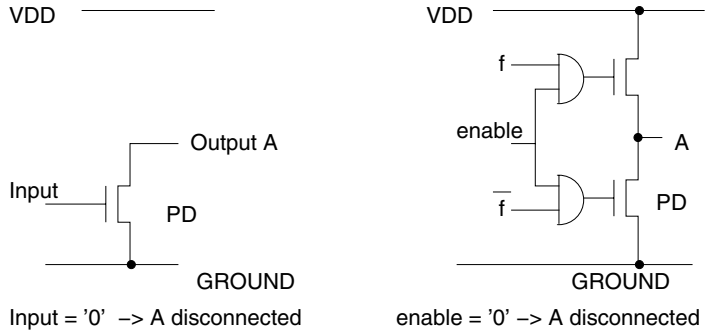

Figure 2.51. Outputs that can be effectively disconnected from a wire

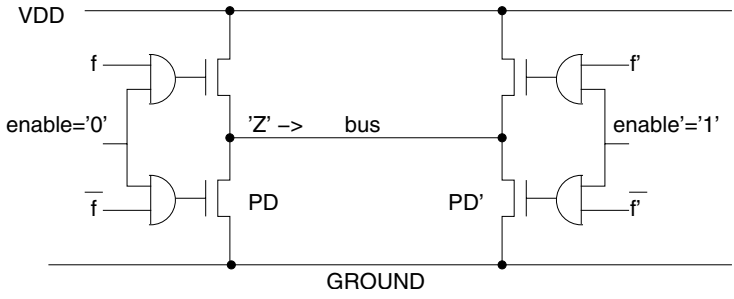

Figure 2.52. Right output dominates bus

In VHDL, each output is associated with a so-called signal **driver**. Computing the value resulting from the contributions of multiple drivers to the same signal is called **resolution** and resulting values are computed by functions called **resolution functions**.

In most, cases three-valued logic sets {'0','1','Z'} are extended by a fourth value called 'X'. 'X' represents an unknown signal level of the same strength as '0' or '1'. More precisely, we are using 'X' to represent unknown values of signals that can be either '0' or '1' or some voltage representing neither '0' nor '1'[6].

The resolution that is required if multiple drivers get connected can be computed very easily, if we make use of a partial order among the four signal values '0', '1', 'Z', and 'X'. The partial order is depicted in fig. 2.53.

Edges in this figure reflect the domination of signal values. Edges define a relation $>$. If $a > b$, then a dominates b. '0' and '1' dominate 'Z'. 'X' dominates

[6]There are other interpretations of 'X', but the one presented above is the most useful one in our context.

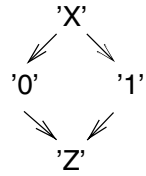

Figure 2.53. Partial order for value set {'0', '1', 'Z', 'X'}

all other signal values. Based on the relation $>$, we define a relation $\geq$. $a \geq b$ holds iff $a > b$ or $a = b$.

We define an operation *sup* on two signals, which returns the **supremum** of the two signal values. The supremum c of the two values a and b is the weakest value for which $c \geq a$ and $c \geq b$ holds. For example, *sup* ('Z', '0')='0', *sup*('Z','1')='1' etc. **The interesting observation is that resolution functions should compute the *sup* function according to the above definition**.

2.11.3.3 Seven signal values (3 signal strengths)

In many circuits, two signal strengths are not sufficient. A common case that requires more values is the use of depletion transistors (see fig. 2.54).

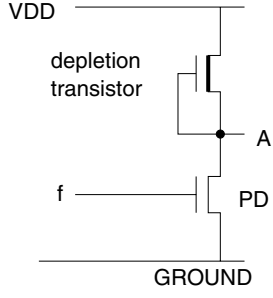

Figure 2.54. Output using depletion transistor

The effect of the depletion transistor is similar to that of a resistor providing a low conductance path to the supply voltage VDD. The depletion transistor as well as the "pull-down transistor" PD act as drivers for node A of the circuit and the signal value at node A can be computed using resolution. The pull-down transistor provides a driver value of '0' or 'Z', depending upon the input to PD. The depletion transistor provides a signal value, which is weaker than '0' and '1'. Its signal level corresponds to the signal level of '1'. We represent the value contributed by the depletion transistor by 'H', and we call it a "weak logic one". Similarity, there can be weak logic zeros, represented by 'L'. The value resulting from the possible connection between 'H' and 'L' is called a "weak

logic undefined", denoted as 'W'. As a result, we have three signal strengths and seven logic values {'0', '1', 'Z', 'X', 'H', 'L', 'W'}. Resolution can again be based on a partial order among these seven values. The corresponding partial order is shown in fig. 2.55.

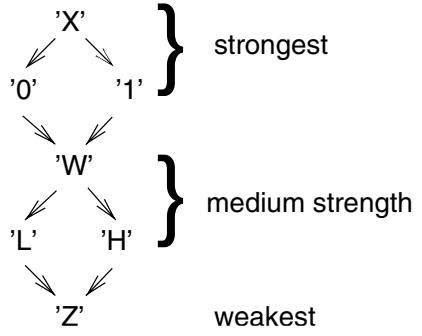

Figure 2.55. Partial order for value set {'0', '1', 'Z', 'X', 'H', 'L', 'W'}

This order also defines an operation *sup* returning the weakest value at least as strong as the two arguments. For example, *sup*('H','0') = '0', *sup*('H','Z') = 'H', *sup*('H','L') = 'W'.

'0' and 'L' represent the same signal levels, but a different strength. The same holds for the pairs '1' and 'H'. Devices increasing the signal strength are called **amplifiers**, devices reducing the signal strength are called **attenuators**.

2.11.3.4 Ten signal values (4 signal strengths)

In some cases, three signal strengths are not sufficient. For example, there are circuits using charges stored on wires. Such wires are charged to levels corresponding to '0' or '1' during some phases of the operation of the electronic circuit. This stored charge can control the (high impedance) inputs of some transistors. However, if these wires get connected to even the weakest signal source (except 'Z'), they loose their charge and the signal value from that source dominates.

For example, in fig. 2.56, we are driving a bus from a specialized output. The bus has a high capacitive load C. While function f is still '0', we set ϕ to '1', charging capacitor C. Then we set ϕ to '0'. If the real value of function f becomes known and it turns out to be '1', we discharge the bus. The key reason for using pre-charging is that charging a bus using an output like the one shown in fig. 2.54 is a slow process, since the resistance of depletion transistors is large. Discharging through regular pull-down transistors PD is a much faster process.

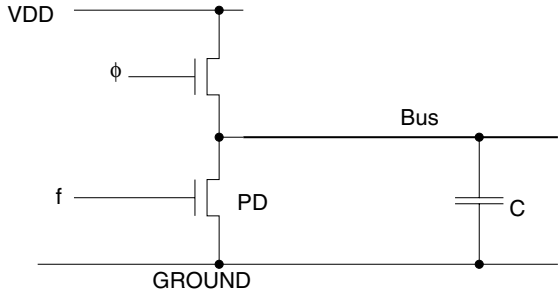

Figure 2.56. Pre-charging a bus

In order to model such cases, we need signal values which are weaker than 'H' and 'L', but stronger than 'Z'. We call such values "very weak signal values" and denote them by 'h' and 'l'. The corresponding very weak unknown value is denoted by 'w'. As a result, we obtain ten signal values {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'h', 'l', 'w'}. Using the signal strength, we can again define a partial order among these values (see fig. 2.57).

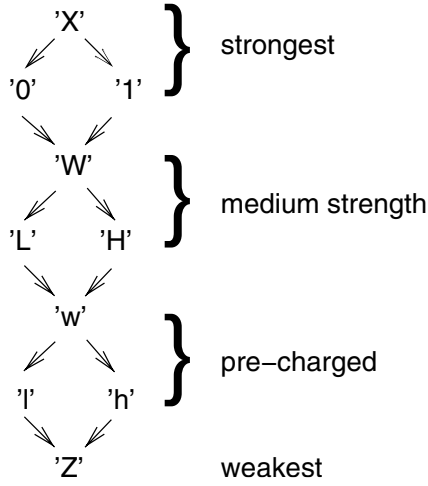

Figure 2.57. Partial order for value set {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'h', 'l', 'w'}

2.11.3.5 Five signal strengths

So far, we have ignored power supply signals. These are stronger than the strongest signals we have considered so far. Signal value sets taking power supply signals into account have resulted in the definition of 46-valued value sets [Coelho, 1989]. However, such models are not very popular.

2.11.3.6 IEEE 1164

In VHDL, there is no predefined number of signal values, except for some basic support for two-valued logic. Instead, the used value sets can be defined in VHDL itself and different VHDL models can use different value sets.

However, portability of models would suffer severely if this capability of VHDL was applied in this way. In order to simplify exchanging VHDL models, a standard value set was defined and standardized by the IEEE. This standard is called IEEE 1164 and is employed in many system models. IEEE 1164 has nine values: {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}. The first seven values correspond to the seven signal values described above. 'U' denotes an uninitialized value. It is used by simulators for signals that have not been explicitely defined.

'-' denotes the **input don't care**. This value needs some explanation. Frequently, hardware description languages are used for describing Boolean functions. The VHDL **select** statement is a very convenient means for doing that. The **select** statement corresponds to **switch** and **case** statements found in other languages and its meaning is different from the select statement in ADA.

Example: Suppose that we would like to represent the Boolean function

$$f(a,b,c) = a\overline{b} + bc$$

Furthermore, suppose that f should be undefined for the case of $a = b = c =$'0'. A very convenient way of specifying this function would be the following:

```
f <= select a & b & c  -- & denotes concatenation
     '1' when "10-"    -- corresponds to first term
     '1' when "-11"    -- corresponds to second term
     'X' when "000"
```

This way, functions given above could be easily translated into VHDL. Unfortunately, the select statement denotes something completely different. Since IEEE 1164 is just one of a large number of possible value sets, it does not include any knowledge about the "meaning" of '-'. Whenever VHDL tools evaluate select statements like the one above, they check if the selecting expression (a & b & c in the case above) is equal to the values in the **when** clauses. In particular, they check if e.g. a & b & c is equal to "10-". In this context, '-' behaves like any other value: VHDL systems check if c has a value of '-'. Since '-' is never assigned to any of the variables, these tests will never be true. Therefore, '-' is of limited benefit. The non-availability of convenient input don't care values is the price that one has to pay for the flexibility of defining value sets in VHDL itself.

The nice property of the general discussion on pages 63 to 67 is the following: it allows us to immediately draw conclusions about the modeling power of IEEE 1164. The IEEE standard is based on the 7-valued value set described on page 65 and, therefore, is capable of modeling circuits containing depletion transistors. It is, however, not capable of modeling charge storage[7].

2.11.4 VHDL processes and simulation semantics

VHDL treats components described above as processes. The syntax used above is just a shorthand for processes. The general syntax for processes is as follows:

```
label : -- optional
process
   declarations -- optional
  begin
   statements-- optional
  end process ;
```

Processes may contain **wait** statements. Such statements can be used to suspend a process. There are the following kinds of **wait** statements:

- **wait on** *signal list*; suspend until one of the signals in the list changes;
- **wait until** *condition*; suspend until *condition* is met, e.g. a = '1';
- **wait for** *duration*; suspend for a specified period of time;
- **wait**; suspend indefinitely.

As an alternative to explicit **wait** statements, a list of signals can be added to the process header. In that case, the process is activated whenever one of the signals in that list changes its value. Example: The following model of an and-gate will execute its body once and will restart from the beginning every time one of the inputs changes its value:

```
process(x, y) begin
   prod <= x  AND y ;
end process;
```

[7] As an exception, if the capability of modeling depletion transistors or pull-up resistors is not needed, one could interpret weak values as stored charges. This is, however, not very practical since pull-up resistors are found in most actual systems.

This model is equivalent to

```
process begin
   prod <= x  AND y ;
   wait on x,y;
end process;
```

According to the original standards document [IEEE, 1997], the execution of a VHDL model is described as follows: *The execution of a model consists of an **initialization phase** followed by the **repetitive execution of process statements** in the description of that model. Each such repetition is said to be a **simulation cycle**. In each cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.*

The initialization phase takes signal initializations into account and executes each process once. It is described in the standards as follows[8]:

At the beginning of initialization, the current time, T_c is assumed to be 0 ns. The initialization phase consists of the following steps:[9]

- *The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation. ...*
- *Each ... process in the model is executed until it suspends. ...*
- *The time of the next simulation cycle (which in this case is the first simulation cycle), T_n is calculated according to the rules of step f of the simulation cycle, below.*

Each simulation cycle starts with setting the current time to the next time at which changes must be considered. This time T_n was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time reaches its maximum, $TIME'HIGH$. According to the original document, the simulation cycle is described as follows: *A simulation cycle consists of the following steps:*

[8] We leave out the discussion of implicitly declared signals and so-called postponed processes introduced in the 1997 version of VHDL.

[9] In order not to get lost in the amount of details provided by the standard, some of its sections (indicated by "...") are omitted in the citation.

a) *The current time, T_c is set equal to T_n. Simulation is complete when $T_n =$ TIME'HIGH and there are no active drivers or process resumptions at T_n.*

b) *Each active explicit signal in the model is updated. (Events may occur as a result.) ...*

   This phrase from the document refers to the fact that in the cycle preceeding the current cycle, new future values for some of the signals have been computed. If T_c corresponds to the time at which these values become valid, they are now assigned. Note that new values of signals are never immediately assigned while executing a simulation cycle. They are not assigned before the next simulation cycle, at the earliest. Signals that change their value generate so-called events which, in-turn, may enable the execution of processes that are sensitive to that signal.

c) *For each process P, if P is currently sensitive to a signal S and if an event has occurred on S in this simulation cycle, then P resumes.*

d) *Each ... process that has resumed in the current simulation cycle is executed until it suspends.*

e) *The time of the next simulation cycle, T_n is determined by setting it to the earliest of*

   1 *TIME'HIGH* (This is the end of simulation time).

   2 *The next time at which a driver becomes active* (this is the next instance in time, at which a driver specifies a new value), *or*

   3 *The next time at which a process resumes* (this time is determined by wait on statements).

   If $T_n = T_c$, then the next simulation cycle (if any) will be a delta cycle.

The iterative nature of simulation cycles is shown in fig. 2.58.

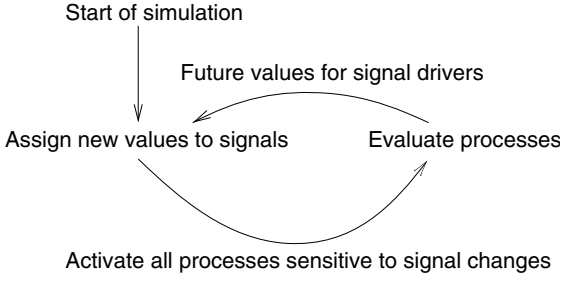

Figure 2.58. VHDL simulation cycles

Delta (δ) simulation cycles have been the source of many discussions. Their purpose is to introduce a infinitesimally small delay even in cases in which the user did not specify any. As an example, we will show the effect of these cycles using a flip-flop as an example. Fig. 2.59 shows the schematic of the flip-flop.

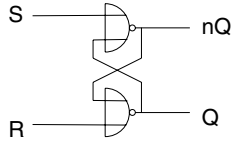

Figure 2.59. RS-Flipflop

The flip-flop is modeled in VHDL as follows:

```
entity RS_Flipflop is
port (R: in    BIT; -- reset
      S: in    BIT; -- set
      Q: inout BIT; -- output
      nQ: inout BIT; -- Q-bar
      );
end RS_Flipflop;
architecture one of RS_Flipflop is
begin
process: (R,S,Q,nQ)
    begin
      Q  <= R nor nQ;
      nQ <= S nor Q;
     end process;
end one;
```

Ports Q and nQ must be of mode inout since they are also read internally, which would not be possible if they were of mode out. Fig. 2.60 shows the simulation times at which signals are updated for this model.

Simulation terminates after two δ cycles. δ cycles correspond to an infinitesimally small unit of time, which will always exist in reality. δ cycles ensure that simulation respects causality and that the results do not depend on the order in which parts of the model are executed by the simulation. Otherwise, simulation would become non-deterministic, which is not what we expect from the simulation of a real circuit with deterministic behavior. There can be arbitrar-

	0ns	0ns+δ	0ns+2 ∗ δ
R	1	1	1
S	0	0	0
Q	1	0	0
nQ	0	0	1

Figure 2.60. δ cycles for RS-flip-flop

ily many δ cycles before the current time T_c is advanced. This possibility of infinite loops can be confusing. One of the options of avoiding this possibility would be to disallow zero delays, which we used in our model of the flip-flop.

A very important concept of VHDL is the separation between the computation of new values for signals and their actual assignment. This separation enables deterministic simulation results. In a model containing the lines

```
a <= b;
b <= a;
```

signals a and b will always be swapped. If the assignments were performed immediately, the result would depend on the order in which we execute the assignments (see also page 25).

2.12 SystemC

Due to the trend of implementing more and more functionality in software, a growing number of embedded systems includes a mixture of hardware and software. Most of the embedded system software is specified in C. For example, embedded systems implement standards such as MPEG 1/2/4 or decoders for mobile phone standards such as GSM. The standards are frequently available in the form of "reference implementations", consisting of C programs not optimized for speed but providing the required functionality. The disadvantage of design methodologies based on VHDL or Verilog is the fact that these standards have to be rewritten in order to generate hardware. Furthermore, simulating hardware and software together requires to interfacing software and hardware simulators. Typically, this involves a loss of simulation efficiency and inconsistent user interfaces. Also, designers have to learn several languages.

Therefore, there has been a search for techniques for representing hardware structures in software languages. Some fundamental problems have to be solved before hardware can be modeled with software languages:

- **Concurrency**, as it is found in hardware, has to be modeled in software.

- There has to be a representation for simulation **time**.

- **Multiple-valued logic** and **resolution** as described earlier must be supported.

- The **deterministic behavior** of almost all useful hardware circuits must be guaranteed.

SystemC^{TM} [SystemC, 2002] is a C++ class library designed to solve these problems. With SystemC, specifications can be written in C or C++, making appropriate references to the class libraries.

SystemC comprises a notion of processes executed concurrently. Simulation semantics are similar to VHDL, including the presence of delta cycles. The execution of these processes is controlled via sensivity lists and calls to wait primitives. The sensivity list concept of VHDL has been extended to also include dynamic sensivity lists (in SystemC 2.0).

SystemC includes a model of time. SystemC 1.0 uses floating point numbers to denote time. In SystemC 2.0, an integer model of time is preferred. SystemC 2.0 also supports physical units such as picoseconds, nanoseconds, microseconds etc.

SystemC data types include all common hardware types: four-valued logic ('0', '1', 'X' and 'Z') and bitvectors of different lengths are supported. Writing digital signal processing applications is simplified due to the availability of fixed-point data types.

Deterministic behavior (see page 27) is not guaranteed in general, unless a certain modeling style is used. Using a command line option, the simulator can be directed to run processes in different orders. This way, the user can check if the simulation results depend on the sequence in which the processes are executed. However, for models of realistic complexity, only the presence of non-deterministic behavior can be proved, not its absence.

Reusing hardware components in different contexts is simplified by the separation of computation and communication. SystemC 2.0 provides channels, ports and interfaces as abstract components for communication.

SystemC has the potential for replacing existing VHDL-based design flows, even though hardware synthesis from SystemC only starts becoming available at the time of the writing of this book [Herrera et al., 2003a], [Herrera et al., 2003b]. Methodology and applications for SystemC-based design is described in a book on that topic [Müller et al., 2003].

2.13 Verilog and SystemVerilog

Verilog [Thomas and Moorby, 1991] is another hardware description language. Initially it was a proprietary language, but it was later standardized as IEEE standard 1364, with versions called IEEE standard 1364-1995 (Verilog version 1.0) and IEEE standard 1394-2001 (Verilog 2.0). Some features of Verilog are quite similar to VHDL. Just like in VHDL, designs are described as a set of connected design entities, and design entities can be described behaviorally. Also, processes are used to model concurrency of hardware components. Just like in VHDL, bitvectors and time units are supported. There are, however, some areas in which Verilog is less flexible and focuses more on comfortable built-in features. For example, standard Verilog does not include the flexible mechanisms for defining enumerated types like the ones defined in the IEEE 1164 standard. However, Verilog support for four-valued logic is built into the language, and the standard IEEE 1364 also provides multiple valued logic with 8 different signal strengths. Multiple-valued logic is more tightly integrated into Verilog than into VHDL. The Verilog logic system also provides more features for transistor-level descriptions. However, VHDL is more flexible. For example, VHDL allows hardware entities to be instantiated in loops. This can be used to generate a structural description for, e.g. n-bit adders without having to specify n adders and their interconnections manually.

Verilog has a similar number of users as VHDL. While VHDL is more popular in Europe, Verilog is more popular in the US.

Verilog versions 3.0 and 3.1 are also known as SystemVerilog. They include numerous extensions to Verilog 2.0. These extensions include [Accellera, 2005]:

- additional language elements for modeling behavior,
- C data types such as int and type definition facilities such as typedef and struct,
- definition of interfaces of hardware components as separate entities,
- standardized mechanism for calling C/C++ functions and, to some extend, to call built-in Verilog functions from C,
- significantly enhanced features for describing an environment (called testbench) for the hardware under design (called CUD), and for using the testbench to verify the CUD by simulation,
- classes known from object-oriented programming for use within testbenches,

- dynamic process creation,
- standardized interprocess communication and synchronization, including semaphores,
- automatic memory allocation and deallocation,
- language features that provide a standardized interface to formal verification (see page 199).

Due to the capability of interfacing with C and C++, interfacing to SystemC models is also possible. Improved facilities for simulation- as well as for formal verification-based design validation and the possible interfacing to SystemC will potentially create a very good acceptance of SystemVerilog.

2.14 SpecC

The SpecC language [Gajski et al., 2000] is based on the clear separation between communication and computation that should be used for modeling embedded systems. This separation paves the way for re-using components in different contexts and enables *plug-and-play* for system components. SpecC models systems as hierarchical networks of behaviors communicating through channels. SpecC descriptions consist of behaviors, channels and interfaces. Behaviors include ports, locally instantiated components, private variables and functions and a public main function. Channels encapsulate communication. They include variables and functions, which are used for the definition of a communication protocol. Interfaces are linking behaviors and channels together. They declare the communication protocols which are defined in a channel.

SpecC can model hierarchies with nested behaviors. Fig. 2.61 [Gajski et al., 2000] shows a component B including sub-components b1 and b2.

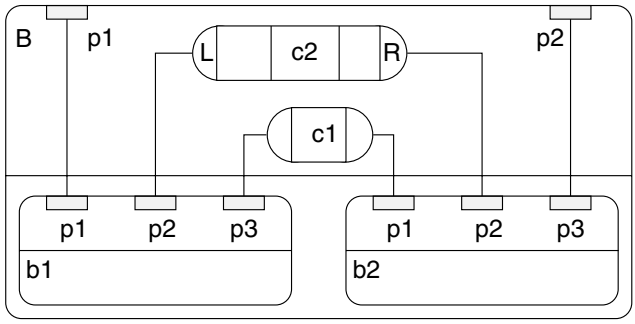

Figure 2.61. Structural hierarchy of SpecC example

The sub-components are communicating through integer c1 and through channel c2. The structural hierarchy includes b1 and b2 as the leaves. b1 and b2 are executed concurrently, denoted by the keyword **par** in SpecC. This structural hierarchy is described in the following SpecC model.

```
interface L {void Write(int x); };
interface R {int Read(void); };
channel C implements L,R
   {int Data; bool Valid;
   void Write(int x) {Data=x; Valid=true;}
   int Read (void)
     {while (!Valid) waitfor (10); return (Data);} }
  behavior B1(in int p1, L p2, in int p3)
   {void main (void) {/* ...*/ p2.Write(p1);} };
  behavior B2 (out int p1, R p2, out int p3)
   {void main(void) {/*...*/ p3=p2.Read(); } };
behavior B(in int p1, out int p2)
  {int c1;  C c2;  B1 b1(p1, c2, c1);  B2 b2(c1, c2, p2);
  void main (void)
   {par {b1.main(); b2.main();}}
  };
```

Note that the interface protocol implemented in channel C, consisting of methods for read and write operations, can be changed without changing behaviors B1 and B2. For example, communication can be bit-serial or parallel and the choice does not affect the models of B1 and B2. This is a necessary feature for IP-reuse.

In order to simplify designs containing software and hardware components, the syntax of SpecC is based on C and C++. In fact, SpecC models are translated into C++ for simulation.

The communication model of SpecC has inspired communication in SystemC 2.0.

2.15 Additional languages

A large number of languages has been designed for embedded applications. The following is a list of some of them:

- **Pearl:** Pearl [Deutsches Institut für Normung, 1997] was designed for industrial control applications. It does include a large repertoire of language elements for controlling processes and referring to time. It requires an underlying real-time operation system. Pearl has been very popular in Europe and a large number of industrial control projects has been implemented in Pearl.

- **Chill:** Chill [Winkler, 2002] was designed for telephone exchange stations. It was standardized by the CCITT and used in telecommunication equipment. Chill is a kind of extended PASCAL.

- **IEC 60848**, **STEP 7** : IEC 60848 [IEC, 2002] and STEP 7 [Berger, 2001] are two languages that are used in control applications. Both provide graphical elements for describing the system functionality.

- **SpecCharts:** SpecCharts [Gajski et al., 1994], as a predecessor to SpecC, combines the advantages of StateCharts and VHDL. It is based on the StateChart modeling paradigm of automata, but allows their behavior to be described in VHDL. In addition, a distinction between transitions to be taken immediately and transitions to be taken after the completion of all computations is made. The first type of transitions simplifies the modeling of exceptions. Just like StateCharts, SpecCharts has problems describing structural hierarchies.

- **Estelle:** This language was designed to describe communication protocols. Similar to SDL, Estelle assumes communication via channels and FIFO-buffers. Attempts to unify Estelle and SDL failed.

- **LOTOS, Z:** These languages [Jeffrey and Leduc, 1996], [Spivey, 1992] are algebraic specification languages enabling precise specifications and formal proofs. Unfortunately, they are not executable and hence cannot be used for early design validations.

- **Silage:** This language is tailored towards digital signal processing. It is a functional language, paving the way for getting rid of sequential, von-Neumann style of programming. Unfortunately, it has not been accepted by designers.

- **Rosetta:** The development of the Rosetta language is part of the activities of the Accellera initiative. *Accellera's mission is to drive worldwide development and use of standards required by systems, semiconductor and design tools companies, which enhance a language-based design automation process* [Accellera, 2002]. The Accellera initiative also works on updating VHDL and Verilog standards.

- **Esterel:** The definition of Esterel comprises the following salient features [Boussinot and de Simone, 1991]: Esterel is a reactive language: when activated with an input event, Esterel models react by producing an output event. Esterel is a synchronous language: all reactions are assumed to be completed in zero time and it is sufficient to analyze the behavior at discrete moments in time. This idealized model avoids all discussions about overlapping time ranges and about events that arrive while the previous reaction has not been completed. As other concurrent languages, Esterel has a parallelism operator, written $||$. Like in StateCharts, communication is based on a broadcast mechanism. In contrast to StateCharts, however, communication is instantaneous. This means that all signals generated at a particular moment in time are also seen by the others parts of the model at the same moment in time and these other parts, if sensitive to the generated signals, react at the same moment in time. Several rounds of evaluations may be required until a stable state (if any) is reached. This propagation of values during the same macroscopic instant of time corresponds to the δ-cycles of VHDL and the generation of a next status for the same moment in time in StateCharts, except that the broadcast is now instantaneous. Instantaneous broadcast can lead to instantaneous loops. These loops are detected by the Esterel compiler, but not all infinite loops can be detected by the compiler. For more and updated information about Esterel, refer to a web site [Esterel, 2002].
- **MATLAB/Simulink:** MATLAB/Simulink [Tewari, 2001] is a modeling and simulation tool based on mathematics. Actual systems can be described, for example, in the form of partial differential equations. This approach is appropriate for modeling physical systems such as cars or trains and then simulating the behavior of these systems. Also, digital signal processing systems can be conveniently modeled with MATLAB. In order to generate implementations, MATLAB/Simulink models first have to be translated into a language supported by software or hardware design systems, such as C or VHDL.

2.16 Levels of hardware modeling

In practice, designers start design cycles at various levels of abstraction. In some cases, these are high levels describing the overall behavior of the system to be designed. In other cases, the design process starts with the specification of electrical circuits at lower levels of abstraction. For each of the levels, a variety of languages exists, and some languages cover at various levels. In the following, we will describe a set of possible levels. Some lower end levels are presented here for context reasons. Specifications should not start at those levels. The following is a list of frequently used names and attributes of levels:

- **System level models:** The term system level is not clearly defined. It is used here to denote the entire embedded system and the system into which information processing is embedded ("the product"), and possibly also the environment (the physical input to the system, reflecting e.g. the roads, weather conditions etc.). Obviously, such models include mechanical as well as information processing aspects and it may be difficult to find appropriate simulators. Possible solutions include VHDL-AMS (the analog extension to VHDL), SystemC or MATLAB. MATLAB and VHDL-AMS support modeling partial differential equations, which is a key requirement for modeling mechanical systems. It is a challenge to model information processing parts of the system in such a way that the simulation model can also be used for the synthesis of the embedded system. If this is not possible, error-prone manual translations between different models may be needed.

- **Algorithmic level:** At this level, we are simulating the algorithms that we intend to use within the embedded system. For example, we might be simulating MPEG video encoding algorithms in order to evaluate the resulting video quality. For such simulations, no reference is made to processors or instruction sets.

  Data types may still allow a higher precision than the final implementation. For example, MPEG standards use double precision floating point numbers. The final embedded system will hardly include such data types. If data types have been selected such that every bit corresponds to exactly one bit in the final implementation, the model is said to be **bit-true**. Translating non-bit-true into bit-true models should be done with tool support (see page 158).

  Models at this level may consist of single processes or of sets of cooperating processes.

- **Instruction set level:** In this case, algorithms have already been compiled for the instruction set of the processor(s) to be used. Simulations at this level allow counting the executed number of instructions. There are several variations of the instruction set level:

  - In a coarse-grained model, only the effect of the instructions is simulated and their timing is not considered. The information available in assembly reference manuals (instruction set architecture (ISA)) is sufficient for defining such models.
  - **Transaction level modeling:** In transaction level modeling, transactions, such as bus reads and writes, and communication between different components is modeled. Transaction level modeling includes less

details than cycle-true modeling (see below), enabling significantly superior simulation speeds [Clouard et al., 2003].

- In a more fine-grained model, we might have **cycle-true instruction set simulation**. In this case, the exact number of clock cycles required to run an application can be computed. Defining cycle-true models requires a detailed knowledge about processor hardware in order to correctly model, for example, pipeline stalls, resource hazards and memory wait cycles.

- **Register-transfer level (RTL):** At this level, we model all the components at the register-transfer level, including arithmetic/logic units (ALUs), registers, memories, muxes and decoders. Models at this level are always cycle-true. Automatic synthesis from such models is not a major challenge.

- **Gate-level models:** In this case, models contain gates as the basic components. Gate-level models provide accurate information about signal transition probabilities and can therefore also be used for power estimations. Also delay calculations can be more precise than for the RTL. However, typically no information about the length of wires and hence no information about capacitances is available. Hence, delay and power consumption calculations are still estimates.

  The term "gate-level model" is sometimes also employed in situations in which gates are only used to denote Boolean functions. Gates in such a model do not necessarily represent physical gates; we are only considering the behavior of the gates, not the fact that they also represent physical components. More precisely, such models should be called "Boolean function models"[10], but this term is not frequently used.

- **Switch-level models:** Switch level models use switches (transistors) as their basic components. Switch level models use digital values models (refer to page 62 for a description of possible value sets). In contrast to gate-level models, switch level models are capable of reflecting bidirectional transfer of information.

- **Circuit-level models:** Circuit theory and its components (current and voltage sources, resistors, capacitances, inductances and possibly macro-models of semiconductors) form the basis of simulations at this level. Simulations involve partial differential equations. These equations are linear if and only if the behavior of semiconductors is linearized (approximated). The most frequently used simulator at this level is SPICE [Vladimirescu, 1987] and its variants.

[10] These models could be represented with binary decision diagrams (BDDs) [Wegener, 2000].

- **Layout models:** Layout models reflect the actual circuit layout. Such models include **geometric** information. Layout models cannot be simulated directly, since the geometric information does not directly provide information about the behavior. Behavior can be deduced by correlating the layout model with a behavioral description at a higher level or by extracting circuits from the layout, using knowledge about the representation of circuit components at the layout level. In a typical design flow, the length of wires and the corresponding capacitances are extracted from the layout and **back-annotated** to descriptions at higher levels. This way, more precision can be gained for delay and power estimations.

- **Process and device models:** At even lower levels, we can model fabrication processes. Using information from such models, we can compute parameters (gains, capacitances etc) for devices (transistors).

2.17 Language comparison

None of the languages presented so far meets all the requirements for specification languages for embedded systems. Fig. 2.62 presents an overview over some of the key properties of some of the languages. Exceptions and dynamic process creation are supposed to be supported in SystemC 3.0.

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+-	+-	+-	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	- (2.0)	- (2.0)
ADA	+	-	+	+	+

Figure 2.62. Language comparison

It is not very likely that a single language will ever meet all requirements, since some of the requirements are essentially conflicting. A language supporting hard real-time requirements well may be inconvenient to use for less strict real-time requirements. A language appropriate for distributed control-dominated applications may be poor for local data-flow dominated applications. Hence, we can expect that we will have to live with compromises.

Which compromises are actually used in practice? In practice, assembly language programming was very common in the early years of embedded systems programming. Programs were small enough to handle the complexity of problems in assembly languages. The next step is the use of C or derivatives of C. Due to the ever increasing complexity of embedded system software (see page 9), higher level languages are to follow the introduction of C. Object oriented languages and SDL are languages which provide the next level of abstraction. Also, languages like UML are required to capture specifications at an early design stage. In practice, these languages can be used like shown in fig. 2.63.

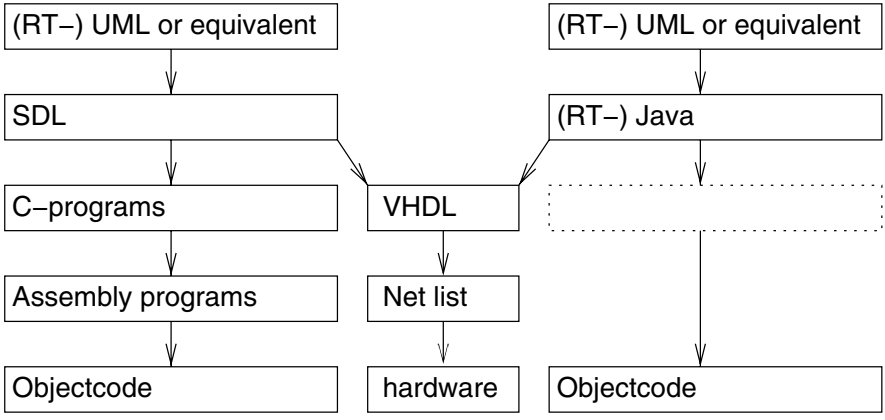

Figure 2.63. Using various languages in combination

According to fig. 2.63, languages like SDL or StateCharts can be translated into C. These C descriptions are then compiled. Starting with SDL or StateChart also opens the way to implementing the functionality in hardware, if translators from these languages to VHDL are provided. Both C and VHDL will certainly survive as intermediate languages for many years. Java does not need intermediate steps but does also benefit from good translation concepts to assembly languages.

2.18 Dependability requirements

The previous sections have mostly focused on the specification of the functional behavior of the system to be designed. However, there may be safety-critical systems, for which safety requirements are actually the dominating requirement. Safety requirements cannot come in as an afterthought, but have to be considered right from the beginning. The design of safe and dependable systems is a topic by its own. This book can only provide a few hints into this direction.

According to Kopetz [Kopetz, 2003], the following must be taken into account: For safety-critical systems, the system as a whole must be more dependable than any of its parts. Allowed failure may be in the order of 1 failure per 10^9 hours. This may be in the order of 1000 times less than typical failure rates of chips. Obviously, fault-tolerance mechanisms must be used. Due to the low acceptable failure rate, systems are not 100% testable. Instead, safety must be shown by a combination of testing and reasoning. Abstraction must be used to make the system explainable using a hierarchical set of behavioral models. Design faults and human failures must be taken into account. In order to address these challenges, Kopetz proposed the following twelve design principles:

1 Safety considerations may have to be used as **the** important part of the specification, driving the entire design process.

2 Precise specifications of design hypotheses must be made right at the beginning. These include expected failures and their probability.

3 Fault containment regions (FCRs) must be considered. Faults in one FCR should not affect other FCRs.

4 A consistent notion of time and state must be established. Otherwise, it will be impossible to differentiate between original and follow-up errors.

5 Well-defined interfaces have to hide the internals of components.

6 It must be ensured that components fail independently.

7 Components should consider themselves to be correct unless two or more other components pretend the contrary to be true (principle of self-confidence).

8 Fault tolerance mechanisms must be designed such that they do not create any additional difficulty in explaining the behavior of the system. Fault tolerance mechanisms should be decoupled from the regular function.

9 The system must be designed for diagnosis. For example, it has to be possible to identifying existing (but masked) errors.

10 The man-machine interface must be intuitive and forgiving. Safety should be maintained despite mistakes made by humans.

11 Every anomaly should be recorded. These anomalies may be unobservable at the regular interface level. This recording should involve internal effects, since otherwise they may be masked by fault-tolerance mechanisms.

12 Provide a never-give up strategy. Embedded systems may have to provide uninterrupted service. The generation of pop-up windows or going offline is unacceptable.

For further information about dependability and safety issues, contact books [Laprie, 1992], [Neumann, 1995], [Leveson, 1995], [Storey, 1996], [Geffroy and Motet, 2002] on those areas.

Chapter 3

EMBEDDED SYSTEM HARDWARE

3.1 Introduction

It is one of the characteristics of embedded systems that both hardware and software must be taken into account. The reuse of available hard- and software components is at the heart of the proposed **platform-based design methodology**. The methodology will be described starting at page 151. Consistent with the need to consider available hardware components and with the design information flow shown in fig. 3.1, we are now going to describe some of the essentials of embedded system hardware.

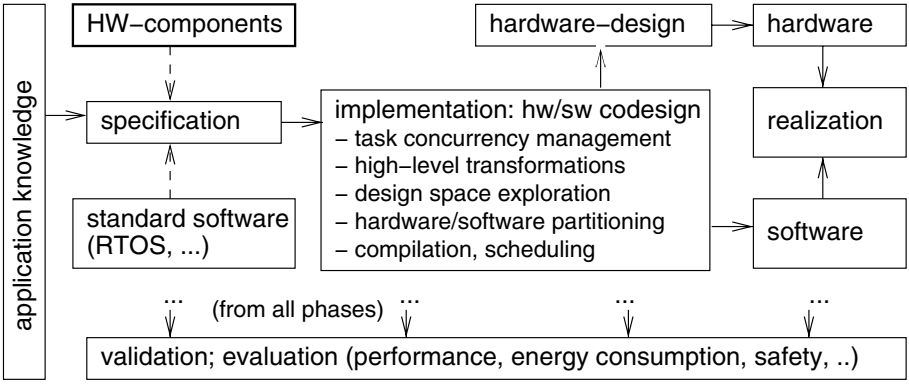

Figure 3.1. Simplified design information flow

Hardware for embedded systems is much less standardized than hardware for personal computers. Due to the huge variety of embedded system hardware, it is impossible to provide a comprehensive overview over all types of hardware

components. Nevertheless, we will try to provide an overview over some of the essential components which can be found in most systems.

In many of the embedded systems, especially in control systems, embedded hardware is used in a loop (see fig. 3.2).

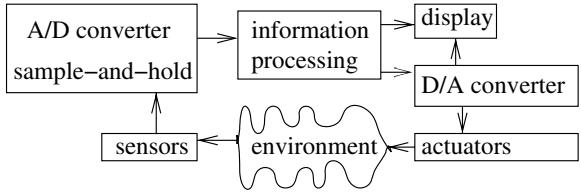

Figure 3.2. Hardware in the loop

In this loop, information about the physical environment is made available through **sensors**. Typically, sensors generate continuous sequences of analog values. In this book, we will restrict ourselves to information processing in digital computers processing discrete sequences of values. Appropriate conversions are performed by two kinds of circuits: sample-and-hold-circuits and analog-to-digital (A/D) converters. After this conversion, information can be processed digitally. Generated results can be displayed and also be used to control the physical environment through actuators. Since most actuators are analog actuators, conversion from digital to analog signals is also needed.

This model is obviously appropriate for control applications. For other applications, it can be employed as a first order approximation. For example, in mobile phones, sensors correspond to the antenna and actuators correspond to the speakers. In the following, we will describe essential hardware components of embedded systems following the loop structure of fig. 3.2.

3.2 Input

3.2.1 Sensors

We start with a brief discussion of sensors. Sensors can be designed for virtually every physical quantity. There are sensors for weight, velocity, acceleration, electrical current, voltage, temperatures etc. A large amount of physical effects can be used for constructing sensors [Elsevier B.V., 2003a]. Examples include the law of induction (generation of voltages in an electric field), or light-electric effects. Also, there are sensors for chemical substances [Elsevier B.V., 2003b].

In recent years, a huge amount of sensors has been designed and much of the progress in designing smart systems can be attributed to modern sensor

technology. Hence, it is impossible to cover this subset of embedded hardware technology comprehensively and we can only give characteristic examples:

- **Acceleration sensors:** Fig. 3.3 shows a small sensor manufactured using microsystem technology. The sensor contains a small mass in its center. When accelerated, the mass will be displaced from its standard position, thereby changing the resistance of the tiny wires connected to the mass.

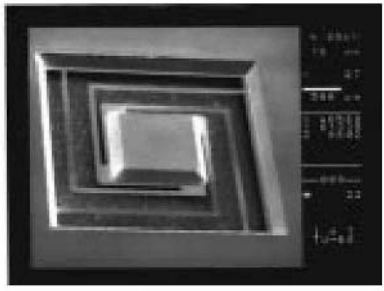

Figure 3.3. Acceleration sensor (courtesy S. Bütgenbach, IMT, TU Braunschweig), ©TU Braunschweig, Germany

- **Rain sensors:** In order to remove distraction from drivers, some recent high end cars contain rain sensors. Using these, the speed of the wipers can be automatically adjusted to the amount of rain.

- **Image sensors:** There are essentially two kinds of image sensors: charge-coupled devices (CCDs) and CMOS sensors. In both cases, arrays of light sensors are used. The architecture of CMOS sensor arrays is similar to that of standard memories: individual pixels can be randomly addressed and read out. CMOS sensors use standard CMOS technology for integrated circuits [Dierickx, 2000]. Due to this, sensors and logic circuits can be integrated on the same chip. This allows some preprocessing to be done already on the sensor chip, leading to so-called smart sensors. CMOS sensors require only a single standard supply voltage and interfacing in general is easy. Therefore, CMOS-based sensors can be cheap. In contrast, CCD technology is optimized for optical applications. In CCD technology, charges have to be transfered from one pixel to the next until they can finally be read out at an array boundary. This sequential charge transfer also gave CCDs their name. Images generated with CCDs can be of higher quality than those generated using CMOS sensors, since they generate less noise. However, interfacing is more complex. As a result, CMOS sensors are appropriate for applications requiring low or medium costs and low or medium image quality. CCD sensors are more adequate for high quality,

expensive image sensors (such as those found in video cameras and optical telescopes, for example).

- **Bio-metrical sensors:** Demands for higher security standards as well as the need to protect mobile and removable equipment have led to an increased interest in authentication. Due to the limitations of password based security (e.g. stolen and lost passwords), smartcards, bio-metrical sensors and bio-medical authentication receive significant attention. Bio-medical authentication tries to identify whether or not a certain person is actually the person she or he claims to be. Methods for bio-medical authentication include iris scans, finger print sensors and face recognition. Finger print sensors are typically fabricated using the same CMOS technology [Weste et al., 2000] which is used for manufacturing integrated circuits. Possible applications include notebooks which grant access only if the user's finger print is recognized [IBM Inc., 2002]. CCD and CMOS image sensors described above are used for face recognition. False accepts as well as false rejects are an inherent problem of bio-medical authentication. In contrast to password based authentication, exact matches are not possible.

- **Artifical eyes**: Artificial eye projects have received significant attention. While some projects attempt to actually affect the eye, others try to provide vision in an indirect way. The Dobelle Institute is experimenting with a setup in which a little camera is attached to glasses. This camera is connected to a computer translating these patterns into electrical pulses. These pulses are then sent directly to the brain, using a direct contact through an electrode. Currently (2003), the resolution is in the order of 128 by 128 pixels, enabling blind persons to drive a car in controlled areas [The Dobelle Institute, 2003].

- **Other sensors:** Other common sensors include: pressure sensors, proximity sensors, engine control sensors, Hall effect sensors, and many more.

3.2.2 Sample-and-hold circuits

All known digital computers work in the discrete time domain. This means they can process discrete sequences of values. Hence, values in the continuous domain have to be converted to the discrete domain. This is the purpose of sample-and-hold circuits. Fig. 3.4 (left) shows a simple sample-and-hold-circuit.

In essence, the circuit consists of a clocked transistor and a capacitor. The transistor operates like a switch. Each time the switch is closed by the clock signal, the capacitor is charged so that its voltage is practically the same as the incoming voltage V_e. After opening the switch again, this voltage will

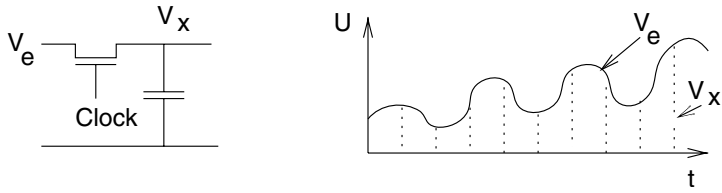

Figure 3.4. Sample-and-hold-circuit

remain essentially unchanged until the switch is closed again. Each of the values stored on the capacitor can be considered as an element of a discrete sequence of values V_x, generated from a continuous sequence V_e (see fig. 3.4, right).

An ideal sample-and-hold circuit would be able to change the voltage at the capacitor in an arbitrarily short amount of time. This way, the input voltage at a particular instance in time could be transfered to the capacitor and each element in the discrete sequence would correspond to the input voltage at a particular point in time. In practice, however, the transistor has to be kept closed for a short time window in order to really charge or discharge the capacitor. The voltage stored on the capacitor will then correspond to a voltage averaged over that short time window.

3.2.3 A/D-converters

Since we are restricting ourselves to digital computers, we will also have to work with discrete values representing our input signals. The conversion from analog to digital values is done by analog-to-digital (A/D) converters. There is a large range of A/D converters with varying speed/precision characteristics. In this book, we will present two extreme cases:

- **Flash A/D converter**: This type of A/D converters uses a large number of comparators. Each comparator has two inputs, denoted as + and -. If the voltage at input + exceeds that at input -, the output corresponds to a logical '1' and it corresponds to a logical '0' otherwise[1].

  In the A/D-converter, all - inputs are connected to a voltage divider. Now, if input voltage V_x exceeds constant voltage V_{ref}, the comparator at the top of fig. 3.5 will generate a '1'. The encoder at the output of the comparators will

[1] In practice, the case of equal voltages is not relevant, as the actual behavior for very small differences between the voltages at the two inputs depends on many factors (like temperatures, manufacturing processes etc.) anyway.

try to identify the most significant '1' and will encode the case of $V_x > V_{ref}$ as the largest output value.

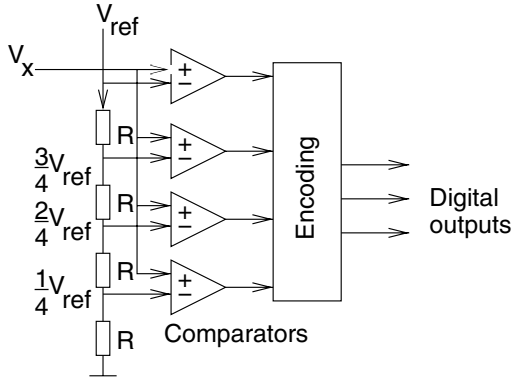

Figure 3.5. Flash A/D converter

Now if input voltage V_x is less than V_{ref}, but still larger than $\frac{3}{4}V_{ref}$, the comparator at the top of fig. 3.5 will generate a '0', while the next comparator will still signal a '1'. The encoder will encode this as the second-largest value.

Similar arguments hold for cases $\frac{2}{4}V_{ref} < V_x < \frac{3}{4}V_{ref}$, $\frac{1}{4}V_{ref} < V_x < \frac{2}{4}V_{ref}$, and $0 < V_x < \frac{1}{4}V_{ref}$, which will be encoded as the third-largest, fourth-largest and smallest value, respectively.

The circuit can convert positive analog input voltages into digital values. Converting both positive and negative voltages requires some extensions.

The key advantage of the circuit is its speed. It does not need any clock. The delay between the input and the output is very small and the circuit can be used easily, for example, for high-speed video applications. The disadvantage is its hardware complexity: we need $n-1$ comparators in order to distinguish between n values. Imagine using this circuit in generating digital audio signals for CD recorders. We would need $2^{16}-1$ comparators!

- **Successive approximation:** Distinguishing between a large number of digital values is possible with A/D converters using successive approximation. The circuit is shown in fig. 3.6.

  The key idea of this circuit is to use binary search. Initially, the most significant output bit of the successive approximation register is set to '1', all other bits are set to '0'. This digital value is then converted to an analog

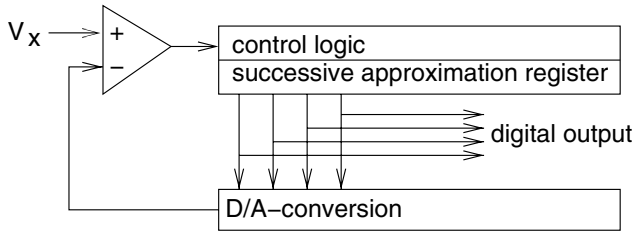

Figure 3.6. Circuit using successive approximation

value, corresponding to $0.5\times$ the maximum input voltage[2]. If V_x exceeds the generated analog value, the most significant bit is kept at '1', otherwise it is reset to '0'.

This process is repeated with the next bit. It will remain set to '1' if the input value is either within the second or the fourth quarter of the input value range. The same procedure is repeated for all the other bits.

The key advantage of the successive approximation technique is its hardware efficiency. In order to distinguish between n digital values, we need $log_2(n)$ bits in the successive approximation register and the D/A converter. The disadvantage is its speed, since it needs $O(log_2(n))$ steps. These converters can therefore be used for applications, where high precision conversions at moderate speeds are required. Examples include audio applications.

There are several other types of A/D-converters. Techniques for automatically selecting the most appropriate converter exist [Vogels and Gielen, 2003].

3.3 Communication

Information must be available before it can be processed in an embedded system. Information can be communicated through various **channels**. Channels are abstract entities characterized by the essential properties of communication, like maximum information transfer capacity and noise parameters. The probability of communication errors can be computed using communication theory techniques. The physical entities enabling communication are called communication **media**. Important media classes include: wireless media (radio frequency media, infrared), optical media (fibers), and wires.

[2]Fortunately, the conversion from digital to analog values (D/A-conversion) can be implemented very efficiently and can be very fast (see page 121).

There is a huge variety of communication requirements between the various classes of embedded systems. In general, connecting the different embedded hardware components is far from trivial. Some common requirements can be identified.

3.3.1 Requirements

The following list contains some of the requirements that have to be met:

- **Real-time behavior**: This requirement has far-reaching consequences on the design of the communication system. Some of the low-cost solutions such as Ethernet fail to meet this requirement.
- **Efficiency**: Connecting different hardware components can be quite expensive. For example, point to point connections in large buildings are almost impossible. Also, it has been found that separate wires between control units and external devices in cars significantly add to the cost and the weight of the car. With separate wires, it is also very difficult to add components. The need of providing cost efficient designs also affects the way in which power is made available to external devices. There is frequently the need to use a central power supply in order to reduce the cost.
- **Appropriate bandwidth and communication delay**: Bandwidth requirements of embedded systems may vary. It is important to provide sufficient bandwidth without making the communication system too expensive.
- **Support for event-driven communication**: Polling-based systems provide a very predictable real-time behavior. However, their communication delay may be too large and there should be mechanisms for fast, event-oriented communication. For example, emergency situations should be communicated immediately and should not remain unnoticed until some central controller polls for messages.
- **Robustness**: Embedded systems may be used at extreme temperatures, close to major sources of electromagnetic radiation etc. Car engines, for example, can be exposed to temperatures less than -20 and up to +180 degrees Celsius (-4 to 356 degrees Fahrenheit). Voltage levels and clock frequencies could be affected due to this large variation in temperatures. Still, reliable communication must be maintained.
- **Fault tolerance**: Despite all the efforts for robustness, faults may occur. Embedded systems should be operational even after such faults. Restarts, like the ones found in personal computers, cannot be accepted. This means that retries may be required after attempts to communicate failed. A conflict

exists with the first requirement: If we allow retries, then it is difficult to meet strict real-time requirements.

- **Maintainability, diagnosability**: Obviously, it should be possible to repair embedded systems within reasonable time frames.
- **Privacy:** Ensuring privacy of confidential information may require the use of encryption.

These communication requirements are a direct consequence of the general characteristics of embedded systems mentioned in chapter 1. Due to the conflicts between some of the requirements, compromises have to be made. For example, there may be different communication modes: one high-bandwidth mode guaranteeing real-time behavior but no fault tolerance (this mode is appropriate for multimedia streams) and a second fault-tolerant, low-bandwidth mode for short messages that must not be dropped.

3.3.2 Electrical robustness

There are some basic techniques for electrical robustness. Digital communication within chips is normally using so-called single-ended signaling. For single-ended signaling, signals are propagated on a single wire (see fig. 3.8).

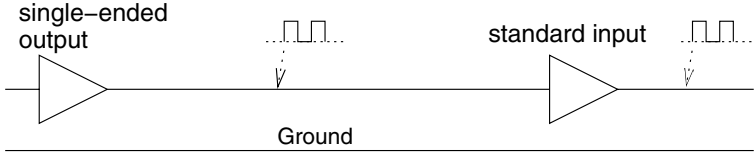

Figure 3.7. Single-ended signaling

Such signals are represented by voltages with respect to a common ground (less frequently by currents). A single ground wire is sufficient for a number of single-ended signals. Single ended signaling is very much susceptible to external noise. If external noise (originating from, for example, motors being switched on) affects the voltage, messages can easily be corrupted. Also, it is difficult to establish high-quality common ground signals between a large number of communicating systems, due to the resistance (and inductance) on the ground wires. This is different for differential signaling. For differential signaling, each signal needs two wires (see fig. 3.8).

Using differential signaling, binary values are encoded as follows: If the voltage on the first wire with respect to the second is positive, then this decoded as '1', otherwise values are decoded as '0'. The two wires will typically be twisted

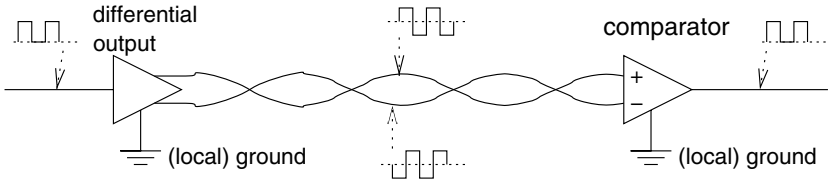

Figure 3.8. Differential signaling

to form so-called **twisted pairs**. There will be local ground signals, but a non-zero voltage between the local ground signals does not hurt. Advantages of differential signaling include:

- Noise is added to the two wires in essentially the same way. The comparator therefore removes almost all the noise.
- The logic value depends just on the polarity of the voltage between the two wires. The magnitude of the voltage can be affected by reflections or because of the resistance of the wires; this has no effect on the decoded value.
- Signals do not generate any currents on the ground wires. Hence, the quality of the ground wires becomes less important.
- No common ground wire is required. Hence, there is no need to establish a high quality ground wiring between a large number of communicating partners (this is one of the reasons for using differential signaling for Ethernet).
- As a consequence of the properties mentioned so far, differential signaling allows a larger throughput than single-ended signaling.

However, differential signaling requires two wires for every signal and it also requires negative voltages (unless it is based on complementary logic signals using voltages for single-ended signals).

Differential signaling is used, for example, in standard Ethernet-based networks.

3.3.3 Guaranteeing real-time behavior

Most computer networks are based on Ethernet standards. For 10 Mbit/s and 100 Mbit/s versions of Ethernet, there can be collisions between various communication partners. This means: several partners are trying to communicate at about the same time and the signals on the wires are corrupted. Whenever this occurs, the partners have to stop communications, wait for some time,

and then retry. The waiting time is chosen at random, so that it is not very likely that the next attempt to communicate results in another collision. This method is called **carrier-sense multiple access/collision detect** (CSMA/CD). For CSMA/CD, communication time can get huge, since conflicts can repeat a large number of times, even though this is not very likely. Hence, CSMA/CD cannot be used when real-time constraints have to be met.

This problem can be solved with CSMA/CA (**carrier-sense multiple access/ collision avoidance**). As the name indicates, collisions are completely avoided, rather than just detected. For CSMA/CA, priorities are assigned to all partners. Communication media are allocated to communication partners during **arbitration phases**, which follow communication phases. During arbitration phases, partners wanting to communicate indicate this on the media. Partners finding such indications of higher priority have to immediately remove their indication.

Provided that there is an upper bound on the time between arbitration phases, CSMA/CA guarantees a predictable real-time behavior for the partner having the highest priority. For other partners, real-time behavior can be guaranteed if the higher priority partners do not continuously request access to the media.

Note that high-speed versions of Ethernet (1 Gbit/s) also avoid collisions.

3.3.4 Examples

- **Sensor/actuator busses:** Sensor/actuator busses provide communication between simple devices such as switches or lamps and the processing equipment. There may be many such devices and the cost of the wiring needs special attention for such busses.

- **Field busses:** Field busses are similar to sensor/actuator busses. In general, they are supposed to support larger data rates than sensor/actuator busses. Examples of field busses include the following:

  - **Controller Area Network (CAN):** This bus was developed in 1981 by Bosch and Intel for connecting controllers and peripherals. It is popular in the automotive industry, since it allows the replacement of a large amount of wires by a single bus. Due to the size of the automotive market, CAN components are relatively cheap and are therefore also used in other areas such as smart homes and fabrication equipment. CAN has the following properties:

    * differential signaling with twisted pairs,
    * arbitration using CSMA/CA,
    * throughput between 10kbit/s and 1 Mbit/s,

  - low and high-priority signals,
  - maximum latency of 134 μs for high priority signals,
  - coding of signals similar to that of serial (RS-232) lines of PCs, with modifications for differential signaling.
- The **Time-Triggered-Protocol (TTP)** [Kopetz and Grunsteidl, 1994] for fault-tolerant safety systems like airbags in cars.
- **FlexRay** [FlexRay Consortium, 2002] is a TDMA (Time Division Multiple Access) protocol which has been developed by the FlexRay consortium (BMW, DaimlerChrysler, General Motors, Ford, Bosch, Motorola and Philips Semiconductors). FlexRay is a combination of a variant of the TTP and the byteflight [Byteflight Consortium, 2003] protocol.
- **MAP:** MAP is a bus designed for car factories.
- **EIB:** The European Installation Bus (EIB) is a bus designed for smart homes.

- **Wireless communication:** Wireless communication is becoming more popular, but communication bandwidth is becoming a scarce resource. As a result, the frequencies reserved for third generation UMTS mobile phones have been sold at extremely high prices (at about 500 Euros or dollars per person living in Germany).

  Bluetooth is a standard for connecting devices such as mobile phones and their headsets.

  The wireless version of Ethernet is standardized as IEEE standard 802.11. It is being used in local area networks (LANs).

  DECT is a standard used for wireless phones in Europe.

  HomeRF [palowireless, 2003] is a standard for synchronous wireless transmission of speech and multimedia data.

3.4 Processing Units

3.4.1 Overview

For information processing, we will consider ASICs (application-specific integrated circuits) using hardwired multiplexed designs, reconfigurable logic, and processors. These three technologies are quite different, for example, as far as their energy efficiency is concerned. Fig. 3.9 (approximating information provided by H. De Man and Th. Claasen [De Man, 2002]) shows the number of operations per Watt that can be achieved with a certain hardware technology.

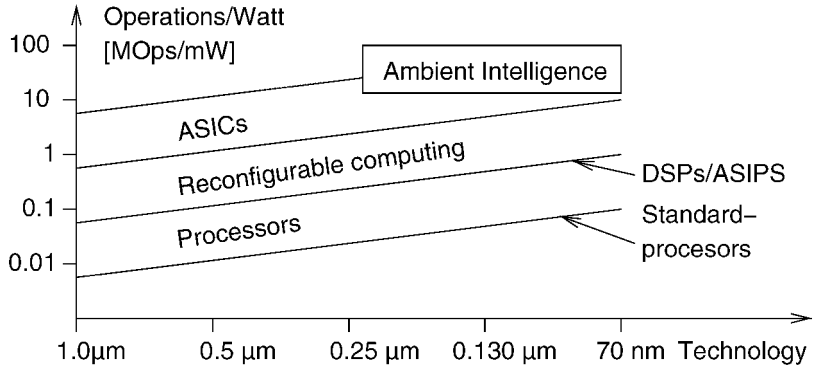

Figure 3.9. Hardware efficiency

Obviously, the number of operations per Watt is increasing as technology advances to smaller and smaller feature sizes of integrated circuits. However, for any given technology, the number of operations per Watt is largest for application specific hardwired circuits. For reconfigurable logic (see page 115), this value is about one order of magnitude lower. For programmable processors, it is about two orders of magnitude lower. On the other hand, processors offer the largest amount of flexibility, resulting from the flexibility of software. There is also some flexibility for reconfigurable logic, but it is limited to the size of applications that can be mapped to such logic. For hardwired designs, there is no flexibility. This observation also applies for processors: For processors optimized for the application domain, such as processors optimized for digital signal processing (DSP processors), power-efficiency values approach those of reconfigurable logic. For general standard microprocessors, the values for this figure of merit are the worst.

The energy E for a certain application is closely related to the power P required per operation, since

$$E = \int P dt$$

Hence, reducing the power consumption also decreases the energy consumption, provided that the integral is taken over the same period of time. In some cases, however, a slightly increased power consumption might lead to a drastic reduction in the execution time and, hence, might lead to a minimized energy consumption. So, in some cases a minimized power consumption also corresponds to a minimized energy consumption, but this is not necessarily always true.

Minimization of power and energy consumption are both important. Power consumption has an effect on the size of the power supply, the design of the voltage regulators, the dimensioning of the interconnect, and short term cooling. Minimizing the energy consumption is required especially for mobile applications, since battery technology is only slowly improving [SEMATECH, 2003], and since the cost of energy may be quite high. Also, a reduced energy consumption decreases cooling requirements and improves the reliability (since the lifetime of electronic circuits decreases for high temperatures).

Fig. 3.9 reflects the efficiency/flexibility conflict of currently available hardware technologies: if we want to aim at very power- and energy-efficient designs, we should not use flexible designs based on processors or re-programmable logic and if we go for excellent flexibility, we cannot be power-efficient. We will consider ASICs first.

3.4.2 Application-Specific Circuits (ASICs)

For high-performance applications and for large markets, application-specific integrated circuits (ASICs) can be designed. However, the cost of designing and manufacturing such chips is quite high. For example, the cost of the mask which is used for transferring geometrical patterns onto the chip can cost about 10^5 Euros or dollars. Therefore, ASICs are appropriate only if either maximum energy efficiency is needed and if the market accepts the costs or if a large number of such systems can be sold.

3.4.3 Processors

The key advantage of processors is their flexibility. With processors, the overall behavior of embedded systems can be changed by just changing the software running on those processors. Changes of the behavior may be required in order to correct design errors, to update the system to a new or changed standard or in order to add features to the previous system. Because of this, processors have become very popular. This popularity has also been stressed in the public press:

At the chip level, embedded chips include micro-controllers and microprocessors. Micro-controllers are the true workhorses of the embedded family. They are the original 'embedded chips' and include those first employed as controllers in elevators and thermostats [Ryan, 1995].

Embedded processors have to be efficient and they do not need to be instruction set compatible with commonly used personal computers (PCs). Therefore, their architectures may be different from those processors found in PCs. Efficiency has a number of different aspects (see page 2) :

- **Energy-efficiency**: Architectures have to be optimized for their energy-efficiency and we have to make sure that we are not loosing efficiency in the software generation process. For example, compilers generating 50% overhead in terms of the number of cycles will take us further away from the efficiency of ASICs, possibly by even more than 50%, if the supply voltage and the clock frequency have to be increased in order to meet deadlines.

  There is a large amount of techniques available that can make processors energy efficient and energy efficiency should be considered at various levels of abstraction, from the design of the instruction set down to the design of the chip manufacturing process [Burd and Brodersen, 2003]. Gated clocking is an example of such a technique. With gated clocking, parts of the processor are decoupled from the clock during idle periods. For example, no clock is applied to the multiplier if no multiplications are executed. Also, there are attempts, to get rid of the clock for major parts of the processor altogether. There are two contrasting approaches: globally synchronous, locally asynchronous processors and globally asynchronous, locally synchronous processors (GALS) [Iyer and Marculescu, 2002].

  Two techniques can be applied at a rather high level of abstraction:

  – **Dynamic power management (DPM):** With this approach, processors have several power saving states in addition to the standard operating state. Each power saving state has a different power consumption and a different time for transitions into the operating state. Fig. 3.10 shows the three states for the StrongArm SA 1100 processor.

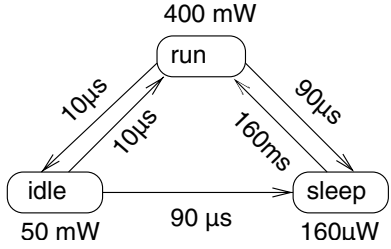

Figure 3.10. Dynamic power management states of the StrongArm Processor SA 1100

    The processor is fully operational in the run state. In the idle state, it is just monitoring the interrupt inputs. In the sleep state, all on-chip activity is shutdown. Note the large difference in the power consumption between the sleep state and the other states, and note also the large delay for transitions from the sleep to the run state.

  – **Dynamic voltage scaling (DVS):** This approach exploits the fact that the energy consumption of CMOS processors increases quadratically

with the supply voltage V_{dd}. The power consumption P of CMOS circuits is given by [Chandrakasan et al., 1992]:

$$P = \alpha\ C_L\ V_{dd}^2\ f \tag{3.1}$$

where α is the switching activity, C_L is the load capacitance, V_{dd} is the supply voltage and f is the clock frequency. The delay of CMOS circuits can be approximated as [Chandrakasan et al., 1992], [Chandrakasan et al., 1995]:

$$\tau = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{3.2}$$

where k is a constant, and V_t is the threshold voltage. V_t has an impact on the transistor input voltage required to switch the transistor on. For example, for a maximum supply voltage of $V_{dd,max}$=3.3 volts, V_t may be in the order of 0.8 volts. Consequently, the maximum clock frequency is a function of the supply voltage. However, decreasing the supply voltage reduces the power quadratically, while the run-time of algorithms is only linearly increased (ignoring the effects of the memory system). This can be exploited in a technique called **dynamic voltage scaling (DVS)**. For example, the Crusoe^{TM} processor by Transmeta provides 32 voltage levels between 1.1 and 1.6 volts, and the clock can be varied between 200 MHz and 700 MHz in increments of 33 MHz. Transitions from one voltage/frequency pair to the next takes about 20 ms. Design issues for DVS-capable processors are described in a paper by Burd and Brodersen [Burd and Brodersen, 2000]. According to the same paper, potential power savings will exist even for future technologies with a decreased maximum V_{dd}, since the threshold voltages will also be decreased (unfortunately, this will lead to increased leakage currents, increasing the standby power consumption). Two different speed/voltage pairs are provided with the Intel® SpeedStep^{TM} technology for the Mobile Pentium® III.

- **Code-size efficiency**: Minimizing the code size is very important for embedded systems, since hard disc drives are typically not available and since the capacity of memory is typically also very limited. This is even more pronounced for **systems on a chip** (SOCs). For SOCs, the memory and processors are implemented on the same chip. In this particular case, memory is called **embedded memory**. Embedded memory may be more expensive to fabricate than separate memory chips, since the fabrication processes for

memories and processors have to be compatible. Nevertheless, a large percentage of the total chip area may be consumed by the memory. There are several techniques for improving the code-size efficiency:

- **CISC machines**: Standard RISC processors have been designed for speed, not for code-size efficiency. Earlier Complex Instruction Set Processors (CISC machines) were actually designed for code-size efficiency, since they had to be connected to slow memories and caches were not frequently used. Therefore, "old-fashioned" CISC processors are finding applications in embedded systems. Motorola's ColdFire processors, which are based on the Motorola 68000 family of CISC processors are an example of this.
- **Compression techniques**: In order to reduce the amount of silicon needed for storing instructions as well as in order to reduce the energy needed for fetching these instructions, instructions are frequently stored in the memory in compressed form. This reduces both the area as well as the energy necessary for fetching instructions. Due to the reduced bandwidth requirements, fetching can also be faster. A (hopefully small and fast) decoder is placed between the processor and the (instruction) memory in order to generate the original instructions on the fly (see fig. 3.11, right). Instead of using a potentially large memory of uncompressed instructions, we are storing the instructions in a compressed format.

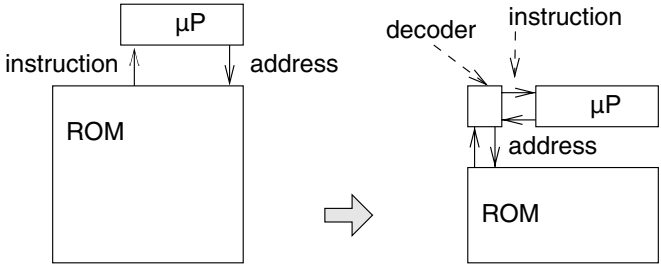

Figure 3.11. Decompression of compressed instructions

The goals of compression can be summarized as follows:

* We would like to save ROM and RAM areas, since these may be more expensive than the processors themselves.
* We would like to use some encoding technique for instructions and possibly also for data with the following properties:
  - There should be little or no run-time penalty for these techniques.

- Decoding should work from a limited context (it is, for example, impossible to read the entire program to find the destination of a branch instruction).
- Word-sizes of the memory, of instructions and addresses have to be taken into account.
- Branch instructions branching to arbitrary destination addresses have to be supported.
- Fast encoding is only required if writable data is encoded. Otherwise, fast decoding is sufficient.

There are several variations of this scheme:

* For some processors, there is a **second instruction set**. This second instruction set has a narrower instruction format. An example of this is the ARM processor family. The ARM instruction set is a 32 bit instruction set. The ARM instruction set includes predicated execution. This means an instruction is executed if and only if a certain condition is met (see page 113). This condition is encoded in the first four bits of the instruction format. Most ARM processors also provide a second instruction set, with 16 bit wide instructions, called THUMB instructions. THUMB instructions are shorter, since they do not support predication, use shorter and less register fields and use shorter immediate fields. (see fig. 3.12).

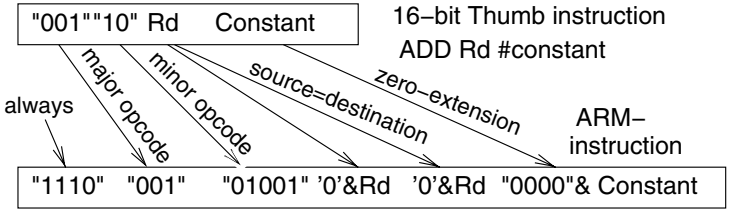

Figure 3.12. Re-encoding THUMB into ARM instructions

THUMB instructions are dynamically converted into ARM instructions while programs are running. THUMB instructions can use only half the registers in arithmetic instructions. Therefore, register-fields of THUMB instructions are concatenated with a '0'-bit[3]. In the THUMB instruction set, source and destination registers are identical and the length of constants that can be used, is reduced by 4 bits. During decoding, pipelining is used to keep the run-time penalty low.

[3]Using VHDL-notation (see page 59), concatenation is denoted by an &-sign and constants are enclosed in quotes in fig. 3.12.

Similar techniques also exist for other processors. The disadvantage of this approach is that the tools (compilers, assemblers, debuggers etc.) have to be extended to support a second instruction set. Therefore, this approach can be quite expensive in terms of software development cost.

* A second approach is the use of **dictionaries**. With this approach, each instruction pattern is stored only once. For each value of the program counter, a look-up table then provides a pointer to the corresponding instruction in the instruction table, the dictionary (see fig. 3.13).

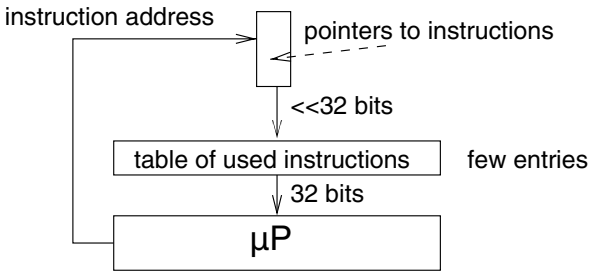

Figure 3.13. Dictionary approach for instruction compression

This approach relies on the idea that only very few different instruction patterns are used. Therefore, only few entries are required for the the instruction table. Correspondingly, the bit width of the pointers can be quite small. Many variations of this scheme exist. Some are called *two-level control store* [Dasgupta, 1979], *nanoprogramming* [Stritter and Gunter, 1979], or *procedure exlining* [Vahid, 1995].

A comprehensive survey over known compression techniques is available on the Internet [van de Wiel, 2002].

- **Run-time efficiency**: In order to meet time constraints without having to use high clock frequencies, architectures can be customized to certain application domains, such as digital signal processing (DSP). One can even go one step further and design application specific instruction set processors (ASIPs). As an example of domain-specific processors, we will consider processors for DSP. In digital signal processing, digital filtering is a very frequent operation. Equation 3.3 describes a digital filter generating an output sequence $y = (y_0, y_1, ...)$ from an input sequence $x = (x_0, x_1, ...)$.

$$y_i = \sum_{j=0}^{n-1} x_{i-j} * a_j \tag{3.3}$$

A certain output element y_i corresponds to a weighted average over the last n sequence elements of x and can be computed iteratively using equations 3.4 to 3.6.

$$y_{i,j} = y_{i,j-1} + x_{i-j} * a_j \tag{3.4}$$
$$\text{where} : y_{i,-1} = 0 \tag{3.5}$$
$$\text{and} : y_i = y_{i,n-1} \tag{3.6}$$

DSPs are designed such that each iteration can be encoded as a single instruction. Let us consider an example. Fig. 3.14 shows the internal architecture of an ADSP 2100 DSP processor.

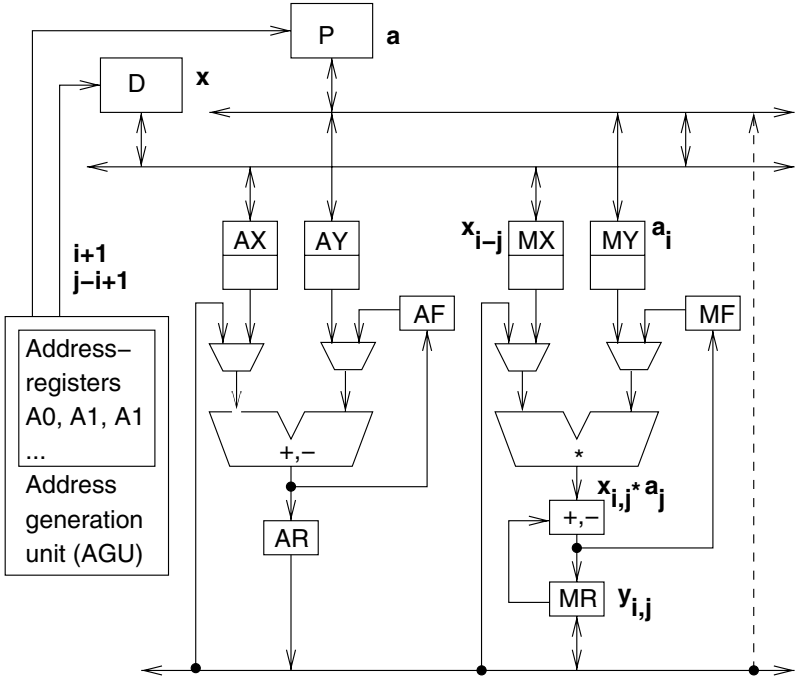

Figure 3.14. Internal architecture of the ADSP 2100 processor

The processor has two memories, called D and P. A special address generating unit (AGU) can be used to provide the pointers for accessing these

memories. There are separate units for additions and multiplications, each with their own argument registers AX, AY, AF, MX, MY and MF. The multiplier is connected to a second adder in order to compute series of multiplications and additions quickly.

For this processor, the update of the partial sum is essentially performed in a single cycle. For this purpose, the two memories are allocated to hold the two arrays x and a and address registers are allocated such that relevant pointers can be easily updated in the AGU. Partial sums $y_{i,j}$ are stored in MR. The pipelined computation involves registers A1, A2, MX and MY, as can be seen from the following implementation of the filter.

```
MR:=0;  A1:=1;  A2:=n-2;  MX:=x[n-1];  MY:=a[0];
for (j=1;  j<=n; j++)
 {MR:=MR + MX * MY;   MX:=x[A2];   MY:=a[A1];
  A1++; A2--  }
```

A single instruction encodes the loop body, comprising the following operations:

- reading of two arguments from argument registers MX and MY, multiplying them and adding the product to register MR storing values $y_{i,j}$,
- fetching the next elements of arrays a and x from memories P and D and storing them in argument registers MX and MY,
- updating pointers to the next arguments, stored in address registers A1 and A2,
- testing for the end of the loop.

This way, each iteration requires just a single instruction. In order to achieve this, several operations are performed in parallel. For given computational requirements, this (limited) form of parallelism leads to relatively low clock frequencies. Furthermore, the registers in this architecture perform different functions. They are said to be **heterogeneous**. Heterogeneous register files are a common characteristic for DSP processors. In order to avoid extra cycles for testing for the end of the loop, **zero-overhead loop instructions** are frequently provided in DSP processors. With such instructions, a single or a small number of instructions can be executed a fixed number of times. Processors not optimized for DSP would probably need several instructions per iteration and would therefore require a higher clock frequency, if available.

3.4.3.1 DSP-Processors

In addition to allowing single instruction realizations of loop bodies for filtering, DSP processors provide a number of other application-domain oriented features:

- **Specialized addressing modes:** In the filter application described above, only the last n elements of x need to be available. Ring buffers can be used for that. These can be implemented easily with modulo addressing. In modulo addressing, addresses can be incremented and decremented until the first or last element of the buffer is reached. Additional increments or decrements will result in addresses pointing to the other end of the buffer.

- **Separate address generation units:** Address generation units (AGUs) are typically directly connected to the address input of the data memory (see fig. 3.15).

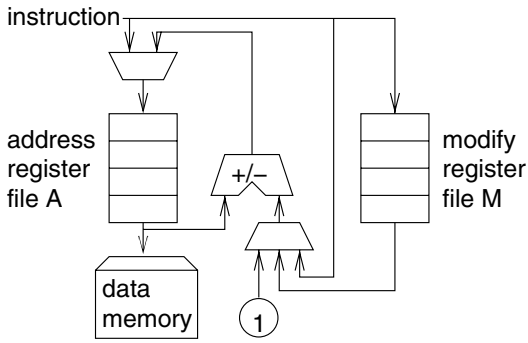

Figure 3.15. AGU using special address registers

Addresses which are available in address registers can be used in register-indirect addressing modes. This saves machine instructions, cycles and energy. In order to increase the usefulness of address registers, instruction sets typically contain auto-increment and -decrement options for most instructions using address registers.

- **Saturating arithmetic:** Saturating arithmetic changes the way overflows and underflows are handled. In standard binary arithmetic, wrap-around is used for the values returned after an overflow or underflow. Fig. 3.16 shows an example in which two unsigned four-bit numbers are added. A carry is generated which cannot be returned in any of the standard registers. The result register will contain a pattern of all zeros. No result could be further away from the true result than this one.

	0111
+	1001
Standard *wrap-around* arithmetic	10000
saturating arithmetic	1111

Figure 3.16. Wrap-around vs. saturating arithmetic for unsigned integers

In saturating arithmetic, we try to return a result which is as close as possible to the true result. For saturating arithmetic, the largest value is returned in the case of an overflow and the smallest value is returned in the case of an underflow. This approach makes sense especially for video and audio applications: the user will hardly recognize the difference between the true result value and the largest value that can be represented. Also, it would be useless to raise exceptions if overflows occur, since it is difficult to handle exceptions in real-time. Note that we need to know whether we are dealing with signed or unsigned add instructions in order to return the right value.

- **Fixed-point arithmetic:** Floating-point hardware increases the cost and power-consumption of processors. Consequently, it has been estimated that 80 % of the DSP processors do not include floating-point hardware [Aamodt and Chow, 2000]. However, in addition to supporting integers, many such processors do support fixed-point numbers. Fixed-point data types can be specified by a 3-tuple $(wl, iwl, sign)$, where wl is the total word-length, iwl is the integer word-length (the number of bits left of the binary point), and sign $s \in \{s, u\}$ denotes whether we are dealing with unsigned or signed numbers. See also fig. 3.17. Furthermore, there may be different rounding modes (e.g. truncation) and overflow modes (e.g. saturating and wrap-around arithmetic).

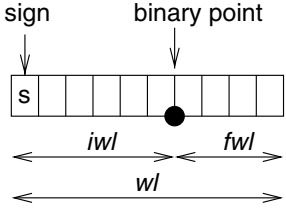

Figure 3.17. Parameters of a fixed-point number system

For fixed-point numbers, the position of the binary point is maintained after multiplication (some low order bits are truncated or rounded). For fixed-point processors, this operation is supported by hardware.

- **Real-time capability:** Some of the features of modern processors used in PCs are designed to improve the average execution time of programs. In many cases, it is difficult if not impossible to formally verify that they improve the worst case execution time. In such cases, it may be better not to implement these features. For example, it is difficult (though not impossible [Absint, 2002]) to guarantee a certain speedup resulting from the use of caches. Therefore, many embedded processors do not have caches. Also, virtual addressing and demand paging are normally not found in embedded systems.

- **Multiple memory banks or memories:** the usefulness of multiple memory banks was demonstrated in the ADSP 2100 example: the two memories D and P allow fetching both arguments at the same time. Several DSP processors come with two memory banks.

- **Heterogenous register files:** heterogenous register files were already mentioned for the filter application.

- **Multiply/accumulate instructions:** these instructions perform multiplications followed by additions. They were also already used in the filter application.

3.4.3.2 Multimedia processors

Registers and arithmetic units of many modern architectures are 64 bits wide. Therefore, two 32 bit data types ("double words"), four 16 bit data types ("words") or eight 8 bit data types ("bytes") can be packed into a single register (see fig. 3.18).

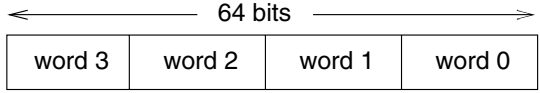

Figure 3.18. Using 64 bit registers for packed words

Arithmetic units can be designed such that they suppress carry bits at double word, word or byte boundaries. Multimedia instruction sets exploit this fact by supporting operations on packed data types. Such instructions are sometimes called single-instruction, multiple-data (SIMD) instructions, since a single instruction encodes operations on several data elements. With bytes packed into 64-bit registers, speed-ups of up to about eight over non-packed data types are possible. Data types are typically stored in packed form in memory. Unpacking and packing are avoided if arithmetic operations on packed data types are used. Furthermore, multimedia instructions can usually be combined with saturating

arithmetic and therefore provide a more efficient form of overflow handling than standard instructions. Hence, the overall speed-up achieved with multimedia instructions can be significantly larger than the factor of eight enabled by operations on packed data types.

3.4.3.3 Very long instruction word (VLIW) processors

Computational demands for embedded systems are increasing, especially when multimedia applications, advanced coding techniques or cryptography are involved. Performance improvement techniques used in high-performance microprocessors are not appropriate for embedded systems: driven by the need for instruction set compatibility, processors found, for example, in PCs spend a huge amount of resources and energy on automatically finding parallelism in application programs. Still, their performance is frequently not sufficient. For embedded systems, we can exploit the fact that instruction set compatibility with PCs is not required. Therefore, we can use instructions which explicitely identify operations to be performed in parallel. This is possible with **explicit parallelism instruction set computers** (EPICs). With EPICs, detection of parallelism is moved from the processor to the compiler[4]. This avoids spending silicon and energy on the detection of parallelism at runtime. As a special case, we consider very long instruction word (VLIW) processors. For VLIW processors, several operations or instructions are encoded in a long instruction word (sometimes called **instruction packet**) and are assumed to be executed in parallel. Each operation/instruction is encoded in a separate field of the instruction packet. Each field controls certain hardware units. Four such fields are used in fig. 3.19, each one controlling one of the hardware units.

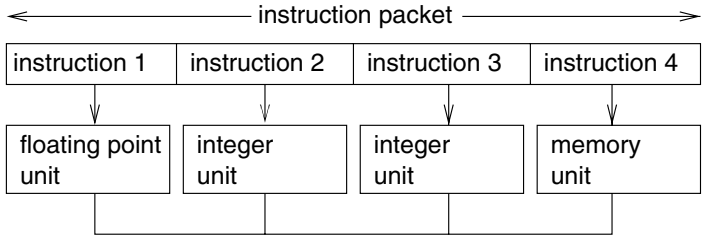

Figure 3.19. VLIW architecture (example)

For VLIW architectures, the compiler has to generate instruction packets. This requires that the compiler is aware of the available hardware units and to schedule their use.

[4]EPICs are sometimes also used for PCs [Transmeta, 2005, Intel, 2005]. However, legacy problems result in severe constraints for doing this.

Instruction fields must be present, regardless of whether or not the corresponding functional unit is actually used in a certain instruction cycle. As a result, the code density of VLIW architectures may be low, if insufficient parallelism is detected to keep all functional units busy. The problem can be avoided if more flexibility is added. For example, the Texas Instruments TMS 320C6xx family of processors implements a variable instruction packet size of up to 256 bits. In each instruction field, one bit is reserved to indicate whether or not the operation encoded in the next field is still assumed to be executed in parallel (see fig. 3.20). No instruction bits are wasted for unused functional units.

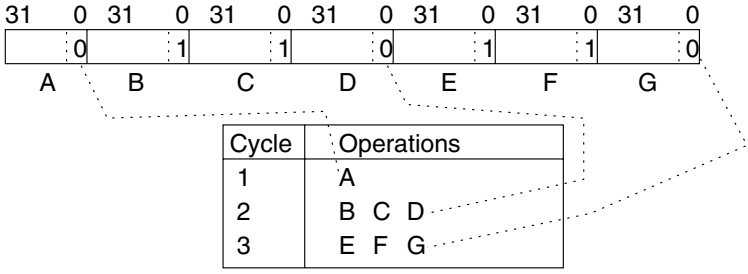

Figure 3.20. Instruction packets for TMS 320C6xx

Due to its variable length instruction packets, TMS 320C6xx processors do not quite correspond to the classical model of VLIW processors. Due to their explicit description of parallelism, they are EPIC processors, though.

Partitioned Register Files. Implementing register files for VLIW and EPIC processors is far from trivial. Due to the large number of operations that can be performed in parallel, a large number of register accesses has to be provided in parallel. Therefore, a large number of ports is required. However, the delay, size and energy consumption of register files increases with their number of ports. Hence, register files with very large numbers of ports are inefficient. As a consequence, many VLIW/EPIC architectures use partitioned register files. Functional units are then only connected to a subset of the register files. As an example, fig. 3.21 shows the internal structure of the TMS 320C6xx processors. These processors have two register files and each of them is connected to half of the functional units. During each clock cycle, only a single path from one register file to the functional units connected to the other register file is available.

Alternative partitionings are considered by Lapinskii et al. [Lapinskii et al., 2001].

Many DSP processors are actually VLIW processors. As an example, we are considering the M3-DSP processor [Fettweis et al., 1998]. The M3-DSP pro-

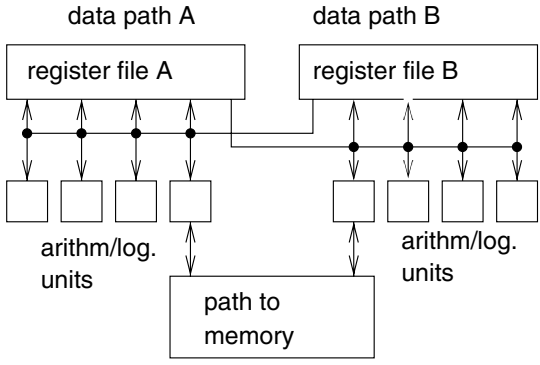

Figure 3.21. Partitioned register files for TMS 320C6xx

cessor is a VLIW processors containing (up to) 16 parallel data paths. These data paths are connected to a group memory, providing the necessary arguments in parallel (see fig. 3.22).

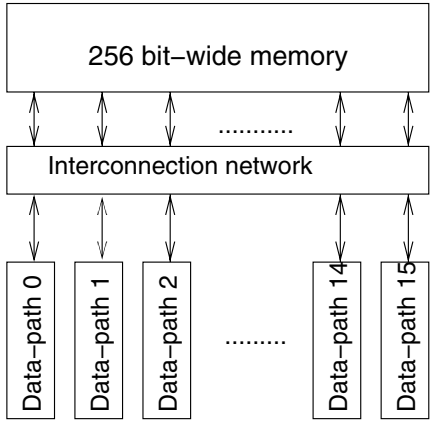

Figure 3.22. M3-DSP (simplified)

Predicated Execution. A potential problem of VLIW and EPIC architectures is their potentially large **delay penalty**: This delay penalty might originate from branch instructions found in some instruction packets. Instruction packets normally have to pass through pipelines. Each stage of these pipelines implements only part of the operations to be performed by the instructions executed. The fact that branch instructions exist cannot be detected in the first stage of the pipeline. When the execution of the branch instruction is finally completed, additional instructions have already entered the pipeline (see fig. 3.23).

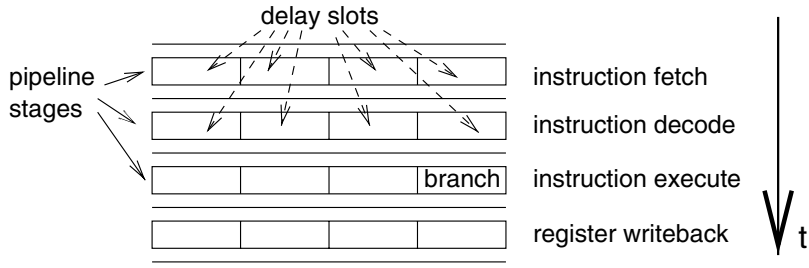

Figure 3.23. Branch instruction and delay slots

There are essentially two ways to deal with these additional instructions:

1 They are executed as if no branch had been present. This case is called **delayed branch**. Instruction packet slots that are still executed after a branch are called **branch delay slots**. These branch delay slots can be filled with instructions which would be executed before the branch if there were no delay slots. However, it is normally difficult to fill all delay slots with useful instructions and some have to be filled with no-operation instructions (NOPs). The term **branch delay penalty** denotes the loss of performance resulting from these NOPs.

2 The pipeline is stalled until instructions from the branch target address have been fetched. There are no branch delay slots in this case. In this organization the branch delay penalty is caused by the stall.

Branch delay penalties can be significant. For example, the TMS 320C6xx family of processors has up to 40 delay slots. Therefore, efficiency can be improved by avoiding branches, if possible. In order to avoid branches originating from if-statements, **predicated instructions** have been introduced. For each predicated instruction, there is a predicate. This predicate is encoded in a few bits and evaluated at run-time. If the result is true, the instruction is executed. Otherwise, it is effectively turned into a NOP. Predication can also be found in RISC machines such as the ARM processor. Example: ARM instructions, as introduced on page 104, include a four-bit field. These four bits encode various expressions involving the condition code registers. Values stored in these registers are checked at run-time. They determine whether or not a certain instruction has an effect.

Predication can be used to implement small if-statements efficiently: the condition is stored in one of the condition registers and if-statement-bodys are implemented as predicated instructions which depend on this condition. This way, if-statement bodys can be evaluated in parallel with other operations and no delay penalty is incurred.

3.4.3.4 Micro-controllers

A large number of the processors in embedded systems are in fact micro-controllers. Micro-controllers are typically not very complex and can be used easily. Due to their relevance for designing control systems, we introduce one of the most frequently used processors: the Intel 8051. This processor has the following characteristics:

- 8 bit CPU, optimized for control applications,
- large set of operations on Boolean data types,
- program address space of 64 k bytes,
- separate data address space of 64 k bytes,
- 4 k bytes of program memory on chip, 128 bytes of data memory on chip,
- 32 I/O lines, each of which can be addressed individually,
- 2 counters on the chip,
- universal asynchronous receiver/transmitter for serial lines available on the chip,
- clock generation on the chip,
- many variations commercially available.

All these characteristics are quite typical for micro-controllers.

3.4.4 Reconfigurable Logic

In many cases, full custom hardware chips (ASICs) are too expensive and software-based solutions are too slow or too energy consuming. Reconfigurable logic provides a solution if algorithms can be efficiently implemented in custom hardware. It can be almost as fast as special purpose hardware. In contrast to special purpose hardware, the performed function can be changed by using configuration data. Due to these properties, reconfigurable logic finds applications in the following areas:

- **Fast prototyping:** modern ASICs can be very complex and the design effort can be large and take a long time. It is therefore frequently desirable to generate a prototype, which can be used for experimenting with a system which behaves "almost" like the final system. The prototype can be more costly and larger than the final system. Also, its power consumption can

be larger than the final system, some timing constraints can be relaxed, and only the essential functions need to be available. Such a system can then be used for checking the fundamental behavior of the future system.

- **Low volume applications:** If the expected market volume is too small to justify the development of special purpose ASICs, reconfigurable logic can be the right hardware technology for applications, which cannot be implemented in software.

Reconfigurable hardware typically includes random access memory (RAM) to store configurations during normal operation of the hardware. Such RAM is normally **volatile** (the information is stored only while power is applied). Therefore, the configuration data must be copied into the configuration RAM at power-up. **Persistent** storage technology such as read-only memories (ROMs) and Flash memories will then provide the configuration data.

Field programmable gate arrays (FPGAs) are the most common form of reconfigurable hardware. As the name indicates, such devices are programmable "in the field" (after fabrication). Furthermore, they consist of arrays of processing elements. As an example, fig. 3.24 shows the array structure of Xilinx Virtex-II arrays (see http://www.xilinx.com).

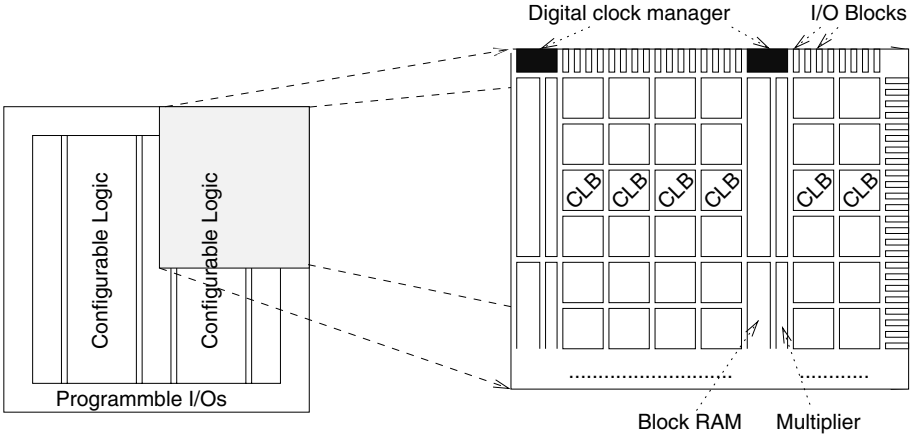

Figure 3.24. Floor-plan of Virtex II FPGAs

Currently (in 2003), Virtex II arrays contain up to 112 $\times$ 104 **configurable logic blocks** (CLBs). These can be connected using a programmable interconnect structure. Arrays also contain up to 1108 input/output connections and special clock processing blocks. In addition, there are up to 168 18 $\times$ 18 bit multipliers and 3024 kbits of RAM (Block RAM). Each CLB consists of 4 so-called slices (see fig. 3.25).

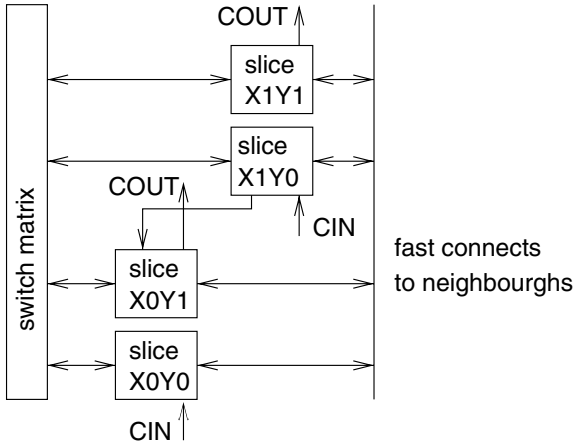

Figure 3.25. Virtex II CLB

Each slice contains two 16 bit memories F and G. These memories can be used as look-up tables (LUT) for implementing all 2^{16} Boolean functions of 4 variables. With the help of multiplexers (MUXF5, MUXFx), several of these memories can also be combined such that table look-ups for up to 8 variables are possible. They can also serve as ordinary RAM or as shift registers (SRLs). Each slice also includes two output registers and some special logic (ORCY, CY, etc.) for additions (see fig. 3.26).

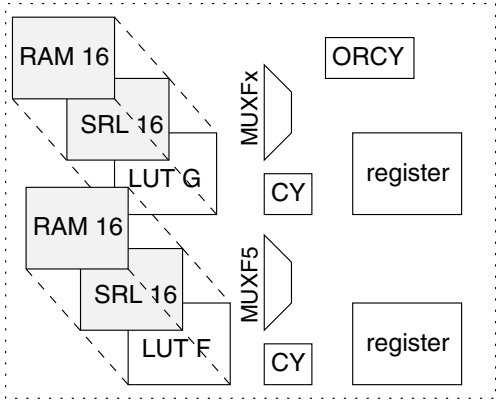

Figure 3.26. Virtex II Slice (simplified)

Configuration data determines the setting of multiplexers in the slices, the clocking of registers and RAM, the content of RAM components and the connections between CLBs. Typically, this configuration data is generated from a high-level description of the functionality of the hardware, for example in

VHDL. Ideally, the same description could also be used for generating ASICs automatically. In practice, some interaction is required.

Integration of reconfigurable computing with processors and software is simplified with the Virtex II Pro series of FPGAs from Xilinx. These FPGAs contain up to 4 Power-PC processors and faster I/O blocks.

3.5 Memories

Data, programs and FPGA configurations must be stored in some kind of memory. This must be done in an efficient way. Efficient means run-time, code-size and energy-efficient. Code-size efficiency requires a good compiler and can be improved with code compression (see page 103). Memory hierarchies can be exploited in order to achieve a good run-time and energy efficiency. The underlying reason is that large memories require more energy per access and are also slower than small memories.

Fig. 3.27 shows the cycle time and the power as a function of the size of the memory [Rixner et al., 2000]. The same behavior can be observed for larger memories.

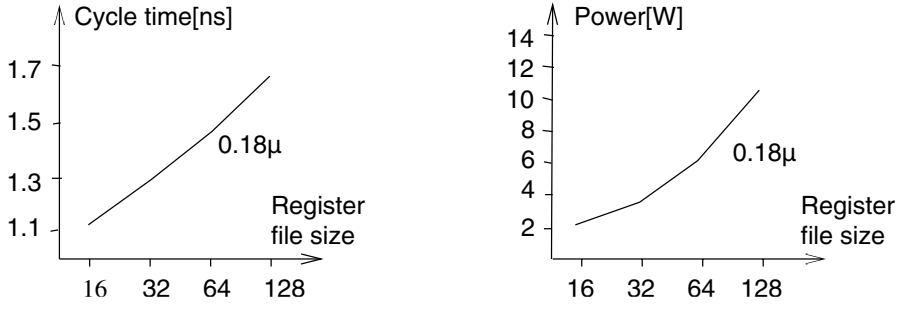

Figure 3.27. Cycle time and power as a function of the memory size

It has been observed that the difference in speeds between processors and memories is expected to increase (see fig. 3.28).

While the speed of memories is increasing by only a factor of about 1.07 per year, processor speeds are so far increasing by a factor of 1.5 to 2 per year [Machanik, 2002]. This means that the gap between processor speeds and memory speeds is becoming larger.

Therefore, it is important to use smaller and faster memories that act as buffers between the main memory and the processor. In contrast to PC-like systems, the architecture of these small memories must guarantee a predictable real-time performance. A combination of small memories containing frequently used data and instructions and a larger memory containing the remaining data

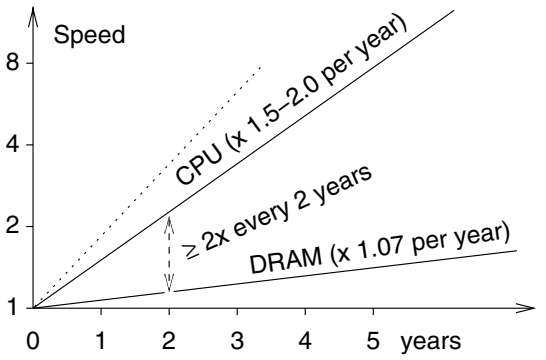

Figure 3.28. Increasing gap between processor and memory speeds

and instructions is generally also more energy efficient than a single, large memory.

Caches were initially introduced in order to provide good run-time efficiency. In the context of fig. 3.27 (right) however, it is obvious that caches potentially also improve the energy-efficiency of a memory system. Accesses to caches are accesses to small memories and therefore may require less energy per access than large memories. However, for caches it is required that the hardware checks whether or not the cache has a valid copy of the information associated with a certain address. This checking involves checking the tag fields of caches, containing a subset of the relevant address bits [Hennessy and Patterson, 1996]. Reading these tags requires additional energy. Also, the predictability of the real-time performance of caches is frequently low.

Alternatively, small memories can be mapped into the address space (see fig. 3.29).

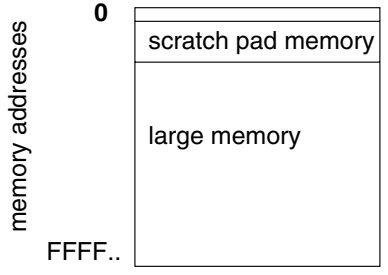

Figure 3.29. Memory map with scratch-pad included

Such memories are called **scratch pad memories** (SPMs). Frequently used variables and instructions should be allocated to that address space and no

checking needs to be done in hardware. As a result, the energy per access is reduced. Fig. 3.30 shows a comparison between the energy required per access to the scratch-pad (SPM) and the energy required per access to the cache.

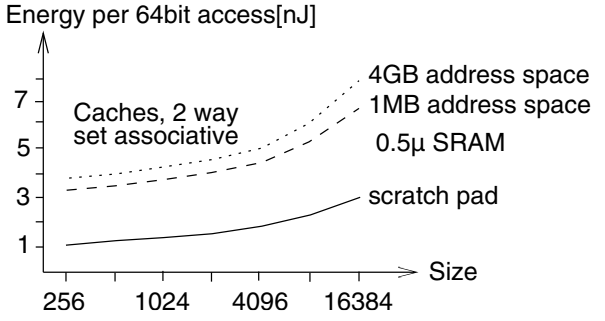

Figure 3.30. Energy consumption per scratch pad and cache access

For a two-way set associative cache, the two values differ by a factor of about three. The values in this example were computed using the energy consumption for RAM arrays as estimated by the CACTI cache estimation tool [Wilton and Jouppi, 1996].

SPMs can improve the memory access times very predictably, if the compiler is in charge of keeping frequently used variables in the SPM (see page 180).

3.6 Output

Output devices of embedded systems include

- **displays:** Display technology is an area which is extremely important. Accordingly, a large amount of information [Society for Display Technology, 2003] exists on this technology. Major research and development efforts lead to new display technology such as organic displays [Gelsen, 2003]. Organic displays are emitting light and can be fabricated with very high densities. In contrast to LCD displays, they do not need back-light and polarizing filters. Major changes are therefore expected in these markets.

- **electro-mechanical devices:** these influence the environment through motors and other electro-mechanical equipment.

Analog as well as digital output devices are used. In the case of analog output devices, the digital information must first be converted by digital-to-analog (D/A)-converters.

3.6.1 D/A-converters

D/A-converters are not very complex. Fig. 3.31 shows the schematic of a simple D/A converter.

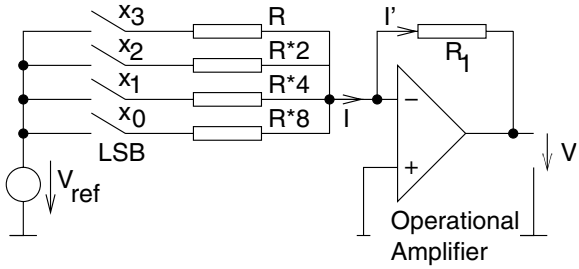

Figure 3.31. D/A-converter

The operational amplifier shown in fig. 3.31 amplifies the voltage difference between the two inputs by a very large factor (some powers of ten). Due to resistor R_1, resulting output voltages are fed back to input -. Whenever a small voltage between the two inputs exist, it will be inverted, amplified and fed back to the inputs, reducing the input voltage. Due to the large amplification, the differential voltage between the two inputs is reduced to virtually zero. Since input + is connected to ground, the voltage between input - and ground is virtually zero and the voltage at input - and ground is said to be virtually zero. "Virtually" means: zero except for some very small voltage resulting from a potentially imperfect operational amplifier and practically zero.

The key idea is to first generate a current which is proportional to the value represented by a bit-vector x and to convert this current into an equivalent voltage.

According to Kirchhoff's laws, current I is the sum of the currents through the resistors. The current through any resistor is zero, if the corresponding element of bit-vector x is '0'. If it is '1', the current corresponds to the weight of that bit, since resistor values are chosen accordingly.

$$\begin{aligned} I &= x_3 * \frac{V_{ref}}{R} + x_2 * \frac{V_{ref}}{2*R} + x_1 * \frac{V_{ref}}{4*R} + x_0 * \frac{V_{ref}}{8*R} \\ &= \frac{V_{ref}}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} \end{aligned} \tag{3.7}$$

Also, according to Kirchhoff's laws and due to the virtual zero at input -, we have: $V + R_1 * I' = 0$.

The current into the inputs of the operational amplifier are practically zero, and the two currents I and I' are equal: $I = I'$. Hence:

$$V + R_1 * I = 0 \tag{3.8}$$

From equations 3.7 and 3.8 we obtain:

$$-V = V_{ref} * \frac{R_1}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} = V_{ref} * \frac{R_1}{8 * R} * nat(x) \tag{3.9}$$

nat denotes the natural number represented by bit-vector x. Obviously, the output voltage is proportional to the value represented by x. Positive output voltages and bit-vectors representing two's complement numbers require minor extensions.

An interesting question is this one: suppose that the processors used in the hardware loop forward values from A/D-converters unchanged to the D/A-converters. Would it be possible to reconstruct the original analog voltage from the sensor outputs at the outputs of the D/A-converters? According to Nyquist's sampling theorem (see Oppenheim et al. [Oppenheim et al., 1999]), it is indeed possible, provided that the clock frequency of the sample-and-hold circuit is at least twice as large as the largest frequency found in the input voltage. This does, however, apply only if we have an infinite precision of the digital values. The limited precision of digital values effectively adds some noise to the digital signals [Oppenheim et al., 1999], which cannot be completely removed.

3.6.2 Actuators

There is a huge amount of actuators [Elsevier B.V., 2003a]. Actuators range from huge ones that are able to move tons of weight to tiny ones with dimensions in the μm area, like the one shown in fig. 3.32.

It is impossible to provide an overview. As an example, we mention only a special kind of actuators which will become more important in the future: microsystem technology enables the fabrication of tiny actuators, which can be put into the human body, for example.

Using such tiny actuators, the amount of drugs fed into the body can be adapted to the actual need. This allows a much better medication than needle-based injections. Fig. 3.32 shows a tiny motor manufactured with microsystem technology. The dimensions are in the μm range. The rotating center is controlled by electrostatic forces.

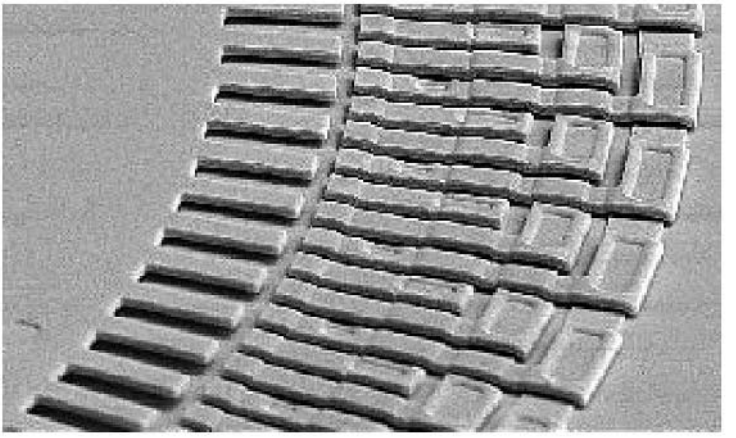

Figure 3.32. Microsystem technology based actuator motor (partial view; courtesy E. Obermeier, MAT, TU Berlin), ©TU Berlin

Chapter 4

STANDARD SOFTWARE: EMBEDDED OPERATING SYSTEMS, MIDDLEWARE, AND SCHEDULING

Not all components of embedded systems need to be designed from scratch. Instead, there are standard components that can be reused. These components comprise of knowledge from earlier design efforts and constitute **intellectual property** (IP). IP reuse is one key technique in coping with the increasing complexity of designs. Re-using available software components is at the heart of the platform-based design methodology, which will be briefly presented starting at page 151.

Standard software components that can be reused include: embedded operating systems (OS), real-time databases, and other forms of *middleware*. The last term denotes software providing an intermediate layer between the OS and application software (including, for example, libraries for communication). Calls to standard software components may already have to be included in the specification. Therefore, information about the application programming interface (API) of these standard components may already be needed for completing executable specifications.

Also, there are some standard approaches for scheduling which must be taken into account and which the designer must be aware of. Particular scheduling approaches may or may not be supported by a certain operating system. This constraint must also be taken into account.

Consistent with the design information flow, we will be describing embedded operating systems, middleware and scheduling in this chapter (see also fig. 4.1).

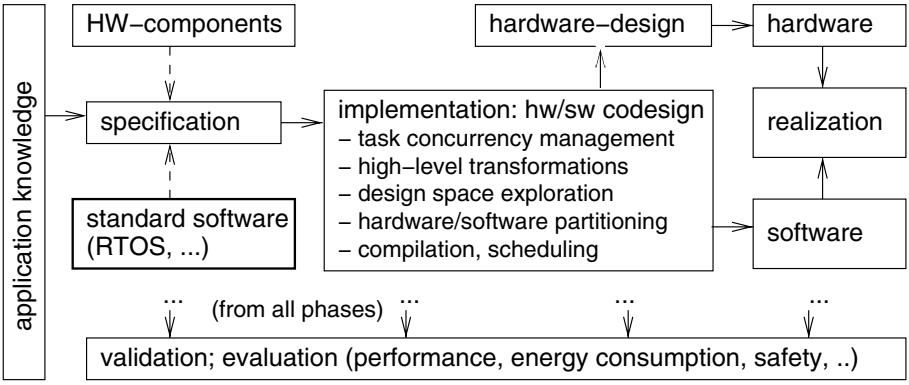

Figure 4.1. Simplified design information flow

4.1 Prediction of execution times

Scheduling of tasks requires some knowledge about the duration of task executions, especially if meeting time constraints has to be guaranteed, as is in real-time (RT) systems. The worst-case execution time is the basis for most scheduling algorithms.

Def.: The **worst-case execution time** (WCET) is an upper bound on the execution times of tasks.

Computing such a bound is undecidable in the general case. This is obvious from the fact that it is undecidable whether or not a program terminates. Hence, the WCET can only be computed for certain programs/tasks. For example, for programs without recursion and while loops and with constant iteration counts, the WCET can be computed.

Computing tight upper bounds on the execution time may still be difficult. Modern processor architectures' pipelines with their different kinds of hazards and memory hierarchies with limited predictability of hit rates are a source of serious overestimations of the WCET. Sometimes, architectural features which reduce the average execution time but cannot guarantee to reduce the WCET are completely omitted from the design (see page 110). Computing the WCET for systems containing caches and pipelines is a research topic (see, for example, Healy et al. [Healy et al., 1999] and the web pages of absint [Absint, 2002]). Interrupts and virtual memory (if present) result in more complications. Accordingly, it is already difficult to compute the WCET for assembly language programs. Computing tight bounds from a program written in a high-level language such as C without any knowledge of the generated assembly code is impossible.

The WCET may be required for different target technologies. If tasks are mapped to hardware, the WCET of that hardware needs to be computed. This may, in turn, require the synthesis of this hardware. Another approach is to respect timing constraints in hardware synthesis.

For some of the design phases, we might also need information on estimated average case execution times, in addition to WCETs. Two different approaches that have been proposed are the following:

- **Estimated cost and performance values:** Quite a number of estimators have been proposed for this purpose. Examples include the work by Jha and Dutt [Jha and Dutt, 1993] for hardware, and Jain et al. [Jain et al., 2001] for software. Generating sufficiently precise estimates requires some efforts.

- **Accurate cost and performance values:** This is only possible if interfaces to "software synthesis tools" (compilers) and hardware synthesis tools exist. This method can be more precise than the previous one, but may be significantly (and sometimes prohibitively) time consuming.

In order to find good estimates communication must also be considered. Unfortunately, it is very hard to predict the communication cost.

4.2 Scheduling in real-time systems

As indicated above, scheduling is one of the key issues in implementing RT-systems. Scheduling algorithms may be required a number of times during the design of RT-systems. Very rough calculations may already be required while fixing the specification. During hardware/software partitioning, somewhat more detailed predictions of execution times may be required. After compilation, even more detailed knowledge exists about the execution times and accordingly, more precise schedules can be made. Finally, it may be necessary to decide at run-time which task is to be executed next. Scheduling is somewhat linked to performance evaluation, mentioned at the bottom of figure 4.1. Like performance evaluation, it cannot be constrained to a single design step. We include scheduling in this chapter since it is closely linked to the RTOS, but the reader has to keep in mind that some of the scheduling technique are independent of the RTOS. In the case of design-time scheduling, RTOS scheduling may be limited to simple table look-ups for tasks to be executed.

4.2.1 Classification of scheduling algorithms

Scheduling algorithms can be classified according to various criteria. Fig. 4.2 shows a possible classification of algorithms (similar schemes are described in books on the topic [Balarin et al., 1998], [Kwok and Ahmad, 1999], [Stankovic et al., 1998], [Liu, 2000], [Buttazzo, 2002]).

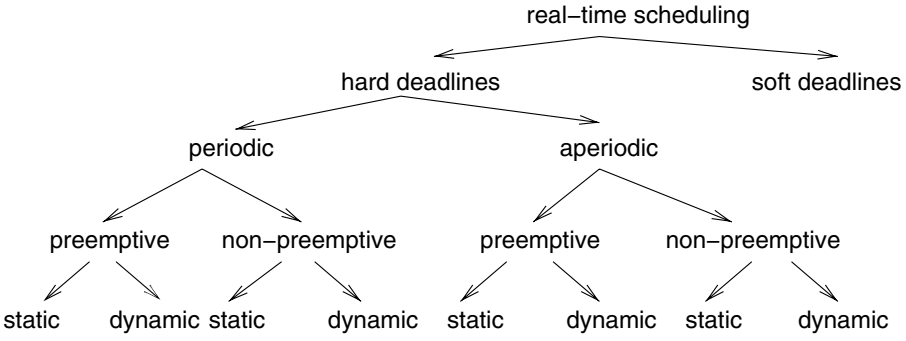

Figure 4.2. Classes of scheduling algorithms

- **Soft and hard deadlines:** Scheduling for soft deadlines is frequently based on extensions to standard operating systems. For example, providing task and operating system call priorities may be sufficient for systems with soft deadlines. We will not discuss these systems further in this book. More work and a detailed analysis is required for hard deadline systems. For these, we can use dynamic and static schedulers.

- **Scheduling for periodic and aperiodic tasks**

  In the following, we will distinguish between periodic, aperiodic and sporadic tasks.

  Definition: Tasks which must be executed once every p units of time are called **periodic tasks**, and p is called their **period**. Each execution of a periodic task is called a **job**.

  Definition: Tasks which are not periodic are called **aperiodic**.

  Definition: Aperiodic tasks requesting the processor at unpredictable times are called **sporadic**, if there is a minimum separation between the times at which they request the processor.

- **Preemptive and non-preemptive scheduling:** Non-preemptive schedulers are based on the assumption that tasks are executed until they are done. As a result the response time for external events may be quite long if some tasks have a large execution time. Preemptive schedulers have to be used

if some tasks have long execution times or if the response time for external events is required to be short.

- **Static and dynamic scheduling:** Dynamic schedulers take decisions at run-time. They are quite flexible, but generate overhead at run-time.

  Also, they are usually not aware of global contexts such as resource requirements or dependences between tasks. For embedded systems, such global contexts are typically available at design time and they should be exploited. Static schedulers take their decisions at design time. They are based on planning the start times of tasks and generate tables of start times forwarded to a simple dispatcher. The dispatcher does not take any decisions, but is just in charge of starting tasks at the times indicated in the table. The dispatcher can be controlled by a timer, causing the dispatcher to analyze the table. Systems which are totally controlled by a timer are said to be **entirely time triggered** (TT systems). Such systems are explained in detail in the book by Kopetz [Kopetz, 1997]:

  In an entirely time-triggered system, the temporal control structure of all tasks is established **a priori** *by off-line support-tools. This temporal control structure is encoded in a* **Task-Descriptor List (TDL)** *that contains the cyclic schedule for all activities of the node (Figure 4.3). This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary.* Figure 4.3 includes scheduled task start, task stop and send message (send) activities.

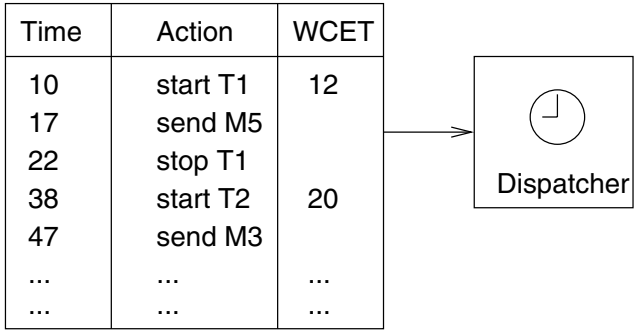

Figure 4.3. Task descriptor list in a TT operating system

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant ....

The main advantage of static scheduling is that it can be easily checked if timing constraints are met:

For satisfying timing constraints in hard real-time systems, predictability of the system behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system (according to Xu and Parnas, as cited by Kopetz).

The main disadvantage is that the response to sporadic events may be quite poor.

- **Centralized and distributed scheduling:** Multiprocessor scheduling algorithms can either be executed locally on one processor or can be distributed among a set of processors.

- **Type and complexity of schedulability test:**

  In practice, it is very important to know whether or not a schedule exists for a given set of tasks and constraints.

  A set of tasks is said to be **schedulable** under a given set of constraints, if a schedule exists for that set of tasks and constraints. For many applications, **schedulability tests** are important. Tests which never give wrong results (called exact tests) are NP-hard in many situations [Garey and Johnson, 1979]. Therefore, sufficient and necessary tests are used instead. For sufficient tests, sufficient conditions for guaranteeing a schedule are checked. There is a (hopefully small) probability of indicating that no schedule exists even if there exists one. Necessary tests are based on checking necessary conditions. They can be used to show that no schedule exists. However, there may be cases in which no schedule exists and we may still be unable to prove this.

- **Mono- and multi-processor scheduling:** Simple scheduling algorithms handle the case of single processors, whereas more complex algorithms also handle systems comprising multiple processors. For the latter, we can distinguish between algorithms for homogeneous multi-processor systems and algorithms for heterogenous multi-processor systems. The latter are able to handle target-specific execution times and can also be applied to mixed hardware/software systems, in which some tasks are mapped to hardware.

- **Online- and offline scheduling:** Online scheduling algorithms schedule tasks at run-time, based on the information about the tasks arrived so far. In contrast, offline algorithms schedule tasks, taking a priori knowledge about arrival times, execution times, and deadlines into account.

- **Cost function:** Different algorithms aim at minimizing different functions.

  Def.: Maximum lateness is defined as the difference between the completion time and the deadline, maximized over all tasks. Maximum lateness is negative if all tasks complete before their deadline.

- **Independent and dependent tasks:**

  It is possible to distinguish between tasks without any inter-task communication (in the following called simple, or S-tasks) and other tasks, called complex tasks. S-tasks can be in one out of two states: ready or running.

  The API of a TT-OS supporting S-tasks is quite simple: *The application program interface (API) of an S-task in a TT system consists of three data structures and two operating system calls. ... The system calls are TERMINATE TASK and ERROR. The TERMINATE TASK system call is executed whenever the task has reached its termination point. In case of an error that cannot be handled within the application task, the task terminates its operation with the ERROR system call* [Kopetz, 1997].

4.2.2 Aperiodic scheduling

4.2.2.1 Scheduling with no precedence constraints

Let $\{T_i\}$ be a set of tasks. Let (see fig. 4.4):

- c_i be the execution time of T_i,
- d_i be the **deadline interval**, that is, the time between T_i becoming available and the time until which T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as

$$l_i = d_i - c_i \tag{4.1}$$

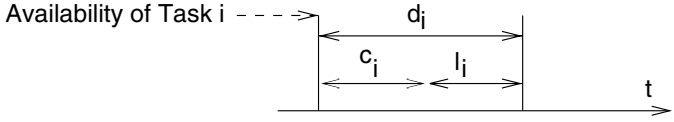

Figure 4.4. Definition of the laxity of a task

If $l_i = 0$, then T_i has to be started immediately after it becomes executable.

Let us first consider[1] the case of uni-processor systems for which all tasks arrive at the same time. If all tasks arrive at the same time, preemption is obviously useless.

[1]We are using some of the material from the book by Buttazzo [Buttazzo, 2002] for this section. Refer to this book for additional references.

A very simple scheduling algorithm for this case was found by Jackson in 1955 [Jackson, 1955]. The algorithm is based on Jackson's rule: *Given a set of n independent tasks, any algorithm that executes the tasks in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness.* The algorithm is called **Earliest Due Date** (EDD). EDD requires all tasks to be sorted by their deadlines. If the deadlines are known in advance, EDD can be implemented as a static scheduling algorithm. Hence, its complexity is $O(n\log(n))$.

Let us consider the case of different arrival times for uni-processor systems next. Under this scenario, preemption can potentially reduce maximum lateness.

The Earliest Deadline First (EDF) algorithm is optimal with respect to minimizing the maximum lateness. It is based on the following theorem [Horn, 1974]: *Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.* See Buttazzo [Buttazzo, 2002] for the proof of this property. EDF requires that, each time a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their deadlines. Hence, EDF is a dynamic scheduling algorithm. If a newly arrived task is inserted at the head of the queue, the currently executing task is **preempted**. If sorted lists are used for the queue, the complexity of EDF is $O(n^2)$. Bucket arrays could be used for reducing the execution time.

Fig. 4.5 shows a schedule derived with the EDF algorithm. Vertical bars indicate the arrival of tasks.

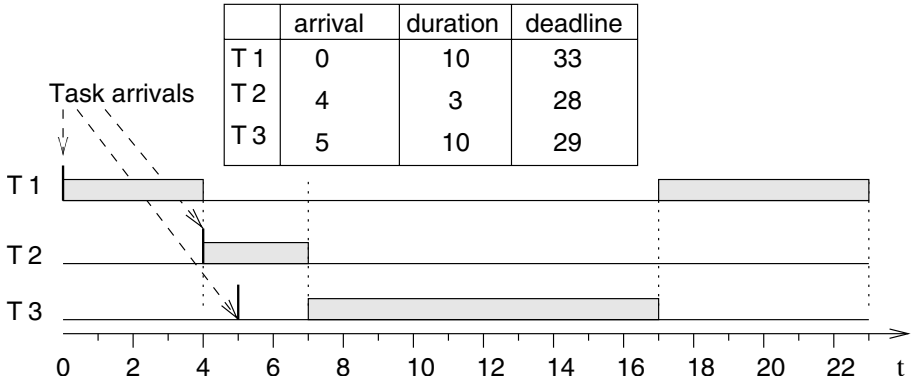

Figure 4.5. EDF schedule

At time 4, task T2 has an earlier deadline. Therefore it preempts T1. At time 5, task T3 arrives. Due to its later deadline it does not preempt T2.

Least Laxity (LL), Least Slack Time First (LST), and Minimum Laxity First (MLF) are three names for another scheduling strategy [Liu, 2000]. According to LL scheduling, task priorities are a monotonically decreasing function of the laxity (see equation 4.1; the less laxity, the higher the priority). The laxity is dynamically changing. LL scheduling is also preemptive. Fig. 4.6 shows an example of an LL schedule, together with the computations of the laxity.

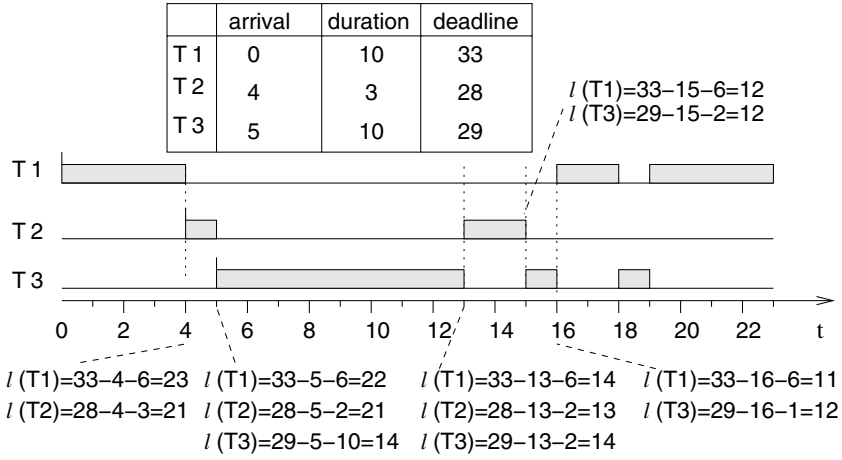

Figure 4.6. Least laxity schedule

At time 4, task T1 is preempted, as before. At time 5, T2 is now also preempted, due to the lower laxity of task T3.

It can be shown (this is left as an exercise in [Liu, 2000]) that LL is also an optimal scheduling policy for mono-processor systems in this sense that it will find a schedule if one exists. Due to its dynamic priorities, it cannot be used with a standard OS providing only fixed priorities. LL scheduling requires periodic checks of the laxity, and (in contrast to EDF scheduling) the knowledge of the execution time (and takes it into account).

If preemption is not allowed, optimal schedules may have to leave the processor idle at certain times in order to finish tasks with early deadlines arriving late.

Proof: Let us assume that an optimal non-preemptive scheduler (not having knowledge about the future) never leaves the processor idle. This scheduler will then have to schedule the example of fig. 4.7 optimally (it will have to find a schedule if one exists).

For the example of fig. 4.7 we assume we are given two tasks. Let T1 be a periodic process with an execution time of 2, a period of 4 and a deadline interval of 4. Let T2 be a task occasionally becoming available at times $4*n+1$

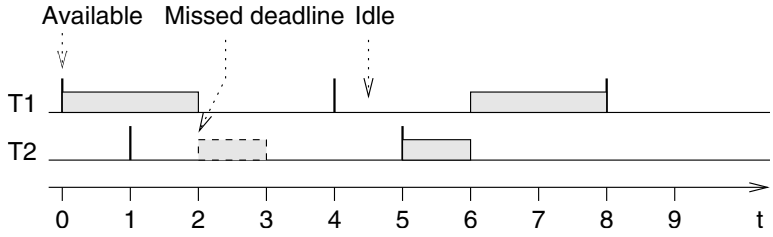

Figure 4.7. Scheduler needs to leave processor idle

and having an execution time and a deadline interval of 1. Let us assume that the concurrent execution of T1 and T2 is not possible due to some resource conflict. Under the above assumptions our scheduler has to start the execution of task T1 at time 0, since it is supposed not to leave any idle time. Since the scheduler is non-preemptive, it cannot start T2 when it becomes available at time 1. Hence, T2 misses its deadline. If the scheduler had left the processor idle (as shown in fig. 4.7 at time 4), a legal schedule would have been found. Hence, the scheduler is not optimal. This is a contradiction to the assumptions that optimal schedulers not leaving the processor idle at certain times exist. q.e.d.

We conclude: In order to avoid missed deadlines the scheduler needs knowledge about the future. If no knowledge about the arrival times is available a priori, then no online algorithm can decide whether or not to keep the processor idle. It has been shown that EDF is still optimal among all scheduling algorithms not keeping the processor idle at certain times. If arrival times are known a priori, the scheduling problem becomes NP-hard in general and branch and bound techniques are typically used for generating schedules.

4.2.2.2 Scheduling with precedence constraints

We start with a task graph reflecting tasks dependences (see fig. 4.8). Task T3 can be executed only after tasks T1 and T2 have completed and sent messages to T3.

This figure also shows a legal schedule. For static scheduling, this schedule can be stored in a table, indicating to the dispatcher the times at which tasks must be started and at which messages must be exchanged.

An optimal algorithm for minimizing the maximum lateness for the case of simultaneous arrival times was presented by Lawler [Lawler, 1973]. The algorithm is called **Latest Deadline First** (LDF). LDF is based on a total order compatible with the partial order described by the task graph. LDF reads the task graph and, among the tasks with no successors, moves the task with the

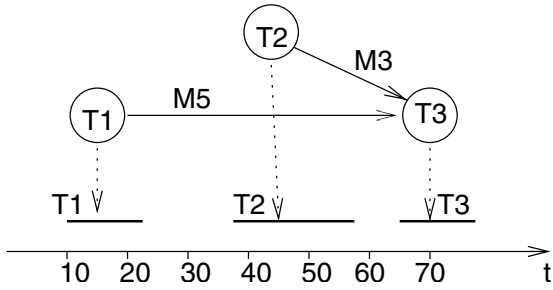

Figure 4.8. Precedence graph and schedule

latest deadline into a queue. It then repeats this process for the remaining tasks. If there is just a global time constraint, LDF essentially performs a **topological sort** [Sedgewick, 1988]. At run-time, the tasks are executed in the generated total order. LDF is non-preemptive and is optimal for mono-processors.

The case of asynchronous arrival times can be handled with a modified EDF algorithm. The key idea is to transform the problem from a given set of dependent tasks into a set of independent tasks with different timing parameters [Chetto et al., 1990]. This algorithm is again optimal for uni-processor systems.

If preemption is not allowed, the heuristic algorithm developed by Stankovic and Ramamritham [Stankovic and Ramamritham, 1991] can be used.

4.2.3 Periodic scheduling

4.2.3.1 Notation

Next, we will consider the case of periodic tasks. For periodic scheduling, the best that we can do is to design an algorithm which will always find a schedule if one exists. A scheduler is defined to be **optimal** iff it will find a schedule if one exists.

Let $\{T_i\}$ be a set of tasks. Each execution of some task T_i is called a **job**. The execution time for each job corresponding to one task is assumed to be the same. Let (see fig. 4.9)

- p_i be the period of task T_i,
- c_i be the execution time of T_i,
- d_i be the **deadline interval**, that is, the time between a job of T_i becoming available and the time after which the same job T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as

$$l_i = d_i - c_i \tag{4.2}$$

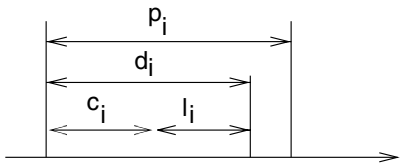

Figure 4.9. Notation used for time intervals

If $l_i = 0$, then T_i has to be started immediately after it becomes executable.

Let μ denote the **accumulated utilization** for a set of n processes, that is, the accumulated execution times of these processes divided by their period:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \tag{4.3}$$

Let us assume that the execution times are equal for a number of m processors. Obviously, equation 4.4 represents a necessary condition for a schedule to exist:

$$\mu \leq m \tag{4.4}$$

4.2.3.2 Independent tasks

Initially, we will restrict ourselves to a description of the case in which tasks are independent.

Rate monotonic scheduling. Rate monotonic (RM) scheduling [Liu and Layland, 1973] is probably the most well-known scheduling algorithm for independent periodic processes. Rate monotonic scheduling is based on the following assumptions ("**RM assumptions**"):

1 All tasks that have hard deadlines are periodic.

2 All tasks are independent.

3 $d_i = p_i$, for all tasks.

4 c_i is constant and is known for all tasks.

5 The time required for context switching is negligible.

6 For a single processor and for n tasks, the following equation holds for the accumulated utilization μ:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1) \tag{4.5}$$

Fig. 4.10 shows the right hand side of equation 4.5.

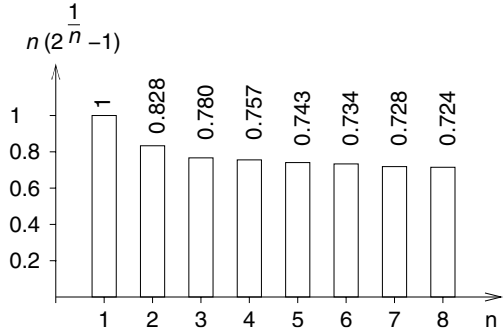

Figure 4.10. Right hand side of equation 4.5

The right hand side is about 0.7 for large n:

$$\lim_{n\to\infty} n * (2^{1/n} - 1) = ln(2) \ (=\sim 0.7) \tag{4.6}$$

Then, according to the policy for rate monotonic scheduling, **the priority of tasks is a monotonically decreasing function of their period**. In other words, tasks with a short period will get a high priority and tasks with a long period will be assigned a low priority. RM scheduling is a **preemptive scheduling policy** with **fixed priorities**. It is possible to prove that rate monotonic scheduling is optimal for mono-processor systems. Equation 4.5 requires that some of the computing power of the processor is not used in order to make sure that all requests are honored in time.

Fig. 4.11 shows an example of a schedule generated with RM scheduling.

Vertical bars indicate the arrival time of the tasks. Tasks 1 to 3 have a period of 2, 6 and 6, respectively. Execution times are, 0.5, 2, and 1. Task 1 has the shortest period and, hence, the highest rate and priority. Each time task 1 becomes available, its jobs preempt the currently active task. Task 2 has the same period as task 3, and neither of them preempts the other.

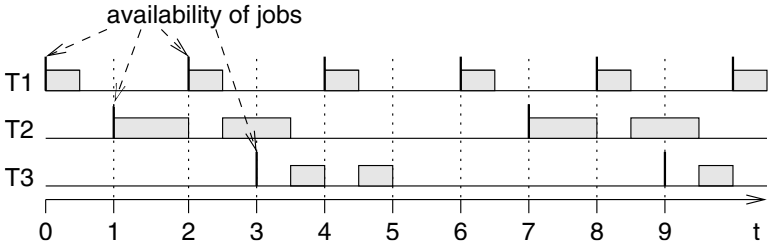

Figure 4.11. Example of a schedule generated with RM scheduling

RM scheduling has the following important advantages:

- RM scheduling is based on **static** priorities. This simplifies the OS and opens opportunities for using RM scheduling in a standard operating system providing fixed priorities, such as Windows NT (see Ramamritham [Ramamritham et al., 1998], [Ramamritham, 2002]).
- If the above six RM-assumptions (see page 136) are met, all deadlines will be met (see Buttazzo [Buttazzo, 2002]).

RM scheduling is also the basis for a number of formal proofs of schedulability.

Fig. 4.12 shows a case for which not enough idle time is available to guarantee schedulability for RM scheduling. One task is has a period of 5, and an execution time of 3, whereas the second task has a period of 8, and an execution time of 3. Task T2 is preempted several times.

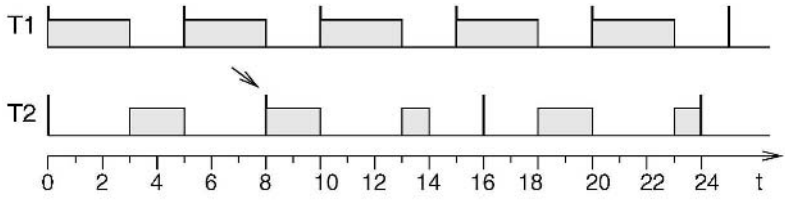

Figure 4.12. RM schedule does not meet deadline at time 8

For this particular case we have $\mu = \frac{3}{5} + \frac{3}{8} = \frac{39}{40}$, which is 0.975. On the other hand, $2 * (2^{\frac{1}{2}} - 1)$ is about 0.828. Hence, schedulability is not guaranteed for RM scheduling and, in fact, the deadline is missed at time 8. We assume that the missing computations are not scheduled in the next period.

However, this idle time or *spare capacity* of the processor is not always required. It is possible to show that RM scheduling is also optimal, iff instead of equation (4.5) we have

$$\mu \leq 1 \tag{4.7}$$

provided that the period of all tasks is a multiple of the period of the highest priority task.

Equations 4.5 or 4.7 provide easy means to check necessary conditions for schedulability.

Earliest deadline first scheduling. EDF can also be applied to periodic task sets. EDF can be extended to handle the case when deadlines are different from the periods.

It follows from the optimality of EDF for non-periodic schedules that EDF is also optimal for periodic schedules. No additional constraints have to be met to guarantee optimality. This implies that EDF is optimal also for the case of $\mu = 1$. Accordingly, no deadline is missed if the example of fig. 4.12 is scheduled with EDF (see fig. 4.13). At time 5, the behavior is different from that of RM-scheduling: due to the earlier deadline of T2, it is not preempted.

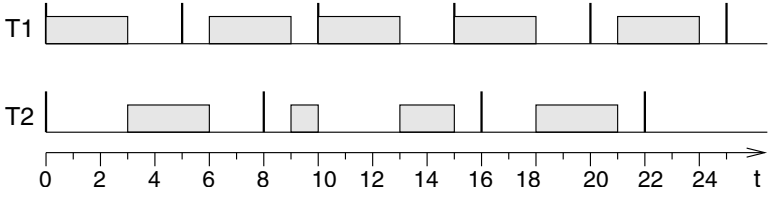

Figure 4.13. EDF generated schedule for the example of 4.12

Since EDF uses dynamic priorities, it cannot be used with a standard operating system providing only fixed priorities.

4.2.3.3 Dependent tasks

Scheduling dependent tasks is more difficult than scheduling independent tasks. The problem of deciding whether or not a schedule exists for a given set of dependent tasks and a given deadline is NP-complete [Garey and Johnson, 1979]. In order to reduce the scheduling effort, different strategies are used:

- adding additional resources such that scheduling becomes easier, and
- partitioning of scheduling into static and dynamic parts. With this approach, as many decisions as possible are taken at design time and only a minimum of decisions is left for run-time.

4.2.3.4 Sporadic events

We could connect sporadic events to interrupts and execute them immediately if their interrupt priority is the highest in the system. However, quite unpredictable timing behavior would result for all the other tasks. Therefore, special **sporadic task servers** are used which execute at regular intervals and check for ready sporadic tasks. This way, sporadic tasks are essentially turned into periodic tasks, thereby improving the predictability of the whole system.

4.2.4 Resource access protocols

4.2.4.1 Priority inversion

There are cases in which tasks must be granted exclusive access to resources such as global shared variables or devices in order to avoid non-deterministic or otherwise unwanted program behavior. Program sections during which such exclusive access is required are called **critical sections**. Operating systems typically provide primitives for requesting and releasing exclusive access to resources, also called **mutex primitives**. Tasks not being granted exclusive access have to wait until the resource is released. Accordingly, the release operation has to check for waiting tasks and resume the task of highest priority. We will call the request operation P(S) and the release operation V(S), where S corresponds to the particular resource requested. Critical sections should be short.

For tasks with critical sections, there is a crucial effect called **priority inversion**. An example of priority inversion is shown in fig. 4.15. We assume that the priority of task T1 is higher than that of task T2.

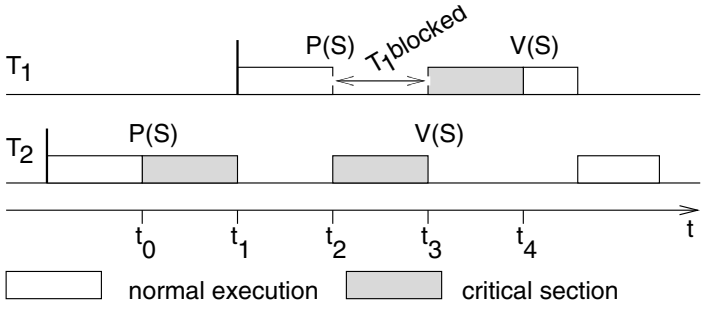

Figure 4.14. Priority inversion for two tasks

At time t_0, task T2 enters a critical section after requesting exclusive access to some resource via a operation P. At time t_1, task T1 becomes ready and preempts T2. At time t_2, T1 fails getting exclusive access to the resource in use by T2 and becomes blocked. Task T2 resumes and after some time releases

the resource. The release operation checks for pending tasks of higher priority and preempts T2. During the time T1 has been blocked, a lower priority task has effectively blocked a higher priority task. This effect is called **priority inversion**.

In the general case, priority inversion exists if some lower priority task is effectively preventing a higher priority task from executing due to the exclusive use of some resource. The necessity of providing exclusive access to some resources is the main reason for the priority inversion effect.

In the particular case of figure 4.14, the duration of the blocking cannot exceed the length of the critical section of T2. Unfortunately, there is no such upper bound in the general case. This can be seen from fig. 4.15.

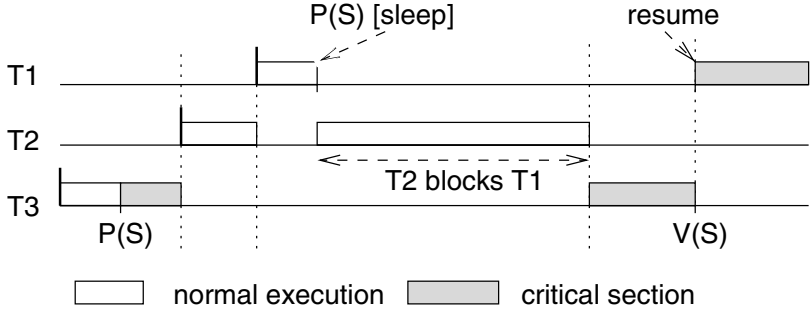

Figure 4.15. Priority inversion with potentially large delay

We assume that tasks T1, T2 and T3 are given. T1 has the highest priority, T2 has a medium priority and T3 has the lowest priority. Furthermore, we assume that T1 and T3 require exclusive use of some resource via operation P(S). Now, let T3 be in its critical section when it its preempted by T2. When T1 preempts T2 and tries to use the same resource that T3 is having exclusive access of, it blocks and lets T2 continue. As long as T2 is continuing, T3 cannot release the resource. Hence, T2 is effectively blocking T1 even though the priority of T1 is higher than that of T2. In this example, priority inversion continues as long as T2 executes. Hence, the duration of the priority inversion situation is not bounded by the length of any critical section.

One of the most prominent cases of priority inversion happened in the Mars Pathfinder, where an exclusive use of its a shared memory area led to priority inversion on Mars [Jones, 1997].

4.2.4.2 Priority inheritance

One way of dealing with priority inversion is to use the priority inheritance protocol. This protocol works as follows:

- Tasks are scheduled according to their active priorities. Tasks with the same priorities are scheduled on a first-come, first-served basis.
- When a task T1 executes P(S) and exclusive access is already granted to some other task T2, then T1 will become blocked. If the priority of T2 is lower than that of T1, T2 inherits the priority of T1. Hence, T2 resumes execution. In general, tasks inherit the highest priority of tasks blocked by it.
- When a task T2 executes V(S), its priority is decreased to the highest priority of the tasks blocked by it. If no other task is blocked by T2, its priority is reset to the original value. Furthermore, the highest priority task so far blocked on S is resumed.
- Priority inheritance is transitive: if T1 blocks T0 and T2 blocks T1, then T2 inherits the priority of T0.

In the example of fig. 4.15, T3 would inherit the priority of T1 when T1 executes P(S). This would avoid the problem mentioned since T2 could not preempt T3 (see fig. 4.16).

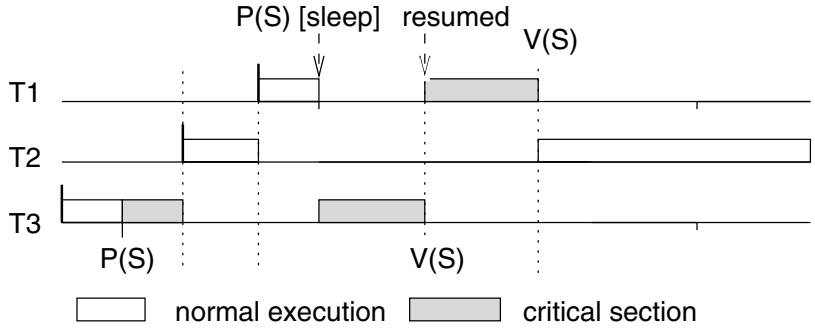

Figure 4.16. Priority inheritance for the example of fig. 4.15

Priority inheritance is also used by ADA: during a rendez-vous, the priority of both tasks is set to their maximum.

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to "on". When the software was shipped, it was set to "off". The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to "on", while the Pathfinder was already on the Mars [Jones, 1997].

While priority inheritance solves some problems, it does not solve others. There may be a large number of tasks having a high priority and there may

even be deadlocks. These problems are avoided with the more complex **priority ceiling protocol** [Sha et al., 1990].

4.3 Embedded operating systems

4.3.1 General requirements

Except for very simple systems, scheduling, task switching, and I/O require the support of an operating system suited for embedded applications. Task switch (or task "dispatch" algorithms multiplex processors such that each task seems to have its own (virtual) processor. The following are essential features of real-time and embedded operating systems:

- Due to the large variety of embedded systems, there is also a large variety of requirements for the functionality of embedded OSs. Due to efficiency requirements, it is not possible to work with OSs which provide the union of all functionalities. Hence, we need operating systems which can be **flexibly tailored** towards the application at hand. **Configurability** is therefore one of the main characteristics of embedded OSs. Configurability in its simplest form might just remove unused functions (to some extent, this can be done by a linker). In a more sophisticated form, conditional compilation can be employed (taking advantage of #if and #ifdef preprocessor commands). Dynamic data might be replaced by static data. Advanced compile-time evaluation and advanced compiler optimizations may also be useful in this context. Object-orientation could lead to a derivation of proper subclasses. Verification is a potential problem of systems with a large number of derived tailored OSs. Each and every derived OS must be tested thoroughly. Takada mentions this as a potential problem for eCos (an open source RTOS from Red Hat), comprising 100 to 200 configuration points [Takada, 2001].

- There is a large variety of peripheral devices employed in embedded systems. Many embedded systems do not have a hard disc, a keyboard, a screen or a mouse. There is effectively **no device that needs to be supported by all versions of the OS**, except maybe the system timer. Hence, it makes sense to handle relatively slow devices such as discs and networks by using special tasks instead of integrating their drivers into the kernel of the OS.

- **Protection mechanisms are not always necessary**, since embedded systems are typically designed for a single purpose and untested programs are hardly ever loaded. After the software has been tested, it can be assumed to be reliable (protection mechanisms may nevertheless still be needed for safety and security reasons). In most cases, embedded systems do not have

protection mechanisms. This also applies to input/output. In contrast to desktop applications, there is no desire to implement I/O instructions as privileged instructions and tasks can be allowed to do their own I/O. This matches nicely with the previous item and reduces the overhead of I/O operations.

Example: Let switch correspond to the (memory-mapped) I/O address of some switch which needs to be checked by some program. We can simply use a

```
load register,switch
```

instruction to query the switch. There is no need to go through an OS service call, which would create a lot of overhead for saving and restoring the task context (registers etc.).

- **Interrupts can be employed by any process**. For desktop applications, it would be a serious source of unreliability to allow any process to use interrupts directly. Since embedded programs can be considered to be thoroughly tested, since protection is not necessary and since efficient control over a variety of devices is required, it is possible to let interrupts directly start or stop tasks (e.g. by storing the tasks start address in the interrupt vector address table). This is substantially more efficient than going through OS services for the same purpose. However, composability may suffer from this: if a specific task is directly connected to some interrupt, then it may be difficult to add another task which also needs to be started by some event.

- Many embedded systems are real-time (RT) systems and, hence, the OS used in this systems **must be a real-time operating system** (RTOS).

4.3.2 Real-time operating systems

Def.: (A) *real-time operating system is an operating system that supports the construction of real-time systems* [Takada, 2001].

What does it take to make an OS an RTOS? The following are the three key requirements[2]:

- **The timing behavior of the OS must be predictable.** For each service of the OS, an upper bound on the execution time must be guaranteed. In practice, there are various levels of predictability. For example, there may be sets of OS service calls for which an upper bound is known and for

[2]This section includes information from Hiroaki Takada's tutorial [Takada, 2001] on real-time operating systems at the Asian South-Pacific Design Automation Conference (ASP-DAC) in 2001 for our description of RTOSs.

which there is not a significant variation of the execution time. Calls like "get me the time of the day" may fall into this class. For other calls, there may be a huge variation. Calls like "get me 4MB of free memory" may fall into this second class. In particular, the scheduling policy of RTOSs must be deterministic (standard Java fails badly in this respect, as no order of execution for a number of executable "threads" is specified). As another special case we mention garbage collection. In the Java context, various attempts have been made to provide predictable garbage collection (see page 58).

There may also be times during which interrupts have to be disabled to avoid interference between tasks (this is a very simple way of guaranteeing mutual exclusion on a mono-processor system). The periods during which interrupts are disabled have to be quite short in order to avoid unpredictable delays in the processing of critical events.

For RTOSs implementing file systems, it may be necessary to implement contiguous files (files stored in contiguous disc areas) to avoid unpredictable disc head movements.

- **The OS must manage the timing and scheduling of tasks**. Scheduling can be defined as mapping from the set of tasks to intervals of execution time, including the mapping to start times as a special case. Also, the OS possibly has to be aware of task deadlines so that the OS can apply appropriate scheduling techniques (there are, however, cases in which scheduling is completely done off-line and the OS only needs to provide services to start tasks at specific times or priority levels).

  The OS must provide precise time services with a high resolution. Time services are required, for example, in order to distinguish between original and subsequent errors. For example, they can help to identify the power plant(s) that are responsible for a blackout such as the one on America's East Coast in 2003. Time services and global synchronization of clocks are described in detail in the book by Kopetz [Kopetz, 1997].

- **The OS must be fast**. In addition to being predictable, the OS must be capable of supporting applications with deadlines that are fractions of a second.

Each RTOS includes a so-called real-time OS **kernel**. This kernel manages the resources which are found in every system, including the processor, the memory and the system timer. Protection mechanisms (except for dependability, safety or privacy reasons) need not be present.

There are two types of RTOSs:

- **General purpose OS type RTOSs:** for these operating systems, some drivers, such as disk, network drivers, or audio drivers are implicitly assumed to be present, and they are embedded into the kernel. The application software and middleware are implemented on top of the application programming interface, which is standard for all applications (see fig. 4.17).

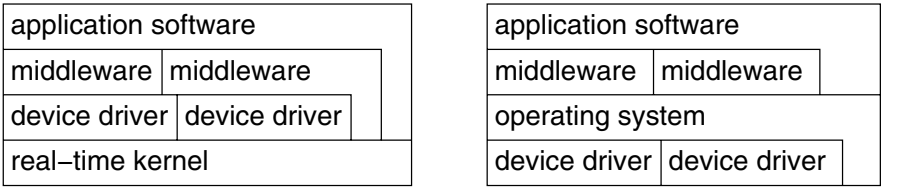

Figure 4.17. Real-time kernel (left) vs. general purpose OS (right)

- **Real-time kernel type of RTOSs:** since there is hardly any standard device in embedded systems, device drivers are not deeply embedded into the kernel, but are implemented on top of the kernel. Only the necessary drivers are included. Applications and middleware may be implemented on top of appropriate drivers, not on top of a standardized API of the OS.

Major functions in the kernel include the task management, inter-task synchronization and communication, time management and memory management.

While some RTOSs are designed for general embedded applications, others focus on a specific area. For example, OSEK/VDX OS focuses on automotive control. Due to this focus, it is a rather compact OS.

Similarly, while some RTOSs provide a standard API, others come with their own, proprietary API. For example, some RTOSs are compliant with the POSIX RT-extension [Harbour, 1993] for UNIX, with OSEK/VDX OS, or with the ITRON specification developed in Japan. Many RT-kernel type of OSs have their own API. ITRON, mentioned in this context, is a mature RTOS which employs link-time configuration.

Currently available RTOSs can further be classified into the following three categories [Gupta, 1998]:

- **Fast proprietary kernels:** According to Gupta, *for complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect*. Examples include QNX, PDOS, VCOS, VTRX32, VxWORKS.

- **Real-time extensions to standard OSs:** In order to take advantage of comfortable main stream operating systems, hybrid systems have been developed. For such systems, there is an RT-kernel running all RT-tasks. The

standard operating system is then executed as one of the tasks (see fig. 4.18).

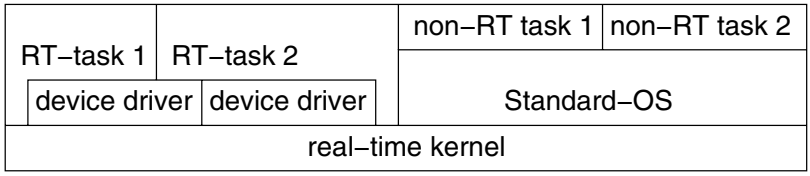

Figure 4.18. Hybrid OSs

This approach has some advantages: the system can be equipped with a standard OS API, can have graphical user interfaces (GUIs), file systems etc. and enhancements to standard OSs become quickly available in the embedded world as well. Also, problems with the standard OS and its non-RT tasks do not negatively affect the RT-tasks. The standard OS can even crash and this would not affect the RT-tasks. On the down side, and this is already visible from fig. 4.18, there may be problems with device drivers, since the standard OS will have its own device drivers. In order to avoid interference between the drivers for RT-tasks and those for the other tasks, it may be necessary to partition devices into those handled by RT-tasks and those handled by the standard OS. Also, RT-tasks cannot use the services of the standard OS. So all the nice features like file-system access and GUIs are normally not available to those tasks, even though some attempts may be made to bridge the gap between the two types of tasks without loosing the RT-capability. RT-Linux is an example of such hybrid OSs.

According to Gupta, trying to use a version of a standard OS is *not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation*. Indeed, dependences between tasks are not very frequent for most applications of standard operating systems and are therefore frequently ignored by such systems. This situation is different for embedded systems, since dependences between tasks are quite common and they should be taken into account. Unfortunately, this is not always done if extensions to standard operating systems are used. Furthermore, resource allocation and scheduling are rarely combined for standard operating systems. However, integrated resource allocation and scheduling algorithms are required in order to guarantee meeting timing constraints.

- There is a number of **research systems** which aim at avoiding the above limitations. These include Melody [Wedde and Lind, 1998], and (accord-

ing to Gupta [Gupta, 1998]) MARS, Spring, MARUTI, Arts, Hartos, and DARK.

Takada [Takada, 2001] mentions low overhead memory protection, temporal protection of computing resources (how to avoid tasks from computing for longer periods of time than initially planned), RTOSs for on-chip multiprocessors (especially for heterogenous multiprocessors and multi-threaded processors) and support for continuous media and quality of service control as research issues.

Due to the potential growth in the embedded system market, vendors of standard OSs are actively trying to sell variations of their products (like Embedded Windows XP and Windows CE [Microsoft Inc., 2003]) and obtain market shares from traditional vendors such as Wind River Systems [Wind River Systems, 2003].

4.4 Middleware

4.4.1 Real-time data bases

Data bases provide a convenient and structured way of storing and accessing information. Accordingly, data bases provide an API for writing and reading information. A sequence of read and write operations is called a **transaction**. Transactions may have to aborted for a variety of reasons: there could be hardware problems, deadlocks, problems with concurrency control etc. A frequent requirement is that transactions do not affect the state of the data base unless they have been executed to their very end. Hence, changes caused by transactions are normally not considered to be final until they have been **committed**. Most transactions are required to be **atomic**. This means that the end result (the new state of the data base) generated by some transaction must be the same as if the transaction has been fully completed or not at all. Also, the data base state resulting from a transaction must be **consistent**. Consistency requirements include, for example, that the values from read requests belonging to the same transaction are consistent (do not describe a state which never existed in the environment modeled by the data base). Furthermore, to some other user of the data base, no intermediate state resulting from a partial execution of a transaction must be visible (the transactions must be performed as if they were executed in **isolation**). Finally, the results of transactions should be persistent. This property is also called **durability** of the transactions. Together, the four properties printed in bold are known as ACID properties (see the book by Krishna and Shin [Krishna and Shin, 1997], chapter 5).

For some data bases, there are soft real-time constraints. For example, time-constraints for airline reservation systems are soft. In contrast, there may also

be hard constraints. For example, automatic recognition of pedestrians in automobile applications and target recognition in military applications must meet hard real-time constraints. The above requirements make it very difficult to guarantee hard real-time constraints. For example, transactions may be aborted various times before they are finally committed. For all data bases relying on demand paging and on hard discs, the access times to discs are hardly predictable. Possible solutions include main memory data bases. Embedded data bases are sometimes small enough to make this approach feasible. In other cases, it may be possible to relax the ACID requirements. For further information, see the book by Krishna and Shin.

4.4.2 Access to remote objects

There are special software packages which facilitate the access to remote services. CORBA®(Common Object Request Broker Architecture) [Object Management Group (OMG), 2003] is one example of this. With CORBA, remote objects can be accessed through standardized interfaces. Clients are communicating with local stubs, imitating the access to the remote objects. These clients send information about the object to be accessed as well as parameters (if any) to the Object Request Broker ORB (see fig. 4.19). The ORB then determines the location of the object to be accessed and sends information via a standardized protocol, e.g. the IIOP protocol to where the object is located. This information is then forward to the object via a skeleton and the information requested from the object (if any) is returned using the ORB again.

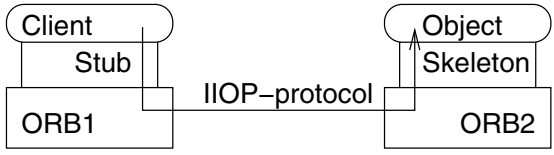

Figure 4.19. Access to remote objects using CORBA

Standard CORBA does not provide the predictability required for real-time applications. Therefore, a separate real-time CORBA (RT-CORBA) standard has been defined [Object Management Group (OMG), 2002]. A very essential feature of RT-CORBA is to provide *end-to-end predictability of timeliness in a fixed priority system*. This involves *respecting thread priorities between client and server for resolving resource contention*, and bounding the latencies of operation invocations. One particular problem of real-time systems is that thread priorities might not be respected when threads obtain mutually exclusive access to resources. This so-called priority inversion problem (see page 140) has to be addressed in RT-CORBA. RT-CORBA includes provisions for bounding

the time during which such priority inversion can happen. RT-CORBA also includes facilities for thread priority management. This priority is independent of the priorities of the underlying operating system, even though it is compatible with the real-time extensions of the POSIX standard for operating systems [Harbour, 1993]. The thread priority of clients can be propagated to the server side. Priority management is also available for primitives providing mutually exclusive access to resources. The priority inheritance protocol (described on page 141) must be available in implementations of RT-CORBA. Pools of pre-existing threads avoid the overhead of thread creation and thread-construction.

As an alternative to CORBA, the message passing interface (MPI) can be used for communicating between different processors. In order to apply the MPI-style of communication to real-time systems, a real-time version of MPI, called MPI/RT has been defined [MPI/RT forum, 2001]. MPI-RT does not cover some of the issues covered in RT-CORBA, such as thread creation and termination. MPI/RT is conceived as a potential layer between the operating system and standard (non real-time) MPI.

Chapter 5

IMPLEMENTING EMBEDDED SYSTEMS: HARDWARE/SOFTWARE CODESIGN

Once the specification has been completed, design activities can start. This is consistent with the simplified design information flow (see fig. 5.1).

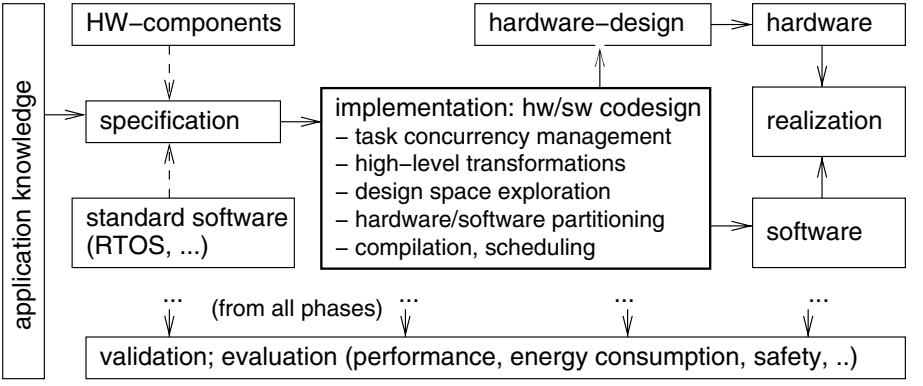

Figure 5.1. Simplified design information flow

It is a characteristic of embedded systems that both hardware and software have to be considered during their design. Therefore, this type of design is also called **hardware/software codesign**. The overall goal is to find the right combination of hardware and software resulting in the most efficient product meeting the specification. Therefore, embedded systems cannot be designed by a synthesis process taking only the behavioral specification into account. Rather, available components have to be accounted for. There are also other reasons for this constraint: in order to cope with the increasing complexity of embedded systems and their stringent time-to-market requirements, reuse is essentially unavoidable. This led to the term **platform-based design:**

A platform is a family of architectures satisfying a set of constraints imposed to allow the reuse of hardware and software components. However, a hardware platform is not enough. Quick, reliable, derivative design requires using a platform application programming interface (API) to extend the platform toward application software. In general, a platform is an abstraction layer that covers many possible refinements to a lower level. Platform-based design is a meet-in-the-middle approach: In the top-down design flow, designers map an instance of the upper platform to an instance of the lower, and propagate design constraints [Sangiovanni-Vincentelli, 2002].

The mapping is an iterative process in which performance evaluation tools guide the next assignment. Fig. 5.2 [Herrera et al., 2003a] visualizes this approach.

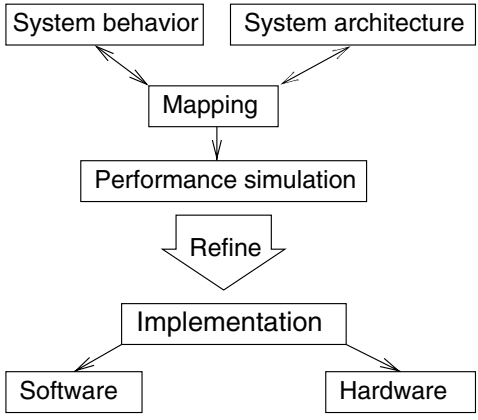

Figure 5.2. Platform-based design

Design activities have to take the existence of available platforms into account. There is actually a large number of design activities, only some of which can be presented here and references to available platforms will not always be explicit. Design activities that are presented include:

- **Task level concurrency management:** This activity is concerned with identifying the tasks that should be present in the final embedded system. These tasks may be different from those that were included in the specification, since there are good reasons for merging and splitting tasks (see section 5.5.1).
- **High-level transformations:** It has been found that there are many optimizing high-level transformations that can be applied to specifications. For example, loops can be interchanged so that accesses to array components become more local. Also, floating point arithmetic can frequently

be replaced by fixed-point arithmetic without any significant loss in quality. These high-level transformations are typically beyond the capabilities of available compilers and have to be applied before any compilation is started.

- **Hardware/software partitioning:** We assume that in the general case, some function has to be performed by special hardware due to increasing computational requirements [De Man, 2002]. Hardware/software partitioning is the activity in charge of mapping operations to either hardware or software.

- **Compilation:** Those parts of the specification that are mapped to software have to be compiled. Efficiency of the generated code is improved if the compiler exploits knowledge about the underlying processor (and possibly the memory) hardware. Therefore, there are special "hardware-aware" compilers for embedded systems.

- **Scheduling:** Scheduling (mapping of operations to start times) has to be performed in several contexts. Schedules have to be approximated during hardware/software partitioning, during task level concurrency management and possibly also during compilation. Precise schedules can be obtained for the final code.

- **Design space exploration:** In most of the cases, several designs meet the specifications. Design space exploration is the process of analyzing the set of possible designs. Among those designs that meet the specifications, one design has to be selected.

Particular design flows may use these activities in different orders. There is no standard set of design activities. We will briefly mention some orders that are being used at the end of this chapter (see page 190) in order to provide a some ideas on how actual design flows can look like.

5.1 Task level concurrency management

As mentioned on page 52, the task graphs' **granularity** is one of their most important properties. Even for hierarchical task graphs, it may be useful to change the granularity of the nodes. The partitioning of specifications into tasks or processes does not necessarily aim at the maximum implementation efficiency. Rather, during the specification phase, a clear separation of concerns and a clean software model are more important than caring about the implementation too much. Hence, there will not necessarily be a one-to-one correspondence between the tasks in the specification and those in the imple-

mentation. This means that a regrouping of tasks may be advisable. Such a regrouping is indeed feasible by merging and splitting of tasks.

Merging of task graphs can be performed whenever some task T_i is the immediate predecessor of some other task T_j and if T_j does not have any other immediate predecessor (see fig. 5.3 with $T_i = T_3$ and $T_j = T_4$). This transformation can lead to a reduced overhead of context-switches if the node is implemented in software, and it can lead to a larger potential for optimizations in general.

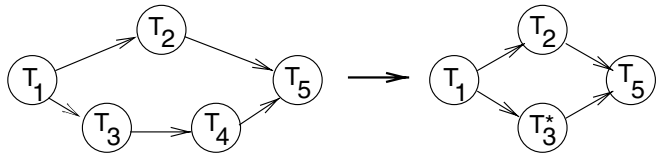

Figure 5.3. Merging of tasks

On the other hand, splitting of tasks may be advantageous for the following reasons:

Tasks may be holding resources (like large amounts of memory) while they are waiting for some input. In order to maximize the use of these resources, it may be best to constrain the use of these resources to the time intervals during which these resources are actually needed. In fig. 5.4, we are assuming that task T_2 requires some input somewhere in its code. In the initial version, the execution of task T_2 can only start if this input is available. We can split the node into T_2^* and T_2^{**} such that the input is only required for the execution of T_2^{**}. Now, T_2^* can start earlier, resulting in more scheduling freedom. This improved scheduling freedom might improve resource utilization and could even enable meeting some deadline. It may also have an impact on the memory required for data storage, since T_2^* could release some of its memory before terminating and this memory could be used by other tasks while T_2^{**} is waiting for input.

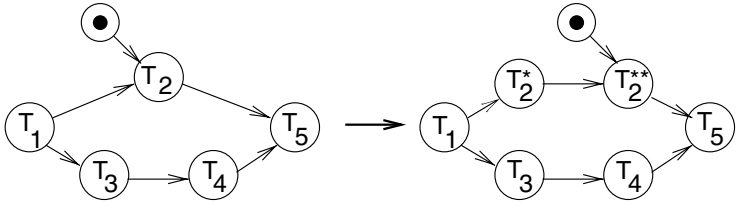

Figure 5.4. Splitting of tasks

One might argue that the tasks should release resources like large amounts of memory anyway before waiting for input. However, the readability of the original specification could suffer from caring about implementation issues in an early design phase.

Quite complex transformations of the specifications can be performed with a Petri-net based technique described by Cortadella et al. [Cortadella et al., 2000]. Their technique starts with a specification consisting of a set of tasks described in a language called *FlowC*. FlowC extends C with process headers and intertask communication specified in the form of READ- and WRITE-function calls. Fig. 5.5 shows an input specification using FlowC.

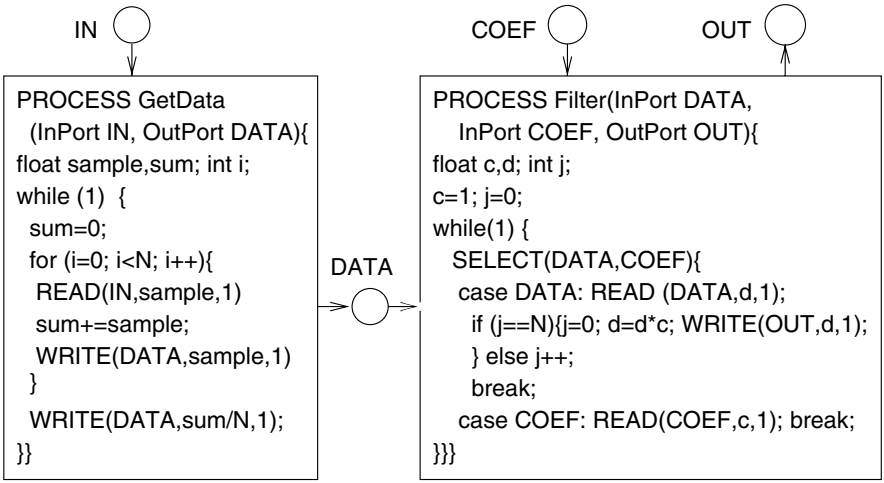

Figure 5.5. System specification

The example uses input ports IN and COEF, as well as output port OUT. Point-to-point interprocess communication between processes is realized through a uni-directional buffered channel DATA. Task GetData reads data from the environment and sends it to channel DATA. Each time N samples have been sent, their average value is also sent via the same channel. Task Filter reads N values from the channel (and ignores them) and then reads the average value, multiplies the average value by c (c can be read in from port COEF) and writes the result to port OUT. The third parameter in READ and WRITE calls is the number of items to be read or written. READ calls are blocking, WRITE calls are blocking if the number of items in the channel exceed a predefined threshold. The SELECT statement has the same semantics as the statement with the same name in ADA (see page 57): execution of this task is suspended until input arrives from one of the ports. This example meets all criteria for splitting tasks

that were mentioned in the context of fig. 5.4. Both tasks will be waiting for input while occupying resources. Efficiency could be improved by restructuring these tasks. However, the simple splitting of fig. 5.4 is not sufficient. The technique proposed by Cortadella et al. is a more comprehensive one. Using their technique, FlowC-programs are first translated into (extended) Petri-nets. Petri-nets for each of the tasks are then merged into a single Petri-net. Using results from Petri-net theory, new tasks are then generated. Fig. 5.6 shows a possible new task structure.

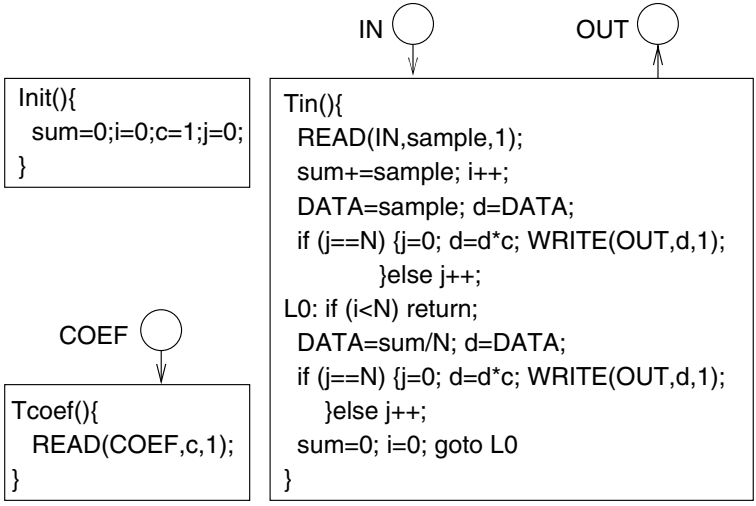

Figure 5.6. Generated software tasks

In this new task structure, there is one task which performs all initializations: In addition, there is one task for each of the input ports. An efficient implementation would raise interrupts each time new input is received for a port. These should be a unique interrupt per port. The tasks could then be started directly by those interrupts, and there would be no need to invoke the operating system for that. Communication can be implemented as a single shared global variable (assuming a shared address space for all tasks). The overall operating system overhead would be quite small, if required at all.

The code for task Tin shown in fig. 5.6 is the one that is generated by the Petri net-based inter-task optimization of the task structure. It should be further optimized by intra-task optimizations, since the test performed for the first if-statement is always false (j is equal to i-1 in this case, and i and j are reset to 0 whenever i becomes equal to N). For the second if-statement, the test is always true, since this point of control is only reached if i is equal to N and i is equal to j whenever label L0 is reached. Also, the number of variables can be reduced. The following is an optimized version Tin:

```
Tin () {
  READ (IN, sample, 1);
  sum += sample;  i++;
  DATA = sample;  d = DATA;
L0:  if (i < N) return;
  DATA = sum/N;  d = DATA;
  d = d*c;  WRITE(OUT,d,1);
  sum = 0;  i = 0;
  return;
}
```

The optimized version of Tin could be generated by a very clever compiler. Unfortunately, hardly any of today's compilers will perform this optimization. Nevertheless, the example shows the type of transformations required for generating "good" task structures. For more details about the task generation, refer to Cortadella et al. [Cortadella et al., 2000].

Optimizations similar to the one just presented are described in the book by Thoen [Thoen and Catthoor, 2000]. A list of IMEC's publications on task concurrency management is available from IMEC's web site [IMEC Desics group, 2003].

5.2 High-level optimizations

There are many high-level optimizations which can potentially improve the efficiency of embedded software.

5.2.1 Floating-point to fixed-point conversion

Floating-point to fixed-point conversion is a commonly used technique. This conversion is motivated by the fact that many signal processing standards (such as MPEG-2 or MPEG-4) are specified in the form of C-programs using floating-point data types. It is left to the designer to find an efficient implementation of these standards.

For many signal processing applications, it is possible to replace floating-point numbers with fixed-point numbers (see page 109). The benefits may be significant. For example, a reduction of the cycle count by 75% and of the energy consumption by 76% has been reported for an MPEG-2 video compression algorithm [Hüls, 2002]. However, some loss of precision is normally incurred. More precisely, there is a tradeoff between the cost of the implementation and

the quality of the algorithm (evaluated e.g. in terms of the so-called signal-to-noise ratio (SNR)). For small word-lengths, the quality may be seriously affected. Consequently, floating-point data types may be replaced by fixed-point data types, but the quality loss has to be analyzed. This replacement was initially performed manually. However, it is a very tedious and error-prone process.

Therefore, researchers have tried to support this replacement with tools. One of the most well-known tools is FRIDGE (fixed-point programming design environment) [Willems et al., 1997], [Keding et al., 1998]. FRIDGE tools have been made available commercially as part of the Synopsys System Studio tool suite [Synopsys, 2005].

In FRIDGE, the design process starts with an algorithm described in C, including floating-point numbers. This algorithm is then converted to an algorithm described in **fixed-C**. Fixed-C extends C by two fixed-point data types, using the type definition features of C++. Fixed-C is a subset of C++ and provides two data types fixed and Fixed. Fixed-point data types can be declared very much like other variables. The following declaration declares a scalar variable, a pointer, and an array to be fixed-point data types.

```
fixed a,*b,c[8]
```

Providing parameters of fixed-point data types can (but does not have to) be delayed until assignment time:

```
a=fixed(5,4,s,wt,*b)
```

This assignment sets the word-length parameter of a to 5 bits, the fractional word-length to 4 bits, sign to present (s), overflow handling to wrap-around (w), and the rounding mode to truncation (t). The parameters for variables that are read in an assignment are determined by the assignment(s) to those variables. The data type Fixed is similar to fixed, except that a consistency check between parameters used in the declaration and those used in the assignment is performed. For every assignment to a variable, parameters (including the word-length) can be different. This parameter information can be added to the original C-program before the application is simulated. Simulation provides value ranges for all assignments. Based on that information, FRIDGE adds parameter information to all assignments. FRIDGE also infers parameter information from the context. For example, the maximum value of additions is considered to be the sum of the arguments. Added parameter information can be either based on simulations or on worst case considerations. Being based on simulations, FRIDGE does not necessarily assume the worst case values that would result from a formal analysis. The resulting C++-program is simulated again to check for the quality loss. The Synopsys version of Fridge uses

SystemC fixed-point data types to express generated data type information. Accordingly, SystemC can be used for simulating fixed-point data types.

An analysis of the tradeoffs between the additional noise introduced and the word-length needed was proposed by Shi and Brodersen [Shi and Brodersen, 2003] and also by Menard et al. [Menard and Sentieys, 2002].

5.2.2 Simple loop transformations

There is a number of loop transformations that can be applied to specifications. The following is a list of standard loop transformations:

- **Loop permutation:** Consider a two-dimensional array. According to the C standard [Kernighan and Ritchie, 1988], two-dimensional arrays are laid out in memory as shown in fig. 5.7. Adjacent index values of the second index are mapped to a contiguous block of locations in memory. This layout is called row-major order [Muchnick, 1997]. Note that the layout for arrays is different for FORTRAN: Adjacent values of the first index are mapped to a contiguous block of locations in memory (column major order). Publications describing optimizations for FORTRAN can therefore be confusing.

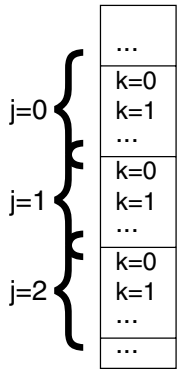

Figure 5.7. Memory layout for two-dimensional array p[j][k] in C

For row-major layout, it is usually beneficial to organize loops such that the last index corresponds to the innermost loop. A corresponding loop permutation is shown in the following example:

```
for (k=0; k<=m; k++)              for (j=0; j<=n; j++)
  for (j=0; j<=n; j++)      ⇒       for (k=0; k<=m; k++)
    p[j][k] = ...                     p[j][k] = ...
```

Such permutations may have a positive effect on the reuse of array elements in the cache, since the next iteration of the loop body will access an adja-

cent location in memory. Caches are normally organized such that adjacent locations can be accessed significantly faster than locations that are further away from the previously accessed location.

- **Loop fusion, loop fission:** There may be cases in which two separate loops can be merged, and there may be cases in which a single loop is split into two. The following is an example:

```
for(j=0; j<=n; j++)                for (j=0; j<=n; j++)
  p[j]= ... ;                        {p[j]= ... ;
for (j=0; j<=n; j++)        ⇔        p[j]= p[j] + ...}
  p[j]= p[j] + ...
```

  The left version may be advantageous if the target processor provides a zero-overhead loop instruction which can only be used for small loops. The right version might lead to an improved cache behavior (due to the improved locality of references to array p), and also increases the potential for parallel computations within the loop body. As with many other transformations, it is difficult to know which of the transformations leads to the best code.

- **Loop unrolling:** Loop unrolling is a standard transformation creating several instances of the loop body. The following is an example in which the loop is being unrolled once:

```
for (j=0; j<=n; j++)               for (j=0; j<=n; j+=2)
  p[j]= ... ;              ⇒         {p[j]= ... ;
                                      p[j+1]= ...}
```

  The number of copies of the loop is called the **unrolling factor**. Unrolling factors larger than two are possible. Unrolling reduces the loop overhead (less branches per execution of the original loop body) and therefore typically improve the speed. As an extreme case, loops can be completely unrolled, removing control overhead and branches altogether. However, unrolling increases code size. Unrolling is normally restricted to loops with a constant number of iterations.

5.2.3 Loop tiling/blocking

It can be observed that the speed of memories is increasing at a slower rate than that of processors. Since small memories are faster than large memories (see page 118), the use of memory hierarchies may be beneficial. Possible "small" memories include caches and scratch-pad memories. A significant reuse factor for the information in those memories is required. Otherwise the memory hierarchy cannot be efficiently exploited.

Reuse effects can be demonstrated by an analysis of the following example. Let us consider matrix multiplication for arrays of size N × N [Lam et al., 1991]:

```
for (i=1; i<=N; i++)
   for(k=1; k<=N; k++){
     r=X[i,k]; /* to be allocated to a register*/
     for (j=1; j<=N; j++)
       Z[i,j] += r* Y[k,j]
   }
```

Let us consider access patterns for this code. The same element X[i,k] is used by all iterations of the innermost loop. Compilers will typically be capable of allocating this element to a register and reuse it for every execution of the innermost loop. We assume that array elements are allocated in row major order (as it is standard for C). This means that array elements with adjacent row (right most) index values are stored in adjacent memory locations. Accordingly, adjacent locations of Z and Y are fetched during the iterations of the innermost loop. This property is beneficial if the memory system uses prefetching (whenever a word is loaded into the cache, loading of the next word is started as well). Fig. 5.8 shows access patterns for this code.

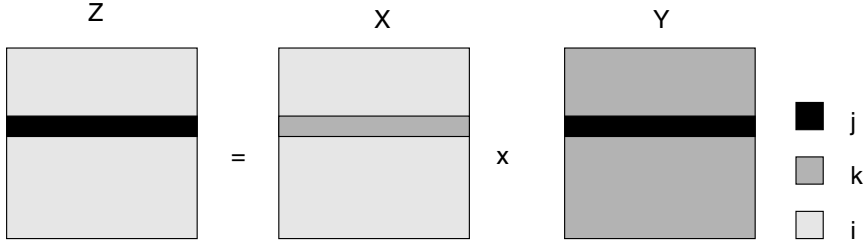

Figure 5.8. Access pattern for unblocked matrix multiplication

For one iteration of the innermost loop, the black areas of arrays Z and Y are accessed (and loaded into the cache). Whether or not the same information is still in the cache for the next iteration of the middle or outermost loops depends on the size of the cache. In the worst case (if N is large or the cache is small), the information has to be reloaded for every execution of the innermost loop and cache elements are not reused. The total number of memory references may be as large as $2\ N^3$ (for references to Z and Y) + N^2 (for references to X).

Research on scientific computing led to the design of **blocked** or **tiled algorithms** [Xue, 2000], which improve the **locality of references**. The following is a tiled version of the above algorithm:

```
for(kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
      }
```

Fig. 5.9 shows the corresponding access pattern.

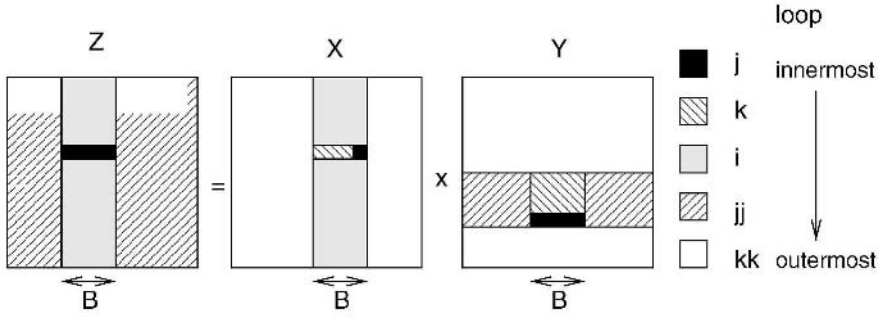

Figure 5.9. Access pattern for tiled/blocked matrix multiplication

The innermost loop is now restricted so that it accesses less array elements (those shown in black). If a proper blocking factor is selected, the elements are still in the cache when the next iteration of the innermost loop starts. The blocking factor B can be chosen such that the elements of the innermost loops fit into the cache. In particular, it can be chosen such that a $B \times B$ sub-matrix of Y fits into the cache. This corresponds to a reuse factor of B for Y, since the elements in the sub-matrix are accessed B times for each iteration of i. Also, a block of B row elements of Z should fit into the cache. These will then be reused during the iterations of k, resulting in a reuse factor of B for Z as well. This reduces the overall number of memory references to at most $2\ N^3/B$ (for references to Z and Y) + N^2 (for references to X). In practice, the reuse factor may be less than B. Optimizing the reuse factor has been an area of comprehensive research. Initial research focused on the performance improvements that can be obtained by tiling. Performance improvements for matrix multiplication by a factor between 3 and 4.3 was reported by Lam [Lam et al., 1991]. Possible improvements are expected to increase with the increasing gap between processor and memory speeds. Tiling can also reduce the energy consumption of memory systems [Chung et al., 2001].

5.2.4 Loop splitting

Next, we discuss loop splitting as another optimization that can be applied before compiling the program. Potentially, this optimization could also be added to compilers.

Many image processing algorithms perform some kind of filtering. This filtering consists of considering the information about a certain pixel as well as that of some of its neighbors. Corresponding computations are typically quite regular. However, if the considered pixel is close to the boundary of the image, not all neighboring pixels exist and the computations have to be modified. In a straightforward description of the filtering algorithm, these modifications may result in tests being performed in the innermost loop of the algorithm. A more efficient version of the algorithm can be generated by splitting the loops such that one loop body handles the regular cases and a second loop body handles the exceptions. Figure 5.10 is a graphical representation of this transformation.

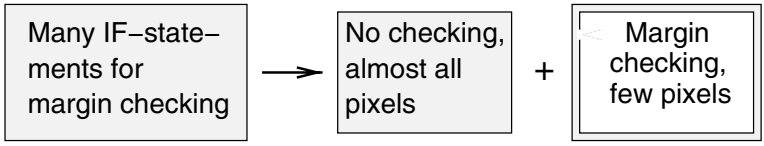

Figure 5.10. Splitting image processing into regular and special cases

Performing this loop splitting manually is a very difficult and error-prone procedure. Falk et al. have published an algorithm [Falk and Marwedel, 2003] to perform a procedure which also works for larger dimensions automatically. It is based on a sophisticated analysis of accesses to array elements in loops. Optimized solutions are generated using genetic algorithms. The following code shows a loop nest from the MPEG-4 standard performing motion estimation:

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
   for (y=0; y<49; y++) {y1=4*y;
    for (k=0; k<9; k++) {x2=x1+k-4;
     for (l=0; l<9; ) {y2=y1+l-4;
      for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
       for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
        if (x3<0 || 35<x3||y3<0||48<y3)
         then_block_1; else else_block_1;
        if (x4<0|| 35<x4||y4<0||48<y4)
```

```
            then_block_2; else else_block_2;
    }}}}}}
```

Using Falk's algorithm, this loop nest is transformed into the following one:

```
    for (z=0; z<20; z++)
      for (x=0; x<36; x++) {x1=4*x;
       for (y=0; y<49; y++)
        if (x>=10||y>=14)
          for (; y<49; y++)
            for (k=0; k<9; k++)
              for (l=0; l<9;l++ )
                for (i=0; i<4; i++)
                  for (j=0; j<4;j++) {
                   then_block_1; then_block_2}
        else {y1=4*y;
          for (k=0; k<9; k++) {x2=x1+k-4;
            for (l=0; l<9; ) {y2=y1+l-4;
              for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
              for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
               if (0 || 35<x3 ||0|| 48<y3)
                then_block_1; else else_block_1;
               if (x4<0|| 35<x4||y4<0||48<y4)
                then_block_2; else else_block_2;
    }}}}}}
```

Instead of complicated tests in the innermost loop, we now have a splitting if-statement after the third for-loop statement. All regular cases are handled in the then-part of this statement. The else-part handles the relatively small number of remaining cases.

Fig. 5.11 shows the number of cycles that can be saved by loop nest splitting for various applications and target processors.

For the motion estimation algorithm, cycle counts can be reduced by up to about 75 % (to 25 % of the original value). Obviously, substantial savings are possible. This potential should certainly not be ignored.

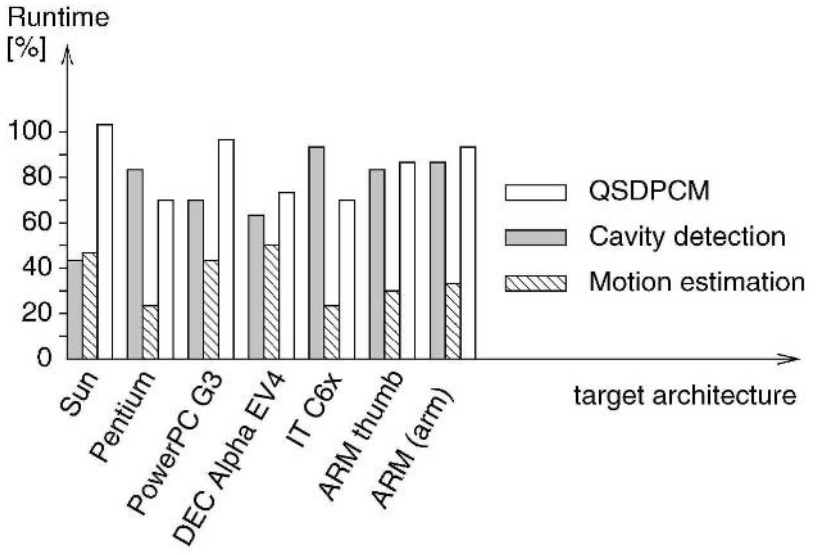

Figure 5.11. Results for loop splitting

5.2.5 Array folding

Some embedded applications, especially in the multimedia domain, include large arrays. Since memory space in embedded systems is limited, options for reducing the storage requirements of arrays should be explored. Fig. 5.12 represents the addresses used by five arrays as a function of time. At any particular time only a subset of array elements is needed. The maximum number of elements needed is called the **address reference window** [De Greef et al., 1997a]. In fig. 5.12, this maximum is indicated by a double-headed arrow.

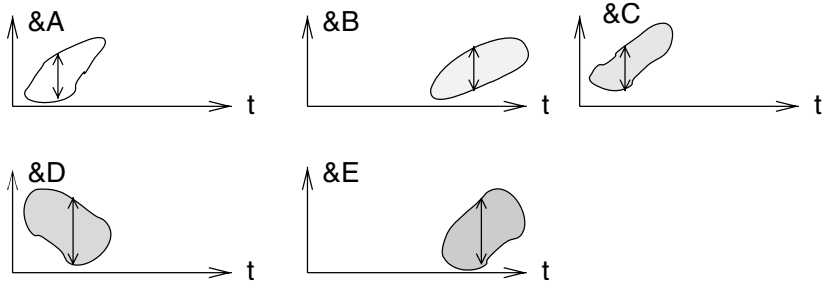

Figure 5.12. Reference patterns for arrays

A classical memory allocation for arrays is shown in fig. 5.13 (left). Each array is allocated the maximum of the space it requires during the entire execution time (if we consider global arrays).

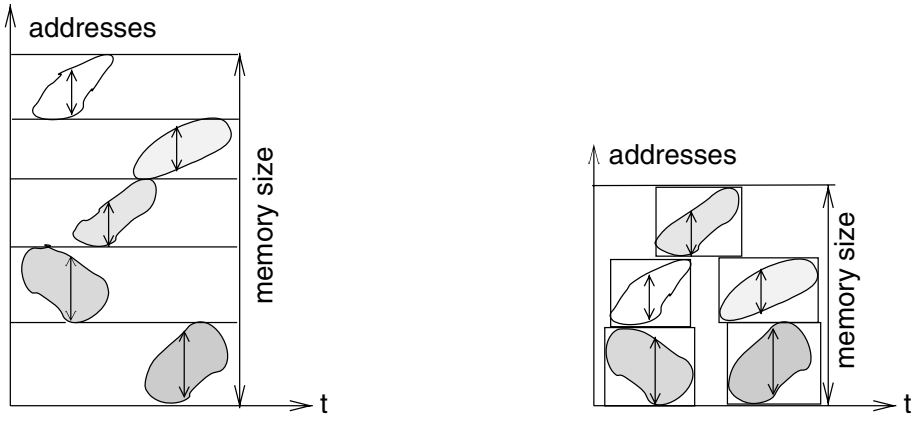

Figure 5.13. Unfolded (left) and inter-array folded (right) arrays

One of the possible improvements, **inter-array folding**, is shown in fig. 5.13 (right). Arrays which are not needed at overlapping time intervals can share the same memory space. A second improvement, intra-array folding [De Greef et al., 1997b], is shown in fig. 5.14. It takes advantage of the limited sets of components needed **within** an array. Storage can be saved at the expense of more complex address computations.

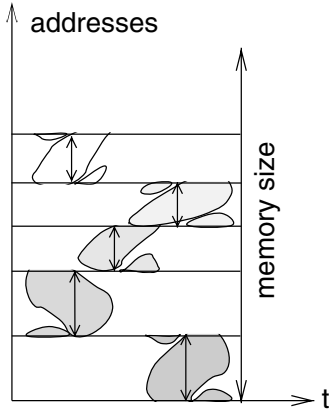

Figure 5.14. Intra-array folded arrays

The two kinds of foldings can also be combined.

Other forms of high-level transformations have been analyzed by Chung, Benini and De Micheli [Chung et al., 2001], [Tan et al., 2003]. There are many additional contributions in this domain in the compiler community.

In particular, function inlining[1] replaces function calls by the code of the called function. This transformation improves the speed of the code, but results in an increase in the code size. Increased code sizes may be a problem in SoC technologies. Traditional in-lining techniques rely on the user identifying functions to be inlined. This is a problem in systems on a chip, since the size of the instruction memory is very critical for such systems. Hence, it is important to be able to constrain the size of the instruction memory and to let design tools find out automatically which of the functions should be in-lined for a certain size of the memory. Known approaches for this include techniques by Teich [Teich et al., 1999], Leupers et al. [Leupers and Marwedel, 1999], and [Palkovic et al., 2002]. These techniques can be either integrated into a compiler or can be applied as a source-to-source transformation before using any compiler.

5.3 Hardware/software partitioning

5.3.1 Introduction

During the design process, we have to solve the problem of implementing the specification either in hardware or in the form of programs running on processors. This section describes some of the techniques for this mapping. Applying these techniques, we will be able to decide which parts have to be implemented in hardware and which in software.

By hardware/software partitioning we mean the mapping of task graph nodes to either hardware or software. A standard procedure for embedding hardware/software partitioning into the overall design flow is shown in fig. 5.15. We start from a common representation of the specification, e.g. in the form of task graphs and information about the platform.

For each of the nodes of the task graphs, we need information concerning the effort required and the benefits received from choosing a certain implementation of these nodes. For example, execution times must be predicted (see page 127). It is very hard to predict times required for communication. Nevertheless, two tasks requiring a very high communication bandwidth should preferably be mapped to the same components. Iterative approaches are used in many cases. An initial solution to the partitioning problem is generated, analyzed and then improved.

[1] The concept of inlining is assumed to be known to the reader from programming courses.

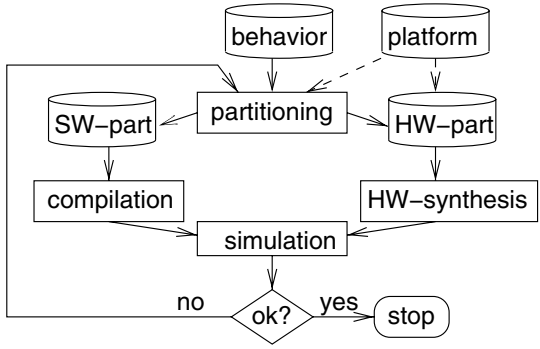

Figure 5.15. General view of hardware/software partitioning

Some approaches for partitioning are restricted to mapping task graph nodes either to special purpose hardware or to software running on a single processor. Such partitioning can be performed with bipartitioning algorithms for graphs [Kuchcinski, 2002].

More elaborate partitioning algorithms are capable of mapping graph nodes to multi-processor systems and hardware. In the following, we will describe how this can be done using a standard optimization technique from operations research, **integer programming**. Our presentation is based on a simplified version of the optimization proposed for the codesign tool COOL [Niemann, 1998].

5.3.2 COOL

For COOL, the input consists of three parts:

- **Target technology:** This part of the input to COOL comprises information about the available hardware platform components. COOL supports multiprocessor systems, but requires that all processors are of the same type, since it does not include automatic or manual processor selection. The name of the processor used (as well as information about the corresponding compiler) must be included in this part of the input to COOL. As far as the application-specific hardware is concerned, the information must be sufficient for starting automatic hardware synthesis with all required parameters. In particular, information about the technology library must be given.
- **Design constraints:** The second part of the input comprises design constraints such as the required throughput, latency, maximum memory size, or maximum area for application-specific hardware.

- **Behavior:** The third part of the input describes the required overall behavior. Hierarchical task graphs are used for this. We can think of, e.g. using the hierarchical task graph of fig. 2.46 for this.

  COOL uses two kinds of edges: communication edges and timing edges. Communication edges may contain information about the amount of information to be exchanged. Timing edges provide timing constraints. COOL requires the behavior of each of the leaf nodes[2] of the graph hierarchy to be known. COOL expects this behavior to be specified in VHDL[3].

For partitioning, COOL uses the following steps:

1 **Translation** of the behavior **into an internal graph model**.

2 **Translation** of the behavior of each node **from VHDL into C**.

3 **Compilation of all C programs** for the selected target processor, computation of the resulting program size, estimation of the resulting execution time. If simulations are used for the latter, simulation input data must be available.

4 **Synthesis of hardware components:** For each leaf node, application-specific hardware is synthesized. Since quite a number of hardware components may have to be synthesized, hardware synthesis should not be too slow. It was found that commercial synthesis tools focusing on gate level synthesis can be too slow to be useful for COOL. However, high-level synthesis tools working at the register-transfer-level (using adders, registers, and multiplexer as components, rather than gates) provide sufficient synthesis speed. Also, such tools can provide sufficiently precise values for delay times and required silicon area. In the actual implementation, the OSCAR high-level synthesis tool [Landwehr and Marwedel, 1997] is used.

5 **Flattening the hierarchy:** The next step is to extract a flat task graph from the hierarchical flow graph. Since no merging or splitting of nodes is performed, the granularity used by the designer is maintained. Cost and performance information gained from compilation and from hardware synthesis are added to the nodes. This is actually one of the key ideas of COOL: the **information required for hardware/software partitioning is precomputed and it is computed with good precision**. This information forms the basis for generating cost-minimized designs meeting the design constraints.

[2] See page 20 for a definition of this term.

[3] In retrospect, we now know that C should have been used for this, as this choice would have made the partitioning for many standards described in C easier.

6 **Generating and solving a mathematical model of the optimization problem:** COOL uses integer programming (IP) to solve the optimization problem. A commercial IP solver is used to find values for decision variables minimizing the cost. The solution is optimal with respect to the cost function derived from the available information. However, this cost includes only a coarse approximation of the communication time. The communication time between any two nodes of the task graph depends on the mapping of those nodes to processors and hardware. If both nodes are mapped to the same processor, communication will be local and thus quite fast. If the nodes are mapped to different hardware components, communication will be non-local and may be slower. Modeling communication costs for all possible mappings of task graph nodes would make the model very complex and is therefore replaced by iterative improvements of the initial solution. More details on this step will be presented below.

7 **Iterative improvements:** In order to work with good estimates of the communication time, adjacent nodes mapped to the same hardware component are now merged. This merging is shown in fig. 5.16.

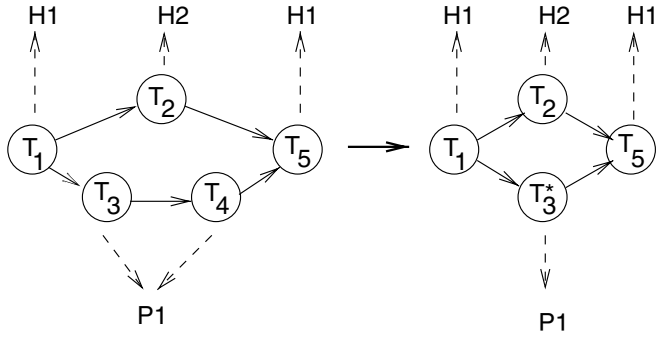

Figure 5.16. Merging of task nodes mapped to the same hardware component

We assume that tasks T_1, T_2 and T_5 are mapped to hardware components H1 and H2, whereas T_3 and T_4 are mapped to processor P1. Accordingly, communication between T_3 and T_4 is **local** communication. Therefore, we merge T_3 and T_4, and assume that the communication between the two tasks does not require a communication channel. Communication time can be now estimated with improved precision. The resulting graph is then used as new input for mathematical optimization . The previous and the current step are repeated until no more graph nodes are merged.

8 **Interface synthesis:** After partitioning, the glue logic required for interfacing processors, application-specific hardware and memories is created.

Next, we will describe step 6 in more detail. IP models provide a general approach for modeling optimization problems. IP models consist of two parts: a cost function and a set of constraints. Both parts involve references to a set $X = \{x_i\}$ of integer-valued variables. Cost functions must be linear functions of those variables. So, they must be of the general form

$$C = \sum_{x_i \in X} a_i x_i, \text{ with } a_i \in \mathbb{R}, x_i \in \mathbb{Z} \tag{5.1}$$

The set J of constraints must also consist of linear functions of integer-valued variables. They have to be of the form

$$\forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \tag{5.2}$$

Note that $\geq$ can be replaced by $\leq$ in equation (5.2) if constants $b_{i,j}$ are modified accordingly.

Def.: The **integer programming (IP-) problem** is the problem of minimizing cost function (5.1) subject to the constraints given in eq. 5.2. If all variables are constrained to being either 0 or 1, the corresponding model is called a **0/1-integer programming model**. In this case, variables are also denoted as **(binary) decision variables**.

For example, assuming that x_1, x_2 and x_3 cannot be negative and must be integers, the following set of equations represent a 0/1-IP model:

$$C = 5x_1 + 6x_2 + 4x_3 \tag{5.3}$$
$$x_1 + x_2 + x_3 \geq 2 \tag{5.4}$$
$$x_1 \leq 1 \tag{5.5}$$
$$x_2 \leq 1 \tag{5.6}$$
$$x_3 \leq 1 \tag{5.7}$$

Due to the constraints, all variables are either 0 or 1. There are four possible solutions. These are listed in table 5.1. The solution with a cost of 9 is optimal.

Applications requiring **maximizing** some **gain** function C' can be changed into the above form by setting $C = -C'$.

IP models can be solved optimally using mathematical programming techniques. Unfortunately, integer programming is NP-complete and execution

x_1	x_2	x_3	C
0	1	1	10
1	0	1	9
1	1	0	11
1	1	1	15

Table 5.1. Possible solutions of the presented IP-problem

times may become very large. Nevertheless, it is useful for solving optimization problems as long as the model sizes are not extremely large. Execution times depend on the number of variables and on the number and structure of the constraints. Good IP solvers (like lp_solve [Berkelaar and et al., 2005] or CPLEX) can solve well-structured problems containing a few thousand variables in acceptable computation times (e.g. minutes). For more information on integer programming and related **linear programming**, refer to books on the topic (e.g. to Wolsey [Wolsey, 1998]). Modeling optimization problems as integer programming problems makes sense despite the complexity of the problem: many problems can be solved in acceptable execution times and if they cannot, IP models provide a good starting point for heuristics.

Next, we will describe how partitioning can be modeled using a 0/1-IP model. The following index sets will be used in the description of the IP model:

- Index set I denotes task graph nodes. Each $i \in I$ corresponds to one task graph node.
- Index set L denotes task graph node **types**. Each $l \in L$ corresponds to one task graph node type. For example, there may be nodes describing square root, Discrete Cosine Transform (DCT) or Discrete Fast Fourier Transform (DFT) computations. Each of them is counted as one type.
- Index set KH denotes hardware component **types**. Each $k \in KH$ corresponds to one hardware component type. For example, there may be special hardware components for the DCT or the DFT. There is one index value for the DCT hardware component and one for the FFT hardware component.
- For each of the hardware components, there may be multiple copies, or "instances". Each instance is identified by an index $j \in J$.
- Index set KP denotes processors. Each $k' \in KP$ identifies one of the processors (all of which are of the same type).

The following decision variables are required by the model:

- $X_{i,k}$: this variable will be 1, if node v_i is mapped to hardware component type $k \in KH$ and 0 otherwise.
- $Y_{i,k}$: this variable will be 1, if node v_i is mapped to processor $k \in KP$ and 0 otherwise.
- $NY_{l,k}$: this variable will be 1, if at least one node of type l is mapped to processor $k \in KP$ and 0 otherwise.
- T is a mapping $I \rightarrow L$ from task graph nodes to their corresponding types.

In our particular case, the cost function accumulates the total cost of all hardware units:

$$C = \text{processor costs + memory costs + cost of application specific hardware}$$

We would obviously minimize the total cost if no processors, memory and application specific hardware were included in the "design". Due to the constraints, this is not a legal solution. We can now present a brief description of some of the constraints of the IP model:

- **Operation assignment constraints:** These constraints guarantee that each operation is implemented either in hardware or in software. The corresponding constraints can be formulated as follows:

$$\forall i \in I \quad : \quad \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1$$

  In plain text, this means the following: for all task graph nodes i, the following must hold: i is implemented either in hardware (setting one of the $X_{i,k}$ variables to 1, for some k) or it is implemented in software (setting one of the $Y_{i,k}$ variables to 1, for some k).

  All variables are assumed to be non-negative integer numbers:

$$X_{i,k} \in I\!N_0, \tag{5.8}$$
$$Y_{i,k} \in I\!N_0 \tag{5.9}$$

  Additional constraints ensure that decision variables $X_{i,k}$ and $Y_{i,k}$ have 1 as an upper bound and, hence, are in fact 0/1-valued variables:

$$\forall i \in I : \forall k \in KH \quad : \quad X_{i,k} \leq 1$$
$$\forall i \in I : \forall k \in KP \quad : \quad Y_{i,k} \leq 1$$

If the functionality of a certain node of type l is mapped to some processor k, then this processors' instruction memory must include a copy of the software for this function:

$$\forall l \in L, \forall i : T(v_i) = c_l, \forall k \in KP : \qquad NY_{l,k} \geq Y_{i,k}$$

In plain text, this means: for all types l of task graph nodes and for all nodes i of this type, the following must hold: if i is mapped to some processor k (indicated by $Y_{i,k}$ being 1), then the software corresponding to functionality l must be provided by processor k, and the corresponding software must exist on that processor (indicated by $NY_{l,k}$ being 1).

Additional constraints ensure that decision variables $NY_{l,k}$ are also 0/1-valued variables:

$$\forall l \in L : \forall k \in KP : \qquad NY_{l,k} \leq 1$$

- **Resource constraints:** The next set of constraints ensures that "not too many" nodes are mapped to the same hardware component at the same time. We assume that, for every clock cycle, at most one operation can be performed per hardware component. Unfortunately, this means that the partitioning algorithm also has to generate a schedule for executing task graph nodes. Scheduling by itself is already an NP-complete problem for most of the relevant problem instances.
- **Precedence constraints:** These constraints ensure that the schedule for executing operations is consistent with the precedence constraints in the task graph.
- **Design constraints:** These constraints put a limit on the cost of certain hardware components, such as memories, processors or area of application-specific hardware.
- **Timing constraints:** Timing constraints, if present in the input to COOL, are converted into IP constraints.
- Some additional, but less important constraints are not included in this list.

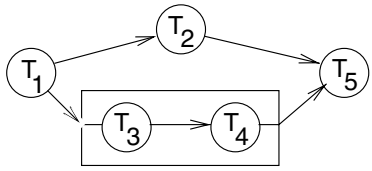

Figure 5.17. Task graph

Example: In the following, we will show how these constraints can be generated for the task graph in fig. 5.17 (the same as the one in fig. 2.46).

Suppose that we have a hardware component library containing three components types H1, H2 and H3 with costs of 20, 25 and 30 cost units, respectively. Furthermore, suppose that we can also use a processor P of cost 5. In addition, we assume that table 5.2 describes the execution times of our tasks on these components.

T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Table 5.2. Execution times of tasks T_1 to T_5 on components

Tasks T_1 to T_5 can only be executed on the processor or on one application-specific hardware unit. Obviously, processors are assumed to be cheap but slow in executing tasks T_1, T_2, and T_5.

The following operation assignment constraints must be generated, assuming that a maximum of one processor (P1) is to be used:

$$X_{1,1} + Y_{1,1} = 1 \text{ (Task 1 either mapped to H1 or to P1)}$$
$$X_{2,2} + Y_{2,1} = 1 \text{ (Task 2 either mapped to H2 or to P1)}$$
$$X_{3,3} + Y_{3,1} = 1 \text{ (Task 3 either mapped to H3 or to P1)}$$
$$X_{4,3} + Y_{4,1} = 1 \text{ (Task 4 either mapped to H3 or to P1)}$$
$$X_{5,1} + Y_{5,1} = 1 \text{ (Task 5 either mapped to H1 or to P1)}$$

Furthermore, assume that the types of tasks T_1 to T_5 are $l = 1,2,3,3$ and 1, respectively. Then, the following additional resource constraints are required:

$$NY_{1,1} \geq Y_{1,1} \tag{5.10}$$
$$NY_{2,1} \geq Y_{2,1}$$
$$NY_{3,1} \geq Y_{3,1}$$
$$NY_{3,1} \geq Y_{4,1}$$
$$NY_{1,1} \geq Y_{5,1} \tag{5.11}$$

Equation 5.10 means: if task 1 is mapped to the processor, then the function $l = 1$ must be implemented on that processor. The same function must also be implemented on the processor if task 5 is mapped to the processor (eq. 5.11).

We have not included timing constraints. However, it is obvious that the processor is slow in executing some of the tasks and that application-specific hardware is required for timing constraints below 100 time units.

The cost function is:

$$C = 20 * \#(\mathsf{H1}) + 25 * \#(\mathsf{H2}) + 30 * \#(\mathsf{H3}) + 5 * \#(\mathsf{P})$$

where #() denotes the number of instances of hardware components. This number can be computed from the variables introduced so far if the schedule is also taken into account. For a timing constraint of 100 time units, the minimum cost design comprises components H1, H2 and P. This means that tasks T_3 and T_4 are implemented in software and all others in hardware.

In general, due to the complexity of the combined partitioning and scheduling problem, only small problem instances of the combined problem can be solved in acceptable run-times. Therefore, the problem is heuristically split into the scheduling and the partitioning problem: an initial partitioning is based on estimated execution times and the final scheduling is done after partitioning. If it turns out that the schedule was too optimistic, the whole process has to be repeated with tighter timing constraints. Experiments for small examples have shown that the cost for heuristic solutions is only 1 or 2 % larger than the cost of optimal results.

Automatic partitioning can be used for analyzing the design space. In the following, we will present results for an audio lab, including mixer, fader, echo, equalizer and balance units. This example uses earlier target technologies in order to demonstrate the effect of partitioning. The target hardware consists of a (slow) SPARC processor, external memory, and application-specific hardware to be designed from an (outdated) 1μ ASIC library. The total allowable

delay is set to 22675 ns, corresponding to a sample rate of 44.1 kHz, as used in CDs. Fig. 5.18 shows different design points which can be generated by changing the delay constraint.

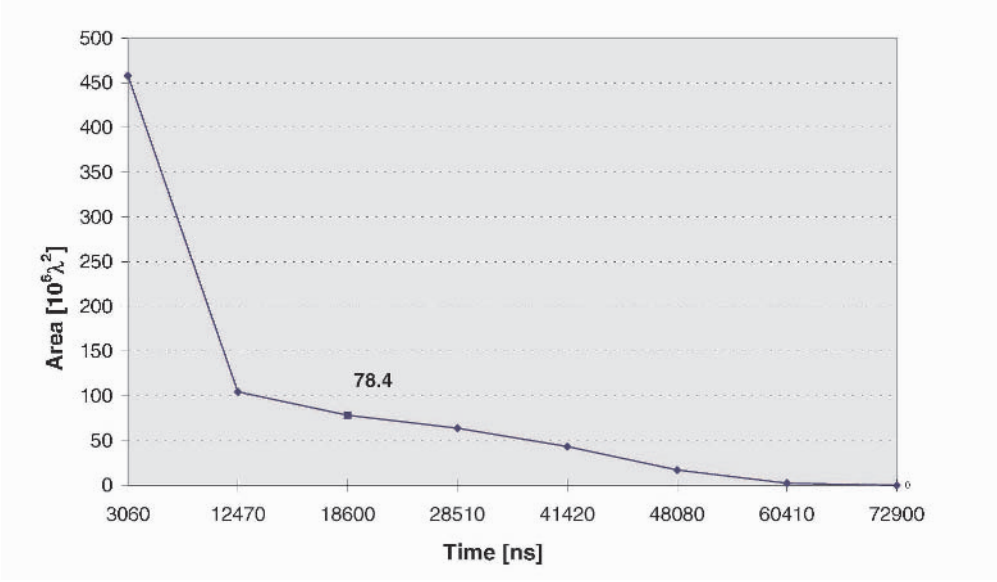

Figure 5.18. Design space for audio lab

The unit λ refers to a technology-dependent length unit. It is essentially one half of the closest distance between the centers of two metal wires on the chip (also called *half-pitch* [SEMATECH, 2003]). The design point at the left corresponds to a solution implemented completely in hardware, the design point at the right to a software solution. Other design points use a mixture of hardware and software. The one corresponding to an area of 78.4 λ^2 is the cheapest meeting the deadline.

Obviously, technology has advanced to allow a 100% software-based audio lab design nowadays. Nevertheless, this example demonstrates the underlying design methodology which can also be used for more demanding applications, especially in the high-speed multimedia domain, such as MPEG-4.

5.4 Compilers for embedded systems

5.4.1 Introduction

Obviously, optimizations and compilers are available for the processors used in PCs and compiler generation for commonly used 32-bit processors is well

understood. For embedded systems, standard compilers are also used in many cases, since they are typically cheap or even freely available.

However, there are several reasons for designing special optimizations and compilers for embedded systems:

- Processor architectures in embedded systems exhibit special features (see page 100). These features should be exploited by compilers in order to generate efficient code.
- High levels of optimization are more important than high compilation speed.
- Compilers could potentially help to meet and prove real-time constraints. For example, it may be beneficial to freeze certain cache lines in order to prevent frequently executed code from being evicted and reloaded several times.
- Compilers may help to reduce the energy consumption of embedded systems. Compilers performing energy optimizations should be available.
- For embedded systems, there is a larger variety of instruction sets. Hence, there are more processors for which compilers should be available. Sometimes there is even the request to support the optimization of instruction sets with **retargetable** compilers. For such compilers, the instruction set can be specified as an input to a compiler generation system. Such systems can be used for experimentally modifying instruction sets and then observing the resulting changes for the generated machine code. This is one particular case of **design space exploration** and is supported, for example, by Tensilica tools [Tensilica Inc., 2003].

Some first approaches for retargetable compilers are described in the first book on this topic [Marwedel and Goossens, 1995]. Optimizations can be found in more recent books by Leupers [Leupers, 1997], [Leupers, 2000a]. In this section, we will present examples of compilation techniques for embedded processors.

Compilation techniques might also have to support compression techniques described on pages 103 to 105.

5.4.2 Energy-aware compilation

Many embedded systems are mobile systems which have to run on batteries. While computational demands on mobile systems are increasing, battery technology is expected to improve only slowly [SEMATECH, 2003]. Hence, the availability of energy is a serious bottleneck for new applications.

Saving energy can be done at various levels, including the fabrication process technology, the device technology, circuit design, the operating system and the application algorithms. Adequate translation from algorithms to machine code can also help. High-level optimization techniques such as those presented on pages 157 to 167 can also help to reduce the energy consumption. In this section, we will look at compiler optimizations which can reduce the energy consumption. **Power models** are very essential ingredients of all power optimizations. A general problem of power models is their frequently very limited precision[4].

- One of the first power models was proposed by Tiwari [Tiwari et al., 1994]. The model includes so-called base costs and inter-instruction costs. Base costs of an instruction correspond to the energy consumed per instruction execution if an infinite sequence of that instruction is executed. Inter-instruction costs model the additional energy consumed by the processor if instructions change. This additional energy is required, for example, due to switching functional units on and off. This power model focuses on the consumption in the processor and does not consider the power consumed in the memory or in other parts of the system.
- Another power model was proposed by Simunic et al. [Simunic et al., 1999]. That model is based on data sheets. The advantage of this approach is that the contribution of all components of an embedded system to the energy consumption can be computed. However, the information in data sheets about average values may be less precise than the information about maximal or minimal values.
- A third model has been proposed by Rusell and Jacome [Russell and Jacome, 1998]. This model is based on precise measurements of two fixed configurations.
- Still another model was proposed by Lee [Lee et al., 2001]. This model includes an detailed analysis of the effects of the pipeline. It does not include multicycle operations and pipeline stalls.
- The encc energy-aware compiler from Dortmund University uses the energy model by Steinke et al. [Steinke et al., 2001]. It is based on precise measurements using real hardware. The consumption of the processor as well as that of the memory in included.
- The energy consumption of caches can be computed with CACTI [Wilton and Jouppi, 1996].

[4]Deviations of about 50% are frequently mentioned in discussions.

Using models like the one above, the following compiler optimizations have been used for reducing the energy consumption:

- **Energy-aware scheduling:** the order of instructions can be changed as long as the meaning of the program does not change. The order can be changed such that the number of transitions on the instruction bus is minimized. This optimization can be performed on the output generated by a compiler and therefore does not require any change to the compiler.
- **Energy-aware instruction selection:** typically, there are different instruction sequences for implementing the same source code. In a standard compiler, the number of instructions or the number of cycles is used as a criterion (cost function) for selecting a good sequence. This criterion can be replaced by the energy consumed by that sequence. Steinke and others found that low-power instruction selection reduces the energy consumption by some percent.
- **Replacing the cost function** is also possible for other standard compiler optimizations, such as register pipelining, loop invariant code motion etc. Possible improvements are also in the order of a few percent.
- **Exploitation of the memory hierarchy:** As explained on page 118, smaller memories provide faster access and consume less energy per access. Therefore, a significant amount of energy can be saved if the existence of small scratch pad memories (SPMs) can be exploited by a compiler. For this purpose, each basic block and each variable can be modeled as a memory segment i. For each segment, there is a corresponding size s_i. Using profiling, it is possible to compute the gain g_i of moving segment i to the scratch pad memory. Let

$$x_i = \begin{cases} 1 & \text{if segment } i \text{ is mapped to the SPM} \\ 0 & \text{otherwise} \end{cases} \tag{5.12}$$

Then, the goal is to maximize

$$\sum_i g_i \cdot x_i \tag{5.13}$$

while respecting the size constraint

$$\sum_i s_i \cdot x_i \leq K \tag{5.14}$$

where K is the size of the SPM.

This problem is known as a knapsack problem. The solution of this problem is a one-to-one mapping. An integer programming model leading to such a mapping was presented by Steinke et al. [Steinke et al., 2002b]. For some benchmark applications, energy reductions of up to about 80% were found, even though the size of the SPM was just a small fraction of the total code size of the application. Results for the bubble sort program are shown in fig. 5.19.

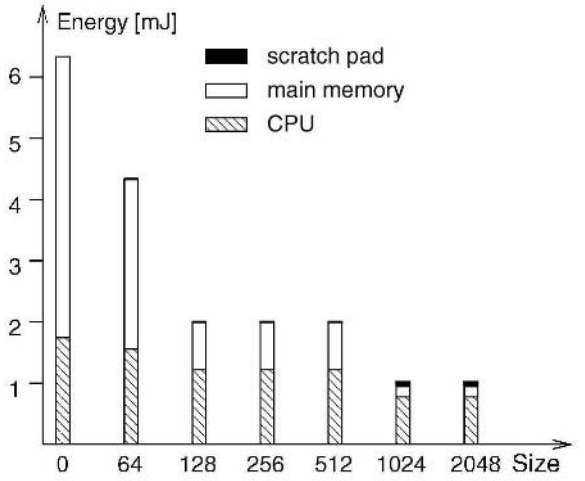

Figure 5.19. Energy reduction by compiler-based mapping to scratch-pad for bubble sort

Obviously, larger SPMs lead to a reduced energy consumption in the main memory. The energy required in the processor is also reduced, since less wait cycles are required. Supply voltages have been assumed to be constant.

Code can also be dynamically copied into the SPM, resulting in a many-to-one mapping. An integer programming model reflecting this more general optimization problem was also proposed by Steinke et al. [Steinke et al., 2002a]. Using this more general model, the energy gain can be increased, especially for applications for which the SPM is too small to contain all hot spots.

Of all the compiler optimizations analyzed by Steinke, the energy savings enabled by memory hierarchies are the largest.

5.4.3 Compilation for digital signal processors

Features of DSP processors are described on page 108. Compilers should exploit these. Techniques for this can be demonstrated using address generation

units as examples. This possibility of generating addresses "for free" has an important impact on how variables should be laid out in memory. Fig. 5.20 shows an example.

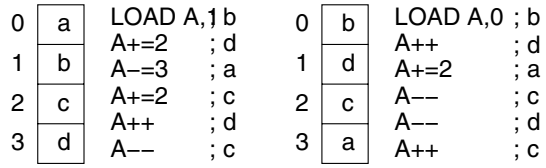

Figure 5.20. Comparison of memory layouts

We assume that in some basic block, variables a to d are accessed in the sequence (b,d,a,c,d,c). Accessing these variables with register-indirect addressing requires, first of all, loading the address of b into an address register (see fig. 5.20, left). The instruction referring to variable b is not shown in fig. 5.20, since the current focus is on address generation. Therefore, the generation of the address for the access to the next variable (d) is considered next. Assuming that there is just a single address register A, A has to be updated to point to variable d. This requires adding 2 to the register. Again, we ignore the instruction loading the variable, and we immediately consider the access to a. For this, we have to subtract 3, and for the next access we have to add 2. Assuming that the auto-increment and -decrement range is restricted to $\pm$ 1, only the last two accesses shown in fig. 5.20 can be implemented with these operations. In total, 4 instructions for calculating addresses are needed.

In contrast, for the layout in fig. 5.20 (right), 4 address calculations are auto-increment and -decrement operations which will be executed in parallel with some operation in the main data path. Only 2 cycles are needed for address calculations with an offset larger than 1. Again, the instructions actually using the variables are not shown.

How do we generate such clever memory layouts? Algorithms doing this typically start from an access graph (see fig. 5.21).

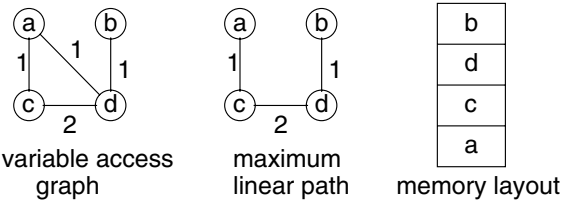

Figure 5.21. Memory allocation for access sequence (b, d, a, c, d, c) for a single address register A

Such access graphs have one node for each of the variables and have an edge for every pair of variables for which there are adjacent accesses. The weight of such edges corresponds to the number of adjacent accesses to the variables connected by that edge.

Variables connected by an edge of a high weight should preferably be allocated to adjacent memory locations. The number of address calculations saved in this way is equal to the weight of the corresponding edge. For example, if `c` and `d` are allocated to adjacent locations, then the last two accesses in the sequence can be implemented with auto-increment and -decrement operations.

The overall goal of memory allocation is to find a linear order of variables in memory maximizing the use of auto-increment and -decrement operations. This corresponds to finding a linear path of maximum weight in the variable access graph. Unfortunately, the maximum weighted path problem in graphs is NP-complete. Hence, it is common to use heuristics for generating such paths [Liao et al., 1995b], [Sudarsanam et al., 1997]. Most of them are based on Kruskal's spanning tree heuristic. They start with a graph with no edges and then incrementally add edges with decreasing weight, always keeping the degree of all nodes to at most 2 and avoiding cycles. The order of the variables in memory will then correspond to the order of the variables along the linear path.

The algorithm just sketched only covers a simple case. Extensions of this algorithm cover more complex situations, such as:

- $n > 1$ address registers [Leupers and Marwedel, 1996],
- also using modify registers present in the AGU [Leupers and Marwedel, 1996], [Leupers and David, 1998],
- extension to arrays [Basu et al., 1999],
- larger auto-increment and -decrement ranges [Sudarsanam et al., 1997].

Memory allocation, as described above, improves both the code-size and the run-time of the generated code. Other proposed optimization algorithms exploit further architectural features of DSP processors, such as:

- multiple memory banks [Sudarsanam and Malik, 1995],
- heterogeneous register files [Araujo and Malik, 1995],
- modulo addressing,
- instruction level parallelism [Leupers and Marwedel, 1995],

- multiple operation modes [Liao et al., 1995a].

Other, new optimization techniques are described by Leupers [Leupers, 2000a].

5.4.4 Compilation for multimedia processors

In order to fully support packed data types as described on page 110, compilers must be able to automatically convert operations in loops to operations on packed data types. Taking advantage of this potential is necessary for generating efficient software. A very challenging task is to use this feature in compilers. Compiler algorithms exploiting operations on packed data types are extensions of vectorizing algorithms originally developed for supercomputers, but only some algorithms have been described so far [Fisher and Dietz, 1998], [Fisher and Dietz, 1999], [Leupers, 2000b], [Krall, 2000], [Larsen and Amarasinghe, 2000].

Automatic parallelization of loops for the M3-DSP (see page 113) requires the use of vectorization techniques, which achieve significant speedups (compared to the case of sequential operations, see fig. 5.22) [Lorenz et al., 2002]. For application dot_product_2, the size of the vectors was too small to lead to a speedup and no vectorization should be performed. The number of cycles can be reduced by 94 % for benchmark example if vectorization is combined with an exploitation of zero-overhead-loop instructions.

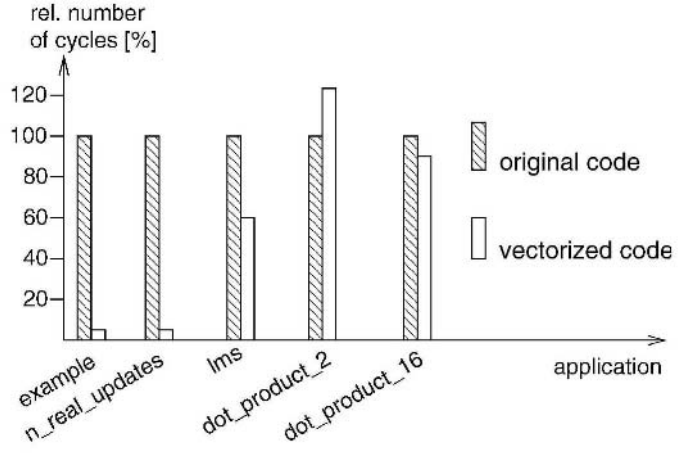

Figure 5.22. Reduction of the cycle count by vectorization for the M3-DSP

5.4.5 Compilation for VLIW processors

VLIW architectures (see page 111) require special compiler optimizations:

- A key optimization required for TMS 320C6xx compilers is to allocate, at compile time, the functional unit that should execute a certain operation. Due to the two data paths (see fig. 3.21), this implies a partitioning of the operations into two sets [Jacome and de Veciana, 1999], [Jacome et al., 2000], [Leupers, 2000c] and also includes an allocation to one of the register files.

- VLIW processors frequently have branch delay slots. For VLIW processors, the branch delay penalty is significantly larger than for other processors, because each of the branch delay slots could hold a full instruction packet, not just a single instruction. For example, for the TMS 320C6xx, the branch delay penalty is $5 \times 8 = 40$ instructions. In order to avoid this large penalty, most VLIW processors support predicated execution for a large number of condition code registers. Predicated execution can be employed to efficiently implement small if-statements. For large if-statements, however, conditional branches are more efficient, since these allow mutual exclusion of then- and else-branches to be exploited in hardware allocation. The precise tradeoff between the two methods for implementing if-statements can be found with proper optimization techniques [Mahlke et al., 1992], [August et al., 1997], [Leupers, 1999].

- Due to the large branch delay penalty, inlining (see page 167) is another optimization that is very useful for VLIW processors.

5.4.6 Compilation for network processors

Network processors are a new type of processors. They are optimized for high-speed Internet applications. Their instruction sets comprise numerous instructions for accessing and processing bit fields in streams of information. Typically, they are programmed in assembly languages, since their throughput is of utmost importance. Nevertheless, network protocols are becoming more and more complex and designing compilers for such processors supports the design of network components. The necessary bit-level details have been analyzed by Wagner et al. [Wagner and Leupers, 2002]. Wagner obtained a 28% performance gain by exploiting special bit-level instructions of a network processor.

5.4.7 Compiler generation, retargetable compilers and design space exploration

When the first compilers were designed, compiler design was a totally manual process. In the meantime, some of the steps involved in generating a compiler have been automated or supported by tools. For example, lex and yacc and

more recent versions of these tools (see http://www.combo.org/lex_yacc_page) provide a standard means for parsing the source code. Generating machine instructions is another step which is now supported by tools. For example, **tree pattern matchers** such as **olive** [Sudarsanam, 1997] can be used for this task. Despite the use of such tools, compiler design is typically not a fully automated process.

However, there have been many attempts to design retargetable compilers. We distinguish between different kinds of retargetability:

- **Developer retargetability:** In this case, compiler specialists are responsible for retargeting compilers to new instruction sets.
- **User retargetability:** In this case, users are responsible for retargeting the compiler. This approach is much more challenging.

More information about retargetable compilers and their use for design space exploration can be found in a book by Leupers and Marwedel [Leupers and Marwedel, 2001].

5.5 Voltage Scaling and Power Management

5.5.1 Dynamic Voltage Scaling

Some embedded processors support dynamic voltage scheduling and dynamic power management (see page 102). An additional optimization step can be used to exploit these features. Typically, such an optimization step follows code generation by the compiler. These optimizations require a global view of all tasks of the system, including their dependencies, slack times etc.

The potential of dynamic voltage scheduling is demonstrated by the following example [Ishihara and Yasuura, 1998]. We assume that we have a processor which runs at three different voltages, 2.5 V, 4.0 V, and 5.0 V. Assuming an energy consumption of 40 nJ per cycle at 5.0 V, equation 3.1 can be used to compute the energy consumption at the other voltages (see table 5.3, where 25 nJ is a rounded value).

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

Table 5.3. Characteristics of processor with DVS

Furthermore, we assume that our task needs to execute 10^9 cycles within 25 seconds. There are several ways of doing this, as can be seen from figures 5.23 and 5.24. Using the maximum voltage (case a), see fig. 5.23), it is possible to shut down the processor during the slack time of 5 seconds (we assume the power consumption to be zero during this time).

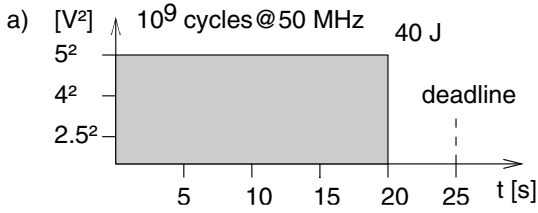

Figure 5.23. Possible voltage schedule

Another option (case b)) is to initially run the processor at full speed and then reduce the voltage when the remaining cycles can be completed at the lowest voltage (see fig. 5.24 (top)). Finally, we can run the processor at a clock rate just large enough to complete the cycles within the available time (case c), see fig. 5.24 (bottom)).

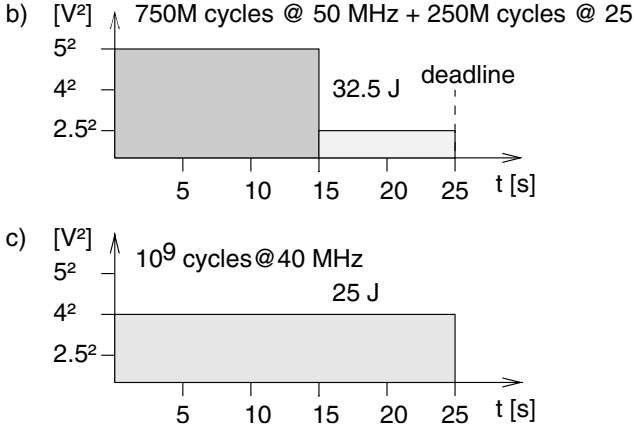

Figure 5.24. Two more possible voltage schedules

The corresponding energy consumptions can be calculated as

$$E_a = 10^9 \times 40 \cdot 10^{-9} = 40\ [J] \quad (5.15)$$

$$E_b = 750 \cdot 10^6 \times 40 \cdot 10^{-9} + 250 \cdot 10^6 \times 10 \cdot 10^{-9} = 32.5\ [J] \quad (5.16)$$

$$E_c = 10^9 \times 25 \cdot 10^{-9} = 25\ [J] \quad (5.17)$$

A minimum energy consumption is achieved for the ideal supply voltage of 4 Volts. In the following, we use the term **variable voltage processor** only for processors that allow **any** supply voltage up to a certain maximum. It is expensive to support truly variable voltages, and therefore, actual processors support only a few fixed voltages.

The observations made for the above example can be generalized into the following statements. The proofs of these statements are given in the paper by Ishihara and Yasuura.

- If a variable voltage processor completes a task before the deadline, the energy consumption can be reduced[5].
- If a processor uses a single supply voltage v and completes a task T just at its deadline, then v is the unique supply voltage which minimizes the energy consumption of T.
- If a processor can only use a number of discrete voltage levels, then a voltage schedule with at most two voltages minimizes the energy consumption under any time constraint.
- If a processor can only use a number of discrete voltage levels, then the two voltages which minimize the energy consumption are the two immediate neighbors of the ideal voltage v_{ideal} possible for a variable voltage processor.

The statements can be used for allocating voltages to tasks. Next, we will consider the allocation of voltages to a set of tasks. We will use the following notation:

N	:	the number of tasks
EC_j	:	the number of executed cycles of task j
L	:	the number of voltages of the target processor
V_i	:	the ith voltage, with $1 \leq i \leq L$
F_i	:	the clock frequency for supply voltage V_i
T	:	the global deadline at which all tasks must have been completed
$X_{i,j}$	:	the number of clock cycles task j is executed at voltage V_i
SC_j	:	the average switching capacitance during the execution of task j (SC_i comprises the actual capacitance C_L and the switching activity α (see eq. 3.1 on page 102))

The voltage scaling problem can then be formulated as an integer programming (IP) problem (see page 171). Simplifying assumptions of the IP-model include the following:

[5]This formulation makes an implicit assumption in lemma 1 of the paper by Ishihara and Yasuura explicit.

- There is one target processor that can be operated at a limited number of discrete voltages.
- The time for voltage and frequency switches is negligible.
- The worst case number of cycles for each task are known.

Using these assumptions, the IP-problem can be formulated as follows:
Minimize

$$E = \sum_{j=1}^{N} \sum_{i=1}^{L} SC_j \cdot x_{i,j} \cdot V_i^2 \tag{5.18}$$

subject to

$$\sum_{i=1}^{L} x_{i,j} = EC_j \tag{5.19}$$

and

$$\sum_{j=1}^{N} \sum_{i=1}^{L} \frac{x_{ij}}{F_i} \leq T \tag{5.20}$$

The goal is to find the number x_{ij} of cycles that each task j is executed at a certain voltage V_i . According to the statements made above, no task will ever need more than two voltages. Using this model, Ishihara and Yasuura show that efficiency is typically improved if tasks have a larger number of voltages to choose from. If large amounts of slack time are available, many voltage levels help to find close to optimal voltage levels. However, four voltage levels do already give good results quite frequently.

There are many cases in which tasks actually run faster than predicted by their worst case execution times. This cannot be exploited by the above algorithm. This limitation can be removed by using checkpoints at which actual and worst case execution times are compared, and then to use this information to potentially scale down the voltage [Azevedo et al., 2002]. Also, voltage scaling in multi-rate task graphs was recently proposed [Schmitz et al., 2002].

5.5.2 Dynamic power management (DPM)

In order to reduce the energy consumption, we can also take advantage of power saving states, as introduced on page 101. The essential question for exploiting DPM is: when should we go to a power-saving state? Straight-forward

approaches just use a simple timer to transition into a power-saving state. More sophisticated approaches model the idle times by stochastic processes and use these to predict the use of subsystems with more accuracy. Models based on exponential distributions have been shown to be inaccurate. Sufficiently accurate models include those based on renewal theory [Simunic et al., 2000].

A comprehensive discussion of power management was published by Benini et al. [Benini and Micheli, 1998]. There are also advanced algorithms which integrate DVS and DPM into a single optimization approach for saving energy [Simunic et al., 2001].

Allocating voltages and computing transition times for DPM may be two of the last steps of optimizing embedded software.

5.6 Actual design flows and tools

5.6.1 SpecC methodology

Chapter 2 includes a brief description of the SpecC language (see page 76). Fig. 5.25 shows a design flow adopted for the SpecC-based SoC methodology [Gajski et al., 2000], [Gerstlauer et al., 2001].

This methodology starts with specification capture in SpecC. The SpecC specification model is executable. Accordingly, simulations can be used to validate and analyze the model as well as to estimate certain key design parameters. The next step is architecture exploration. This step comprises allocation, partitioning and scheduling. Allocation consists of selecting components (processing elements (processors, intellectual property components, or custom hardware), memories, busses) from a library. The next step is partitioning. Partitioning denotes the mapping of parts of the system specification onto the components. Variables are mapped to memories, channels to busses, and behaviors to processing elements. Scheduling is used to serialize the execution. Fig. 5.25 describes the flow of information. The actual design exploration will consist of a number of steps that are consistent with this flow. Architecture exploration is followed by design validation (in fact, validation and estimation will typically be intermixed).

In communication synthesis, abstract busses will be replaced by actual wires in a series of refinements. In the backend, software compilers are used to generate binary machine code and hardware synthesis tools are used to generate custom hardware.

A design flow similar to the one shown is supported by the SoC Environment (SCE) that is available from the University at Irvine. Further information can be found in the SCE documentation [Center for Embedded Computer Systems, 2003].

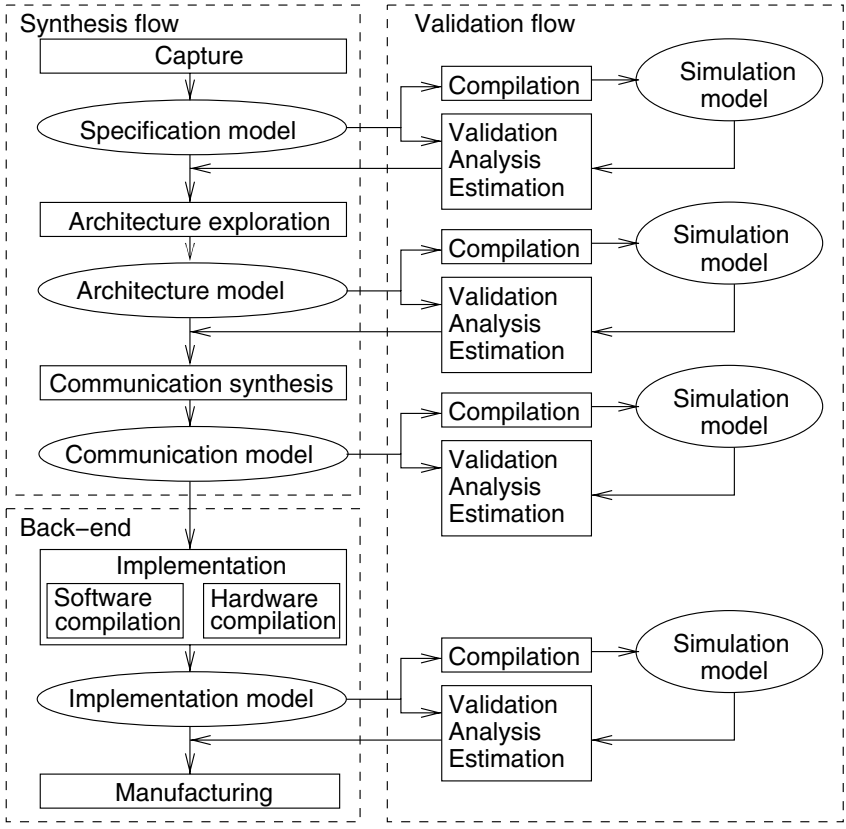

Figure 5.25. Codesign methodology possible with SpecC

5.6.2 IMEC tool flow

The design flow proposed by the "Interuniversitair Micro-Electronica Centrum" (IMEC), Leuven (Belgium) is shown in fig. 5.26. According to this design flow, specifications can be represented in UML, Java, and Concurrent C++.

- The first set of tools, developed in the context of the Matisse/Dynamic Memory Management (DMM) project, considers the system at the concurrent process level as a set of concurrent and dynamic processes, whose specification consists of four types of elements: algorithms, abstract data types, communication primitives, and real-time requirements. Tools at this level are able to perform source code transformations on the dynamic data types (and their access functions) and provide also a memory pool organization in the virtual memory space.

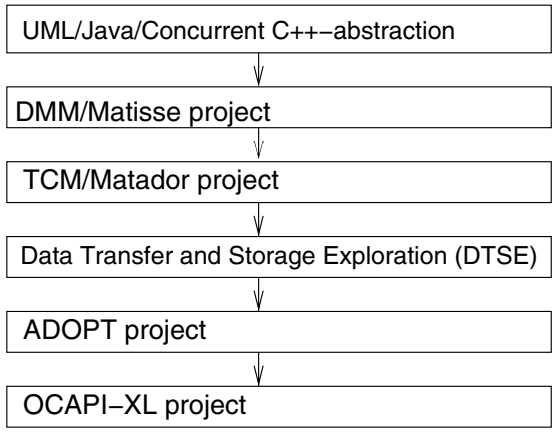

Figure 5.26. Global view of IMEC design flow

- The second set of tools, developed in the context of the Matador/Task Concurrency Management (TCM) project, is again considering a system of concurrent processes. For these tools, the emphasis is on mapping tasks to processors. Different configurations of multi-processor systems are evaluated and curves of designs that are non-inferior to others are generated. These curves provide a view of the design space, and are the basis for final design decisions. Wong et al. [Wong et al., 2001] describe configurations for a personal MPEG-4 player. The authors assume that a combination of StrongArm processors and custom accelerators is to be used and they found 4 configurations that satisfy the timing constraint of 30 ms (see table 5.4).

Processor combination	1	2	3	4
Number of high speed processors	6	5	4	3
Number of low speed processors	0	3	5	7
Total number of processors	6	8	9	10

Table 5.4. Processor configurations

For combinations 1 and 4, the authors report that only one allocation of tasks to processors meets the timing constraints. For combinations 2 and 3, different time budgets lead to different task to processor mappings and different energy consumptions.

Design space exploration is based on the concept of **Pareto curves**, as shown in fig. 5.27 for configurations 2 and 3. Each line indicates a separa-

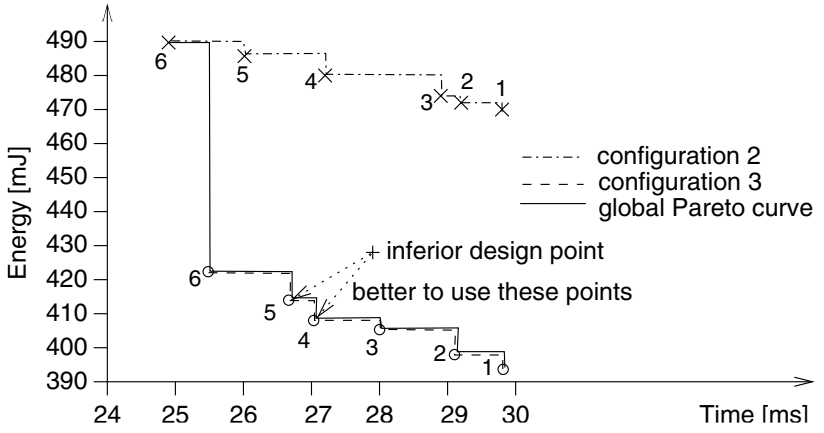

Figure 5.27. Pareto curves for processor combinations 2 and 3

tion of the design space into two subspaces. For example, the area above the dashed line (configuration 3) corresponds to design points that are inferior to the design points found for that configuration. For any design in that area, one could improve either the performance, the energy consumption or both by using the design points found for configuration 3. Hence, whenever task to processor mapping leads to a design point in that area, it is ignored[6].

For configuration 3, design point 6 is the fastest design that can be generated. If the deadline is set to less than about 25.5 ms, configuration 2 has to be used. The overall Pareto curve is obtained as the best of the Pareto curves for configurations 2 and 3. The concept of Pareto curves is frequently used for design space exploration, not just for the IMEC design flow.

TCM tools also address the storage and transfer of data between dynamically created tasks (they include a "task-level" version of the Data Transfer and Storage Exploration (DTSE) tools described next.

- The next design transformations are the subject of research in the Data Transfer and Storage Exploration (DTSE) project. A number of phases is proposed [Miranda et al., 2004], [IMEC, 2003] aiming at a reduction of the data transfers between processing components and at a reduction of the storage requirements.

- DTSE optimizations generate quite complex addressing, including modulo operations. Addressing is subsequently simplified in address optimization (ADOPT) tools [Miranda et al., 1998], [Ghez et al., 2000].

[6]IMEC has proposed to use Pareto curve information also for scheduling at run-time.

- The resulting code can be used as input to compilers or as input to the final set of IMEC tools, designed in the OCAPI-XL project. These tools support the mapping of applications to reconfigurable hardware (see page 115).

5.6.3 The COSYMA design flow

COSYMA (cosynthesis for embedded micro-architectures) [Österling et al., 1997] is a set of tools for the design of embedded systems. The COSYMA design flow is shown in fig. 5.28.

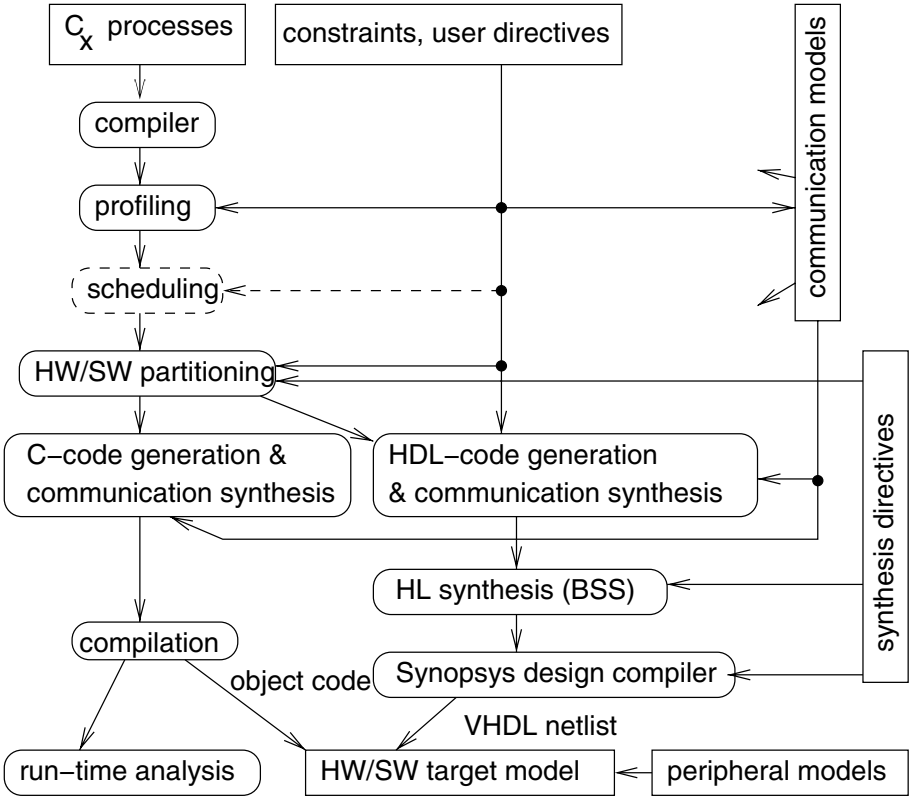

Figure 5.28. COSYMA design flow

COSYMA starts with a specification comprising a set of programs written in a slightly extended version of C, called C_x. The syntax for each C_x-program is essentially that of C, extended by a process header. Interprocess communication is based on predefined C-functions which are later mapped to physical channels. In addition, the specification includes constraints and user directives

contained in a separate file. Finally, tool-specific directives for particular tools can be provided as input.

Programs are analyzed by a compiler front-end built from the SUIF set of tools [The SUIF group, 2003]. This front-end maintains all the information included in the program code and makes it available in an internal data structure. The next step is profiling. Profiling identifies the hot spots of the application programs and provides the information needed for the optimizations that follow. Early versions of COSYMA used simulation-based profiling. Later versions also include profiling based on analytical models.

The next step is scheduling. This step is skipped (and therefore shown with dashed lines) if the input contains only a single process. For multiple processes there are two approaches. With the first approach, scheduling generates a single process from the original set of processes. With the second, newer approach, scheduling is integrated into hardware/software partitioning. The granularity used for partitioning is that of basic blocks (maximal sequences of code containing no branches, except possibly at the end). Since partitioning has to take communication costs into account, a detailed analysis of the information flow in and out of basic blocks is required.

Hardware at the block level (arithmetic units, multiplexers, etc.) is generated by the Braunschweig high-level synthesis system (BSS). The output of this system is fed into the commercial Design Compiler from Synopsys, generating gate-level descriptions. These descriptions are represented in the form of VHDL structural descriptions.

Binary object code is generated using a standard compiler for the target processor. The resulting embedded system consists of both hardware and software. Initial versions of COSYMA supported only mono-processor systems. More recent versions also support multi-processor systems. A final run-time analysis (taking communication delays into account) verifies the timing constraints.

The design flow is similar to that of COOL. However, COSYMA is a more comprehensive system resulting from the effort of a much larger group.

5.6.4 Ptolemy II

The Ptolemy project [Davis et al., 2001] focuses on modeling, simulation, and design of heterogeneous systems. Emphasis is on embedded systems that mix technologies, for example analog and digital electronics, hardware and software, and electrical and mechanical devices. Ptolemy supports different types of applications, including signal processing, control applications, sequential decision making, and user interfaces. Special attention is paid to the generation of embedded software. The idea is to generate this software from the

model of computation which is most appropriate for a certain application. Version 2 of Ptolemy (Ptolemy II) supports the following models of computation and corresponding domains (see also page 17):

1 Communicating sequential processes (CSP).

2 Continuous time (CT): This model is appropriate for mechanical systems and analog circuits. It is supported through a set of extensible differential equation solvers.

3 Discrete event model (DE): this is the model used by many simulators, e.g. VHDL simulators.

4 Distributed discrete events (DDE). Discrete event systems are difficult to simulate in parallel, due to the inherent centralized queue of future events. Attempts to distribute this data structure have not been very successful so far. Therefore, this special (experimental) domain is introduced. Semantics can be defined such that distributed simulation becomes more efficient than in the DE model.

5 Finite state machines (FSM).

6 Process networks (PN), using Kahn process networks (see page 53).

7 Synchronous dataflow (SDF).

8 Synchronous/reactive (SR) model of computation. This model uses discrete time, but signals do not need to have a value at every clock tick. Esterel (see page 79) is a language following this style of modeling.

This list clearly shows the focus on different models of computation in the Ptolemy project.

5.6.5 The OCTOPUS design flow

The OCTOPUS design flow [Awad et al., 1996] is completely dedicated towards the design of embedded software, assuming appropriate wrappers for hardware. It was observed that there is a poor match between the focus of object-oriented design techniques on the software object structure and the need to allocate operations to tasks. This poor match was the main concern that was addressed in the design of OCTOPUS. OCTOPUS is used within Nokia. Its design flow includes the following phases:

1 In the **systems requirement phase**, the behavior of the system is described by use case diagrams (see page 48) and use cases. The structure of the environment is described by a so-called context diagram.

2 In the **system architecture phase**, the structure of the system is broken down into subsystems. Major interfaces between the subsystems are identified, but the behavior of the subsystems is not.

3 The **subsystem analysis phase** is done for every subsystem. In this phase, class diagrams for the subsystems are generated. The behavior of the subsystems can be defined in various ways, including StateCharts, so-called event lists and event sheets.

4 The result of the next phase, **the subsystem design phase**, includes outlines for processes/threads, classes and interprocess messages.

5 In the final phase, called **subsystem implementation phase**, actual code of the selected programming language is generated

Obviously, this flow is very much influenced by software technology, with an adaptation towards distributed systems.

Chapter 6

VALIDATION

6.1 Introduction

One very important aspect of embedded system design has not been considered so far: validation. Validation is the process of checking whether or not a certain (possibly partial) design is appropriate for its purpose, meets all constraints and will perform as expected. Validation is important for any design procedure, and hardly any system would work as expected, had it not been validated during the design process. Validation is extremely important for safety-critical embedded systems.

In theory, we could try to design verified tools which always generate correct implementations from the specification. In practice, this verification of tools does not work, except in very simple cases. As a consequence, each and every design has to be verified. In order to minimize the number of times that we have to verify a design, we could try to verify it at the very end of the design process. Unfortunately, this approach normally does not work, due to the large differences between the level of abstraction used for the specification and that used for the implementation. Therefore, validation is required at various phases during the design procedure (see fig. 6.1). Validation and design should be intertwined and not be considered as two completely independent activities[1].

It would be nice to have a single validation technique applicable to all validation problems. In practice, none of the available techniques solves all the

[1]The same is true for design evaluation, as depicted in fig. 6.1. However, design evaluation is not covered in this book.

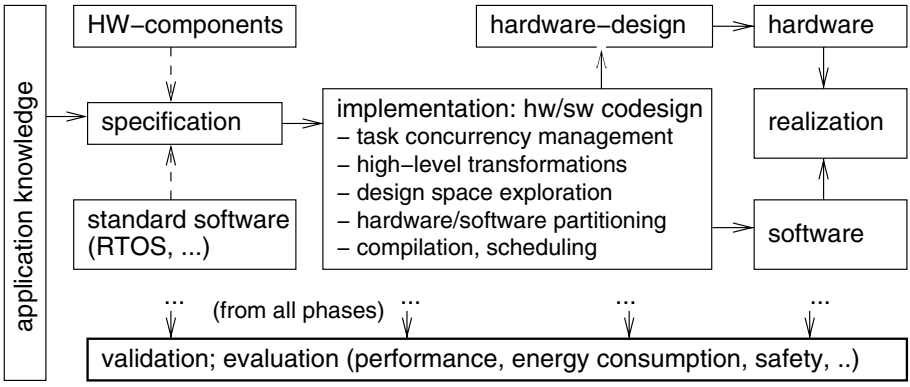

Figure 6.1. Simplified design information flow

problems, and a mix of techniques has to be applied. In this chapter, we will provide a brief overview over the key techniques which are available.

6.2 Simulation

Simulations are a very common technique for validating designs. Simulations consist of executing a design model on appropriate computing hardware, typically on general purpose digital computers. Obviously, this requires models to be executable. All the executable languages introduced in chapter 2 can be used in simulations, and they can be used at various levels as described starting at page 79.

The level at which designs are simulated is always a compromise between simulation speed and accuracy. The faster the simulation, the less accuracy is available.

So far, we have used the term behavior in the sense of the functional behavior of systems (its input/output behavior). There are also simulations of some non-functional behaviors of designs, including the thermal behavior and the electro-magnetic compatibility (EMC) with other electronic equipment.

For embedded systems, simulations have serious limitations:

- Simulations are typically a lot slower than the actual design. Hence, if we tried to interface the simulator with the actual environment, we would have quite a number of **violations of timing constraints**.
- Simulations in the real environment may even be **dangerous** (who would want to drive a car with unstable control software?).

- For many applications, there may be huge amounts of data and it may be impossible to simulate enough data in the available time. Multimedia applications are notoriously known for this. For example, simulating the compression of some video stream takes an enormous amount of time.
- Most actual systems are too complex to allow simulating all possible cases (inputs). Hence, simulations can help us to find errors in our designs. They cannot guarantee absence of errors, since simulations cannot exhaustively be done for all possible combinations of inputs and internal states.

Due to the limitations, there is an increased emphasis on formal verification (see page 209).

6.3 Rapid Prototyping and Emulation

There are many cases in which the designs should be tried out in realistic environments before final versions are manufactured. Control systems in cars are an excellent example for this. Such systems should be used by drivers in different environments before mass production is started. Accordingly, the car industry designs prototypes. These prototypes should essentially behave like the final systems, but they may be larger, more power consuming and have other properties which test drivers can accept. Such prototypes can be built, for example, using FPGAs. Racks containing FPGAs can be stored in the trunk while test drivers exercise the car.

This approach is not limited to the car industry. There are several other cases in which prototypes are built from FPGAs. Commercially available **emulators** consist of a large number of FPGAs. They come with the required mapping tools which map specifications to these emulators. Using these emulators, experiments with systems which behave "almost" like the final systems can be run.

6.4 Test

6.4.1 Scope

In testing, we are applying a set of specially selected input patterns, so-called **test patterns** to the input of the system, observe its behavior and compare this behavior with the expected behavior. Test patterns are normally applied to the real, already manufactured system. The main purpose of testing is to identify systems that have not been correctly manufactured (manufacturing test) and to identify systems that fail later (field test).

Testing includes a number of different actions:

1 **test pattern generation,**

2 **test pattern application,**

3 **response observation,** and

4 **result comparison**.

In test pattern generation, we try to identify a set of test patterns which distinguish correctly working from incorrectly working systems. Test pattern generation is based on **fault models**. Such fault models are models of possible faults. For example, it is possible to use the stuck-at-fault model, which is based on the assumption, that any internal wire of an electronic circuit is either permanently connected to '0' or '1' (this is the so-called **stuck-at model**). It has been observed that many faults actually behave as if some wire was permanently connected that way. However, recent CMOS technologies require more comprehensive fault models. These include transient faults and delay faults (faults changing the delay of a circuit) explicitly. While good fault models exist for hardware testing, the same is not true for software testing. Test pattern generation tries to generate tests for all faults that are possible according to a certain fault model. The quality of the test pattern set can be evaluated using the **fault coverage**. Fault coverage is the percentage of potential faults that can be found for a given test pattern set:

$$Coverage = \frac{\text{Number of detectable faults for a given test pattern set}}{\text{Number of faults possible due to the fault model}}$$

In practice, achieving a good product quality requires fault coverages in the area of 98 to 99 %.

In order to increase the number of options that exist for system validation, it has been proposed to use test methods already during the design phase. For example, test pattern sets can be applied to software models of systems in order to check if two software models behave in the same way. More time-consuming formal methods need to be applied only to those cases in which this test-based equivalence check did not fail.

6.4.2 Design for testability

If testing comes in only as an afterthought, it may be very difficult to test a system. For example, verifying whether or not two finite state machines are equivalent may require complex homing sequences [Kohavi, 1987] (sequences returning the FSM to some initial state). In order to simplify tests, special hardware can be added such that testing becomes easier. The process of designing for better testability is called **design for testability**, or **DfT**. Special

purpose hardware for testing finite state machines is a prominent example of this. Reaching certain states and observing states resulting from the application of input patterns is very much simplified with **scan design**. In scan design, all flip-flops storing states are connected to form serial shift registers (see fig. 6.2).

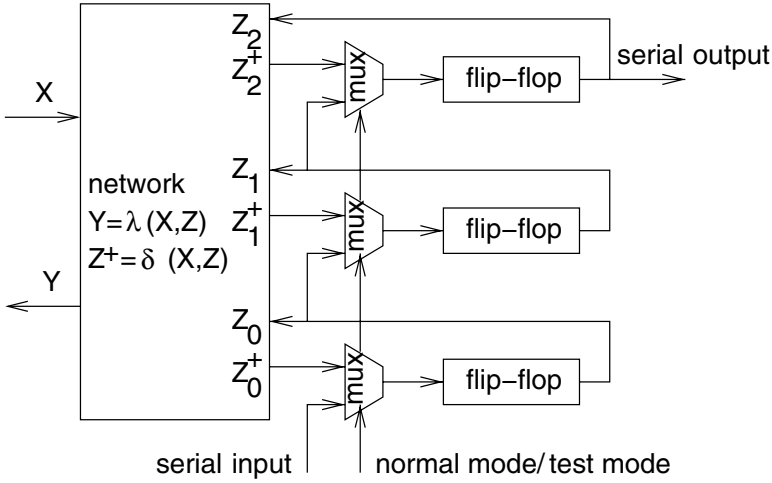

Figure 6.2. Scan path design

Setting the muxes to scan mode, we can load any state into the three flip-flops serially. In a second phase, we can apply input patterns to the FSM while the muxes are set to normal mode. As a result, the FSM will be in a new state. This new state can be serially shifted out in the third and final phase, using the serial mode again. The net effect is that we do not need to worry about how to get into certain states and how to observe whether or not δ has been correctly implemented while we are generating tests for the FSM. A safe way of testing FSM transitions is to first shift the "old" state into the shift register chain, then to apply the input, and finally to shift the resulting new state out of the scan chain. Effectively, the fact that we are dealing with state-based systems has an impact only on the two (simple) shift phases, and test pattern generation for (stateless) Boolean networks can be used for checking for correct outputs. This means that it is sufficient to use test pattern generation methods for Boolean functions (stateless networks) instead of caring about homing sequences etc.

Scan design is a technique which works well for single chips. For board-level integration it is necessary to have some technique for connecting scan chains of several chips. **JTAG** is a standard which does exactly this. The standard defines registers at the boundaries of all chips and a number of test pins and

control commands such that all chips can be connected in scan chains. JTAG is also known as boundary scan [Parker, 1992].

For chips with a large number of flip-flops, in can take quite some time to set and read all of them. In order to speed up the process of generating patterns on the chip, it has been proposed to also integrate hardware for generating test patterns on the chip. Typically pseudo-random patterns, generated by registers with feed-back paths are used as test patterns.

In order to also avoid shifting out the response of the circuit under test, responses are compacted. Compacted responses behave very much like cyclic redundancy check (CRC) characters in that the probability of generating correct compacted test responses from an incorrect response can be made very low (about 2^{-n} where n is the number of bits in the compacted response).

The **built-in logic block observer** (BILBO) [Könemann et al., 1979] has been proposed as a circuit combining test pattern generation, test response compaction and serial input/output capabilities. A BILBO with three D-type flip-flops is shown in fig. 6.3.

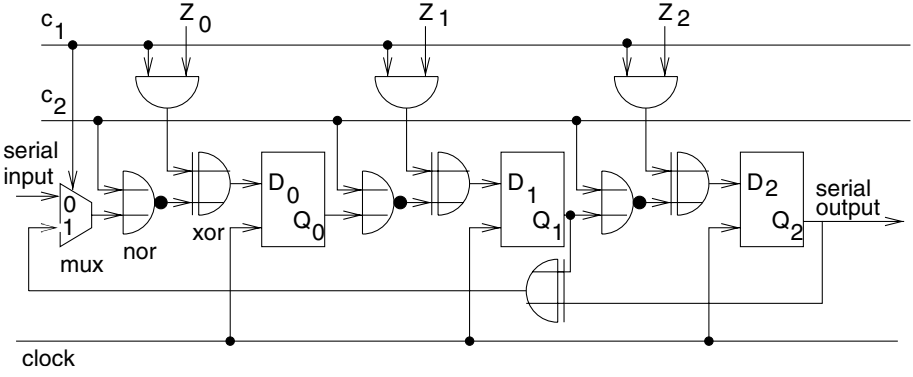

Figure 6.3. BILBO

Modes of BILBO registers are shown in table 6.1. The 3-bit register shown in fig. 6.3 can be in scan path, reset, linear-feedback shift register (LFSR) and normal mode. In LFSR mode, it can be used for either generating pseudo-random pattern or for compacting responses from inputs (Z_0 to Z_2).

Typically, BILBOs are used in pairs. One BILBO generates pseudo-random test patterns, feeding some Boolean network with these patterns. The response of the Boolean network is then compressed by a second BILBO connected to the output of the network. At the end of the test sequence, the compacted response is serially shifted out and compared with the expected response.

c_1	c_2	D_i	
'0'	'0'	$'0' \oplus \overline{Q_{i-1}} = \overline{Q_{i-1}}$	scan path mode
'0'	'1'	$'0' \oplus \overline{'1'} =' 0'$	reset
'1'	'0'	$Z_i \oplus \overline{Q_{i-1}}$	LFSR mode
'1'	'1'	$Z_i \oplus \overline{'1'} = Z_i$	normal mode

Table 6.1. Modes of BILBO registers

DfT hardware is of great help during the prototyping and debugging of hardware. It is also useful to have DfT hardware in the final product, since hardware fabrication never has a zero defect rate. Testing fabricated hardware significantly contributes to the overall cost of a product and mechanisms that reduce this cost are highly appreciated by all companies.

6.4.3 Self-test programs

One of the key problems of testing modern integrated circuits is their limited number of pins, making it more and more difficult to access internal components. Also, it is getting very difficult to test these circuits at full speed, since testers must be at least as fast as the circuits themselves. The fact that many embedded systems are based on processors provides a way out of this dilemma: processors are capable of running test programs or *diagnostics*. Such diagnostics have been used to test main frame machines for decades. Fig. 6.4 shows some components that might be contained in some processor.

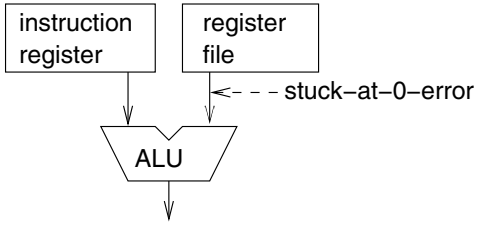

Figure 6.4. Segment from processor hardware

The types of faults that are considered are part of the **fault model**. In order to test for stuck-at-faults at the input of the ALU, we can execute a small test program:

store pattern of all '1's in register file;

perform xor between constant "0000...00" and register,

test if result contains '0' bit,

if yes, report error;

otherwise start test for next stuck-at-fault

Similar small programs can be generated for other stuck-at-errors. Unfortunately, the process of generating diagnostics for main frames has mostly been a manual one. Some researchers have proposed to generate diagnostics automatically [Brahme and Abraham, 1984], [Krüger, 1986], [Bieker and Marwedel, 1995], [Krstic and Dey, 2002].

6.5 Fault simulation

It is currently not feasible (and it will probably not be feasible) to completely predict the behavior of systems in the presence of faults. Therefore, the behavior of systems in the presence of faults is frequently simulated. This type of simulation is called **fault simulation**. In fault simulation, system models are modified to reflect the behavior of the system in the presence of a certain fault.

The goals of fault simulation include:

- to know the effect of a fault of the components at the system level. Faults are called **redundant** if they do not affect the observable behavior of the system, and
- to know whether or not mechanisms for improving fault tolerance actually help.

Fault simulation requires the simulation of the system for all faults feasible for the fault model and also for a possibly large number of different input patterns. Accordingly, fault simulation is an extremely time-consuming process. Different techniques have been proposed to speed up fault simulation. These techniques include **parallel fault simulation**. Parallel fault simulation is especially effective if the system is modeled at the gate level. In this case, internal signals are single bit signals. This fact enables the mapping of a signal to a single bit of some machine word of a simulating host machine. AND- and OR-machine instructions can then be used to simulate Boolean networks. However, only a single bit would be used per machine word. Efficiency is improved with parallel fault simulation. In parallel fault simulation, n different test patterns are simulated at the same time, if n is the machine word size. The values of each of the n test patterns are mapped to a different bit position in the machine word. Executing the same set of AND- and OR-instructions will then simulate the behavior of the Boolean network for n test patterns instead of for just one.

6.6 Fault injection

Fault simulation may be too time-consuming for real systems. If actual systems are available, fault injection can be used instead. In fault injection, real existing systems are modified and the overall effect on the system behavior is checked. Fault injection does not rely on fault models (even though they can be used). Hence, fault injection has the potential of generating faults that would not have been predicted by a fault model.

We can distinguish between two types of fault injection:

- local faults within the system, and
- faults in the environment (behaviors which do not correspond to the specification). For example, we can check how the system behaves if it is operated outside the specified temperature or radiation ranges.

Several methods can be used for fault injection:

- Fault injection at the hardware level: Examples include pin-manipulation, electromagnetic and nuclear radiation.
- Fault injection at the software level: Examples include toggling some memory bits.

According to experiments reported by Kopetz [Kopetz, 1997], software-based fault injection was essentially as effective as hardware-based fault injection. Nuclear radiation was a noticeable exception in that it generated errors which were not generated with other methods.

6.7 Risk- and dependability analysis

Embedded systems (like many products) can cause damages to properties and lives. It is not possible to reduce the risk of damages to zero. The best that we can do is to make the probability of damages small, hopefully orders of magnitude smaller than other risks. For many applications, a probability of a catastrophe has to be less than 10^{-9} per hour [Kopetz, 1997], corresponding to one case per 100,000 systems operating for 10,000 hours. Damages are resulting from **hazards**. For each possible damage there is a severity (the cost) and a probability. Risk can be defined as the product of the two.

Risks can be analyzed with several techniques [Dunn, 2002], [Press, 2003]:

- **Fault tree Analysis (FTA):** FTA is a top-down method of analyzing risks. The analysis starts with a possible damage and then tries to come up with

possible scenarios that lead to that damage. FTA typically uses a graphical representation of possible damages, including symbols for AND- and OR-gates. OR-gates are used if a single event could result in a hazard. AND-gates are used when several events or conditions are required for that hazard to exist. Fig. 6.5 shows an example.

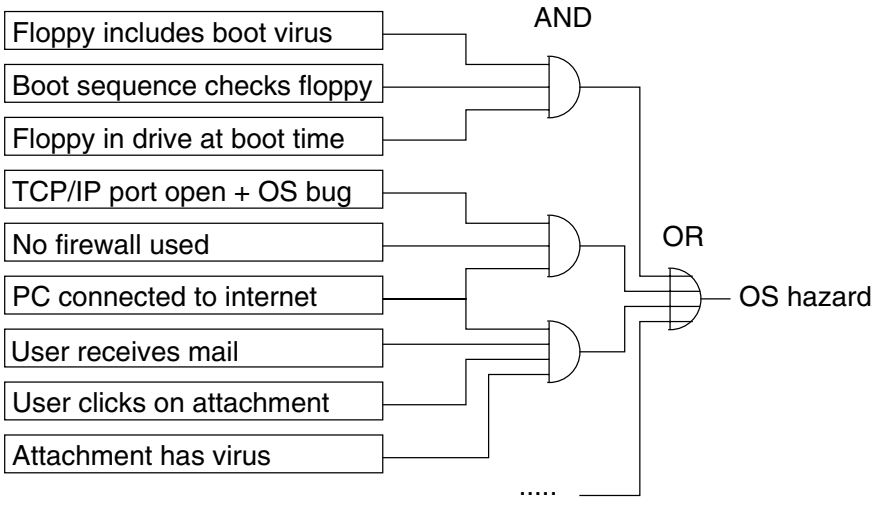

Figure 6.5. Fault tree

The simple AND- and OR-gates cannot model all situations. For example, their modeling power is exceeded if shared resources of some limited amount (like energy or storage locations) exist. Markov models [Bremaud, 1999] may have to be used to cover such cases.

- **Failure mode and effect analysis (FMEA):** FMEA starts at the components and tries to estimate their reliability. Using this information, the reliability of the system is computed from the reliability of its parts (corresponding to a bottom-up analysis). The first step is to create a table containing components, possible faults, probability of faults and consequences on the system behavior. Risks for the system as a whole are then computed from the table. Table 6.2 shows an example[2].

Tools supporting both approaches are available. Both approaches may be used in "safety cases". In such cases, an independent authority has to be convinced that certain technical equipment is indeed safe. One of the commonly requested properties of technical systems is that no single failing component should potentially cause a catastrophe.

[2]Realistic fault probabilities are unknown at the time of writing.

Component	*Failure*	*Consequences*	*Probability*	*Critical?*
Processor	metal migration	no service	10^{-6} /h	yes
...	...	...	...	...

Table 6.2. FMEA table

6.8 Formal Verification

Formal verification is concerned with formally proving a system correct, using the language of mathematics. First of all, a formal model is required to make formal verification applicable. This step can hardly be automated and may require some effort. Once the model is available, we can try to prove certain properties.

Formal verification techniques can be classified by the type of logics employed:

- **Propositional logic:** In this case, models consist of Boolean formulas described with Boolean variables and connectives such as and and or. (Stateless) gate-level logic networks can be conveniently described with propositional logic. Tools typically aim at checking if two models represented this way are equivalent. Such tools are called **tautology checkers** or **equivalence checkers**. Since propositional logic is decidable, it is also decidable whether or not the two representations are equivalent (there will be no cases of doubt). For example, one representation might correspond to gates of an actual circuit and the other to its specification. Proving the equivalence then proves the effect of all design transformations (for example, optimizations for power or delay) to be correct. Tautology checkers can frequently cope with designs which are too large to allow simulation-based exhaustive validation. The key reason for the power of recent tautology checkers is the use of Binary Decision Diagrams (BDDs) [Wegener, 2000]. The complexity of equivalence checks of Boolean functions represented with BDDs is linear in the number of BDD-nodes. In contrast, the equivalence check for functions represented by sums of products is NP-hard. Still, the number of BDD-nodes required to represent a certain function has to be taken into account. Many functions can be efficiently represented with BDDs. In the general, however, the number of nodes of BDDs grows exponentially with the number of variables. In those cases in which functions can be efficiently represented with BDDs, BDD-based equivalence checkers have frequently replaced simulators and are used to verify gate networks with millions of transistors. The ability to also verify finite state-machines is very much limited.

- **First order logic (FOL):** FOL includes quantification, using $\exists$ and $\forall$. Some automation for verifying FOL models is feasible. However, since FOL is undecidable in general, there may be cases of doubt.
- **Higher order logic (HOL):** Higher order allows functions to be manipulated like other objects (see http://archive.comlab.ox.ac.uk/formal-methods/hol.html). For higher order logic, proofs can hardly ever be automated and typically must be done manually with some proof-support.

Model checking

Verification of finite state machines can be performed with **model checking**. Model checking aims at the verification of properties of finite state systems. It analyzes the state space of the system. Verification using this approach requires three stages:

1 the generation of a model of the system to be verified,

2 the definition of the properties expected from the system, and

3 model checking (the actual verification step).

Accordingly, model checking systems accept the model and properties as input (see fig. 6.6).

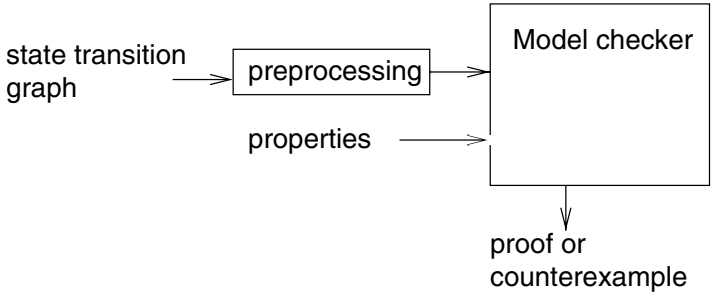

Figure 6.6. Inputs for model checking

Verification tools can prove or disprove the properties. In the latter case, they can provide a counter-example. Model checking is easier to automate than FOL. Languages used for the definition of properties typically allow the quantification of states.

A popular system for model checking is Clarke's EMC-system [Clarke and et al., 2003]. This system accepts properties to be described as CTL formulas. CTL stands for "Computational Tree Logics". CTL-formulas include two parts:

- a **path quantifier** (this part specifies paths in the state transition diagram), and
- a **state quantifier** (this part specifies states).

Example: $M, s \models AGg$ means: In the transition graph M, property g holds for all paths (denoted by A) and all states (denotes by G).

In 1987, model checking was implemented using BDDs. It was possible to locate several errors in the specification of the *future bus* protocol.

Extensions are needed in order to also cover real-time behavior and numbers. More information on formal verification can be found in books on this topic (refer to, for example, the books by Kropf [Kropf, 1999] and Clarke et al. [Clarke et al., 2000]).

References

[Aamodt and Chow, 2000] Aamodt, T. and Chow, P. (2000). Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. *3rd ACM Intern. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 128–137.

[Absint, 2002] Absint (2002). Absint: WCET analyses. *http://www.absint.de/wcet.htm*.

[Accellera, 2002] Accellera (2002). EDA industry working groups (for Rosetta). *http://www.eda.org*.

[Accellera, 2005] Accellera (2005). SystemVerilog. *http://www.systemverilog.org*.

[Ambler, 2005] Ambler, S. (2005). The diagrams of UML 2.0. *http://www.agilemodeling.com/essays/umlDiagrams.htm*.

[Araujo and Malik, 1995] Araujo, G. and Malik, S. (1995). Optimal code generation for embedded memory non-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41.

[August et al., 1997] August, D. I., Hwu, W. W., and Mahlke, S. (1997). A framework for balancing control flow and predication. *Ann. Workshop on Microprogramming and Microarchitecture (MICRO)*, pages 92–103.

[Awad et al., 1996] Awad, M., Kuusela, J., and Ziegler, J. (1996). *Object-Oriented Technology for Real-Time Systems*. Prentice Hall.

[Azevedo et al., 2002] Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., and Nicolau, A. (2002). Profile-based dynamic voltage scheduling using program checkpoints. *Design, Automation, and Test in Europe (DATE)*, pages 168–175.

[Balarin et al., 1998] Balarin, F., Lavagno, L., Murthy, P., and Sangiovanni-Vincentelli, A. (1998). Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, pages 71–82.

[Ball, 1996] Ball, S. R. (1996). *Embedded Microprocessor Systems - Real world designs*. Newnes.

[Ball, 1998] Ball, S. R. (1998). *Debugging Embedded Microprocessor Systems*. Newnes.

[Barr, 1999] Barr, M. (1999). *Programming Embedded Systems*. O'Reilly.

[Basu et al., 1999] Basu, A., Leupers, R., and Marwedel, P. (1999). Array index allocation under register constraints. *Int. Conf. on VLSI Design, Goa/India*.

[Benini and Micheli, 1998] Benini, L. and Micheli, G. D. (1998). *Dynamic Power Management – Design Techniques and CAD Tools*. Kluwer Academic Publishers.

[Berger, 2001] Berger, H. (2001). *Automating with STEP 7 in LAD and FBD: SIMATIC S7-300/400 Programmable Controllers*. Wiley.

[Bergé et al., 1995] Bergé, J.-M., Levia, O., and Rouillard, J. (1995). *High-Level System Modeling*. Kluwer Academic Publishers.

[Berkelaar and et al., 2005] Berkelaar, M. and et al. (2005). Unix^{tm} manual page of lp_solve. *Eindhoven University of Technology, Design Automation Section, current version of source code available at http://groups.yahoo.com/group/lp_solve/files*.

[Bieker and Marwedel, 1995] Bieker, U. and Marwedel, P. (1995). Retargetable self-test program generation using constraint logic programming. *32nd Design Automation Conference (DAC)*, pages 605–611.

[Boussinot and de Simone, 1991] Boussinot, F. and de Simone, R. (1991). The Esterel language. *Proc. of the IEEE, Vol. 79, No. 9*, pages 1293–1304.

[Bouyssounouse and Sifakis, 2005] Bouyssounouse, B. and Sifakis, J., editors (2005). *Embedded Systems Design, The ARTIST Roadmap for Research and Development*. Lecture Notes in Computer Science, Vol. 3436, Springer.

[Brahme and Abraham, 1984] Brahme, D. and Abraham, J. A. (1984). Functional testing of microprocessors. *IEEE Trans. on Computers*, pages 475–485.

[Bremaud, 1999] Bremaud, P. (1999). *Markov Chains*. Springer Verlag.

[Burd and Brodersen, 2000] Burd, T. and Brodersen, R. (2000). Design issues for dynamic voltage scaling. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 9–14.

[Burd and Brodersen, 2003] Burd, T. and Brodersen, R. W. (2003). *Energy efficient microprocessor design*. Kluwer Academic Publishers.

[Burkhardt, 2001] Burkhardt, J. (2001). *Pervasive Computing*. Addison-Wesley.

[Burns and Wellings, 1990] Burns, A. and Wellings, A. (1990). *Real-Time Systems and Their Programming Languages*. Addison-Wesley.

[Burns and Wellings, 2001] Burns, A. and Wellings, A. (2001). *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley.

[Buttazzo, 2002] Buttazzo, G. (2002). *Hard Real-time computing systems*. Kluwer Academic Publishers, 4th printing.

[Byteflight Consortium, 2003] Byteflight Consortium (2003). Home page. *http://www.byteflight.com*.

[Camposano and Wolf, 1996] Camposano, R. and Wolf, W. (1996). Message from the editors-in-chief. *Design Automation for Embedded Systems*.

[Center for Embedded Computer Systems, 2003] Center for Embedded Computer Systems (2003). SoC Environment. *http://www.cecs.uci.edu/~cad/sce.html*.

[Chandrakasan et al., 1992] Chandrakasan, A., Sheng, S., and Brodersen, R. W. (1992). Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):119–123.

[Chandrakasan et al., 1995] Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. (1995). Low power CMOS digital design. *Kluwer Academic Publishers*.

[Chetto et al., 1990] Chetto, H., Silly, M., and Bouchentouf, T. (1990). Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems, 2*.

[Chung et al., 2001] Chung, E.-Y., Benini, L., and Micheli, G. D. (2001). Source code transformation based on software cost analysis. In *Int. Symp. on System Synthesis (ISSS)*, pages 153–158.

[Cinderella ApS, 2003] Cinderella ApS (2003). home page. *http://www.cinderella.dk*.

[Clarke and et al., 2003] Clarke, E. and et al. (2003). Model checking@CMU. *http://www-2.cs.cmu.edu/~modelcheck/index.html*.

[Clarke et al., 2000] Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking*. The MIT Press; Second printing.

[Clouard et al., 2003] Clouard, A., Jain, K., Ghenassia, F., Maillet-Contoz, L., and Strassen, J. (2003). Using transactional models in SoC design flow. *in [Müller et al., 2003]*, pages 29–64.

[Coelho, 1989] Coelho, D. R. (1989). The VHDL handbook. *Kluwer Academic Publishers*.

[Cortadella et al., 2000] Cortadella, J., Kondratyev, A., Lavagno, L., Massot, M., Moral, S., Passerone, C., Watanabe, Y., and Sangiovanni-Vincentelli, A. (2000). Task generation and compile-time scheduling for mixed data-control embedded software. *Design automation conference (DAC)*, pages 489–494.

[Dasgupta, 1979] Dasgupta, S. (1979). The organization of microprogram stores. *ACM Computing Surveys, Vol. 11*, pages 39–65.

[Davis et al., 2001] Davis, J., Hylands, C., Janneck, J., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S., Sachs, S., Stewart, M., Vissers, K., Whitaker, P., and Xiong, Y. (2001). Overview of the Ptolemy project. *Technical Memorandum UCB/ERL M01/11; http://ptolemy.eecs.berkeley.edu*.

[De Greef et al., 1997a] De Greef, E., Catthoor, F., and Man, H. D. (1997a). Array placement for storage size reduction in embedded multimedia systems. *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 66–75.

[De Greef et al., 1997b] De Greef, E., F.Catthoor, and Man, H. (1997b). Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Intern. Parallel Proc. Symp.(IPPS) in Proc. Workshop on "Parallel Processing and Multimedia"*, pages 84–98.

[De Man, 2002] De Man, H. (2002). Keynote session at DATE'02. *http://www.date-conference.com/conference/2002/keynotes/index.htm*.

[Deutsches Institut für Normung, 1997] Deutsches Institut für Normung (1997). *DIN 66253, Programmiersprache PEARL, Teil 2 PEARL 90*. Beuth-Verlag; English version available through http://www.din.de.

[Dierickx, 2000] Dierickx, B. (2000). CMOS image sensors - concepts, Photonics West 2000 short course. *http://www.fillfactory.com/htm/technology/htm/publications.htm*.

[Douglass, 2000] Douglass, B. P. (2000). *Real-Time UML, 2nd edition*. Addison Wesley.

[Drusinsky and Harel, 1989] Drusinsky, D. and Harel, D. (1989). Using statecharts for hardware description and synthesis. *IEEE Trans. on Computer Design*, pages 798–807.

[Dunn, 2002] Dunn, W. (2002). *Practical Design of Safety-Critical Computer Systems*. Reliability Press.

[Elsevier B.V., 2003a] Elsevier B.V. (2003a). Sensors and actuators A: Physical. *An International Journal*.

[Elsevier B.V., 2003b] Elsevier B.V. (2003b). Sensors and actuators B: Chemical. *An International Journal*.

[Esterel, 2002] Esterel, T. I. (2002). Homepage. *http://www.esterel-technologies.com*.

[Falk and Marwedel, 2003] Falk, H. and Marwedel, P. (2003). Control flow driven splitting of loop nests at the source code level. *Design, Automation and Test in Europe (DATE)*, pages 410–415.

[Fettweis et al., 1998] Fettweis, G., Weiss, M., Drescher, W., Walther, U., Engel, F., Kobayashi, S., and Richter, T. (1998). Breaking new grounds over 3000 MMAC/s: a broadband mobile multimedia modem DSP. *Intern. Conf. on Signal Processing Application & Technology (ICSPA), available at http://citeseer.ist.psu.edu/111037.html*.

[Fisher and Dietz, 1998] Fisher, R. and Dietz, H. G. (1998). Compiling for SIMD within a single register. *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 290–304.

[Fisher and Dietz, 1999] Fisher, R. J. and Dietz, H. G. (1999). The Scc compiler: SWARing at MMX and 3DNow! *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 399–414.

[FlexRay Consortium, 2002] FlexRay Consortium (2002). Flexray® requirement specification. version 2.01. *http://www.flexray.de*.

[Fowler and Scott, 1998] Fowler, M. and Scott, K. (1998). *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley.

[Fu et al., 1987] Fu, K., Gonzalez, R., and Lee, C. (1987). *Robotics*. McGraw-Hill.

[Gajski et al., 1994] Gajski, D., Vahid, F., Narayan, S., and Gong, J. (1994). *Specification and Design of Embedded Systems*. Prentice Hall.

[Gajski et al., 2000] Gajski, D., Zhu, J., Dömer, R., Gerstlauer, A., and Zhao, S. (2000). *SpecC: Specification Language Methodology*. Kluwer Academic Publishers.

[Ganssle, 1992] Ganssle, J. G. (1992). *Programming Embedded Systems*. Academic Press.

[Ganssle, 2000] Ganssle, J. G. (2000). *The Art of Designing Embedded Systems*. Newnes.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability*. Bell Labaratories, Murray Hill, New Jersey.

[Geffroy and Motet, 2002] Geffroy, J.-C. and Motet, G. (2002). *Design of Dependable computing Systems*. Kluwer Academic Publishers.

[Gelsen, 2003] Gelsen, O. (2003). Organic displays enter consumer electronics. *Opto & Laser Europe, June; availabe at http://optics.org/articles/ole/8/6/6/1*.

[Gerstlauer et al., 2001] Gerstlauer, A., Dömer, R., Peng, J., and Gajski, D. (2001). *System Design: A Practical Guide with SystemC*. Kluwer Academic Publishers.

[Ghez et al., 2000] Ghez, C., Miranda, M., Vandecapelle, A., et al. (2000). Systematic high-level address code transformations for piece-wise linear indexing. *Proc. of SIPS*.

[Gupta, 1998] Gupta, R. (1998). Introduction to embedded systems. *http://www.ics.uci.edu/~rgupta/ics212.html*.

[Halbwachs, 1998] Halbwachs, N. (1998). Synchronous programming of reactive systems, a tutorial and commented bibliography. *Tenth International Conference on Computer-Aided Verification, CAV'98, LNCS 1427, Springer Verlag; see also: http://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/cav98tutorial.html*.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow language LUSTRE. *Proc. of the IEEE*, 79:1305–1320.

[Hansmann, 2001] Hansmann, U. (2001). *Pervasive Computing*. Springer Verlag.

[Harbour, 1993] Harbour, M. G. (1993). RT-POSIX: An overview. *http://www.ctr.unican.es/publications/mgh-1993a.pdf*.

[Harel, 1987] Harel, D. (1987). StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274.

[Hayes, 1982] Hayes, J. (1982). A unified switching theory with applications to VLSI design. *Proceedings of the IEEE, Vol.70*, pages 1140–1151.

[Healy et al., 1999] Healy, C., Arnold, R., Mueller, F., Whalley, D., and Harmon, M. (1999). Bounding pipeline and instructions cache performance. *IEEE Transactions on Computers*, pages 53–70.

[Hennessy and Patterson, 1995] Hennessy, J. L. and Patterson, D. A. (1995). *Computer Organization – The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc.

[Hennessy and Patterson, 1996] Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[Herrera et al., 2003a] Herrera, F., Fernández, V., Sánchez, P., and Villar, E. (2003a). Embedded software generation from SystemC for platform based design. *in [Müller et al., 2003]*, pages 247–272.

[Herrera et al., 2003b] Herrera, F., Posadas, H., Sánchez, P., and Villar, E. (2003b). Systemic embedded software generation from SystemC. *Design, Automation and Test in Europe (DATE)*, pages 10142–10149.

[Hoare, 1985] Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

[Horn, 1974] Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly, Vol. 21*, pages 177–185.

[Huerlimann, 2003] Huerlimann, D. (2003). Opentrack home page. *http://www.opentrack.ch.*

[Hüls, 2002] Hüls, T. (2002). Optimizing the energy consumption of an MPEG application (in German). *Master thesis, CS Dept., Univ. Dortmund, http://ls12-www.cs.uni-dortmund.de/publications/theses*.

[IBM Inc., 2002] IBM Inc. (2002). Security: User authentication. *http://www.pc.ibm.com/us/security/userauth.html*.

[IEC, 2002] IEC (2002). GRAFCET specification language for sequential function charts. *http://tc3.iec.ch/txt/147.htm*.

[IEEE, 1997] IEEE (1997). *IEEE Standard VHDL Language Reference Manual (1076-1997)*. IEEE.

[IEEE, 1992] IEEE, D. (1992). Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*.

[IMEC, 1997] IMEC (1997). LIC-SMARTpen identifies signer. *IMEC Newsletter, http://www.imec.be/wwwinter/mediacenter/en/newsletter_18.pdf*.

[IMEC, 2003] IMEC (2003). Design technology program. *http://www.imec.be/ovinter/static_research/designtechnology.shtml*.

[IMEC Desics group, 2003] IMEC Desics group (2003). Task concurrency management (overview of IMEC activities). *http://www.imec.be/design/tcm/*.

[Intel, 2005] Intel (2005). Intel Itanium 2 processor. *http://www.intel.com/products/processor/itanium2*.

[Ishihara and Yasuura, 1998] Ishihara, T. and Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 197–2002.

[Iyer and Marculescu, 2002] Iyer, A. and Marculescu, D. (2002). Power and performance evaluation of globally asynchronous locally synchronous processors. *Intern. Symp. on Computer Architecture (ISCA)*, pages 158–168.

[Jackson, 1955] Jackson, J. (1955). Scheduling a production line to minimize maximum tardiness. *Management Science Research Project 43, University of California, Los Angeles*.

[Jacome et al., 2000] Jacome, M., de Veciana, G., and Lapinskii, V. (2000). Exploring performance tradeoffs for clustered VLIW ASIPs. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 504–510.

[Jacome and de Veciana, 1999] Jacome, M. F. and de Veciana, G. (1999). Lower bound on latency for VLIW ASIP datapaths. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 261–269.

[Jain et al., 2001] Jain, M., Balakrishnan, M., and Kumar, A. (2001). ASIP design methodologies : Survey and issues. *Fourteenth International Conference on VLSI Design*, pages 76–81.

[Janka, 2002] Janka, R. (2002). *Specification and Design Methodology for Real-Time Embedded Systems*. Kluwer Academic Publishers.

[Jantsch, 2003] Jantsch, A. (2003). *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann.

[Java Community Process, 2002] Java Community Process (2002). JSR-1 – real-time specification for Java. *http://www.jcp.org/en/jsr/detail?id=1*.

[Jeffrey and Leduc, 1996] Jeffrey, A. and Leduc, G. (1996). E-LOTOS core language. *http://citeseer.ist.psu.edu/jeffrey96elotos.html*.

[Jha and Dutt, 1993] Jha, P. and Dutt, N. (1993). Rapid estimation for parameterized components in high-level synthesis. *IEEE Transactions on VLSI Systems*, pages 296–303.

[Jones, 1997] Jones, M. (1997). What really happened on Mars Rover Pathfinder. *in: P.G.Neumann (ed.): comp.risks, The Risks Digest, Vol. 19, Issue 49; available at http://www.cs.berkeley.edu/~brewer/cs262/PriorityInversion.html*.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Proc. of the IFIP Congress 74*, pages 471–475.

[Keding et al., 1998] Keding, H., Willems, M., Coors, M., and Meyr, H. (1998). FRIDGE: A fixed-point design and simulation environment. *Design, Automation and Test in Europe (DATE)*, pages 429–435.

[Kempe, 1995] Kempe, M. (1995). Ada 95 reference manual, iso/iec standard 8652:1995. *(HTML-version), http://www.adahome.com/rm95/*.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall.

[Kienhuis et al., 2000] Kienhuis, B., Rijjpkema, E., and Deprettere, E. (2000). Compaan: Deriving process networks from Matlab for embedded signal processing architectures. *Proc. 8th Intern. Workshop on Hardware/Software Codesign (CODES)*.

[Kobryn, 2001] Kobryn, C. (2001). UML 2001: A standardization Odyssey. *Communication of the ACM (CACM), available at http://www.omg.org/attachments/pdf/UML_2001_CACM_Oct99_p29-Kobryn.pdf*, pages 29–36.

[Kohavi, 1987] Kohavi, Z. (1987). *Switching and Finite Automata Theory*. Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint.

[Koninklijke Philips Electronics N.V., 2003] Koninklijke Philips Electronics N.V. (2003). Ambient intelligence. *http://www.philips.com/research/ami*.

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems –Design Principles for Distributed Embedded Applications–*. Kluwer Academic Publishers.

[Kopetz, 2003] Kopetz, H. (2003). Architecture of safety-critical distributed real-time systems. *Invited Talk; Design, Automation, and Test in Europe (DATE)*.

[Kopetz and Grunsteidl, 1994] Kopetz, H. and Grunsteidl, G. (1994). TTP –a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27:14–23.

[Krall, 2000] Krall, A. (2000). Compilation techniques for multimedia extensions. *International Journal of Parallel Programming*, 28:347–361.

[Krishna and Shin, 1997] Krishna, C. and Shin, K. G. (1997). *Real-Time Systems*. McGraw-Hill, Computer Science Series.

[Kropf, 1999] Kropf, T. (1999). *Introduction to Formal Hardware Verification*. Springer-Verlag.

[Krstic and Dey, 2002] Krstic, A. and Dey, S. (2002). Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test*, pages 18–27.

[Krüger, 1986] Krüger, G. (1986). Automatic generation of self-test programs: A new feature of the MIMOLA design system. *23rd Design Automation Conference (DAC)*, pages 378–384.

[Kuchcinski, 2002] Kuchcinski, K. (2002). System partitioning (course notes). *http://www.cs.lth.se/home/Krzysztof_Kuchcinski/DES/Lectures/Lecture7.pdf*.

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocation directed task graphs to multiprocessors. *ACM Computing Surveys*, 31:406–471.

[Könemann et al., 1979] Könemann, B., Mucha, J., and Zwiehoff, G. (1979). Built-in logic block observer. *Proc. IEEE Intern. Test Conf.*, pages 261–266.

[Lam et al., 1991] Lam, M. S., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *Proceedings of ASPLOS IV*, pages 63–74.

[Landwehr and Marwedel, 1997] Landwehr, B. and Marwedel, P. (1997). A new optimization technique for improving resource exploitation and critical path minimization. *10th International Symposium on System Synthesis (ISSS)*, pages 65–72.

[Lapinskii et al., 2001] Lapinskii, V., Jacome, M. F., and de Veciana, G. (2001). Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space. Technical Report UT-CERC-TR-MFJ/GDV-01-1, Computer Engineering Research Center, University of Texas at Austin.

[Laprie, 1992] Laprie, J. C., editor (1992). *Dependability: basic concepts and terminology in English, French, German, Italian and Japanese*. IFIP WG 10.4, Dependable Computing and Fault Tolerance, in: volume 5 of Dependable computing and fault tolerant systems, Springer Verlag.

[Larsen and Amarasinghe, 2000] Larsen, S. and Amarasinghe, S. (2000). Exploiting superword parallelism with multimedia instructions sets. *Programming Language Design and Implementation (PLDI)*, pages 145–156.

[Lawler, 1973] Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Managements Science, Vol. 19*, pages 544–546.

[Lee, 1999] Lee, E. (1999). Embedded software – an agenda for research. Technical report, UCB ERL Memorandum M99/63.

[Lee and Messerschmitt, 1987] Lee, E. and Messerschmitt, D. (1987). Synchronous data flow. *Proc. of the IEEE, vol. 75*, pages 1235–1245.

[Lee et al., 2001] Lee, S., Ermedahl, A., , and Min, S. (2001). An accurate instruction-level energy consumption model for embedded risc processors. *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[Leupers, 1997] Leupers, R. (1997). *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers.

[Leupers, 1999] Leupers, R. (1999). Exploiting conditional instructions in code generation for embedded VLIW processors. *Design, Automation and Test in Europe (DATE)*.

[Leupers, 2000a] Leupers, R. (2000a). *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers.

[Leupers, 2000b] Leupers, R. (2000b). Code selection for media processors with SIMD instructions. *Design, Automation and Test in Europe (DATE)*, pages 4–8.

[Leupers, 2000c] Leupers, R. (2000c). Instruction scheduling for clustered VLIW DSPs. *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 291–300.

[Leupers and David, 1998] Leupers, R. and David, F. (1998). A uniform optimization technique for offset assignment problems. *Int. Symp. on System Synthesis (ISSS)*, pages 3–8.

[Leupers and Marwedel, 1995] Leupers, R. and Marwedel, P. (1995). Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, pages 54–59.

[Leupers and Marwedel, 1996] Leupers, R. and Marwedel, P. (1996). Algorithms for address assignment in DSP code generation. *IEEE Int. Conf. on Computer-Aided Design (ICCAD*, pages 109–112.

[Leupers and Marwedel, 1999] Leupers, R. and Marwedel, P. (1999). Function inlining under code size constraints for embedded processors. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 253–256.

[Leupers and Marwedel, 2001] Leupers, R. and Marwedel, P. (2001). *Retargetable Compiler Technology for Embedded Systems – Tools and Applications*. Kluwer Academic Publishers.

[Leveson, 1995] Leveson, N. (1995). *Safeware, System safety and Computers*. Addison Wesley.

[Liao et al., 1995a] Liao, S., Devadas, S., Keutzer, K., and Tijang, S. (1995a). Code optimization techniques for embedded DSP microprocessors. *32nd Design Automation Conference (DAC)*, pages 599–604.

[Liao et al., 1995b] Liao, S., Devadas, S., Keutzer, K., Tijang, S., and Wang, A. (1995b). Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, pages 186–195.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery (JACM)*, pages 40–61.

[Liu, 2000] Liu, J. W. (2000). *Real-Time Systems*. Prentice Hall.

[Lorenz et al., 2002] Lorenz, M., Wehmeyer, L., Draeger, T., and Leupers, R. (2002). Energy aware compilation for DSPs with SIMD instructions. *LCTES/SCOPES '02*, pages 94–101.

[Machanik, 2002] Machanik, P. (2002). Approaches to addressing the memory wall. *Technical Report, November, Univ. Brisbane*.

[Mahlke et al., 1992] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. (1992). Effective compiler support for predicated execution using the hyperblock. *MICRO-92*, pages 45–54.

[Marwedel and Goossens, 1995] Marwedel, P. and Goossens, G., editors (1995). *Code Generation for Embedded Processors*. Kluwer Academic Publishers.

[Marwedel and Schenk, 1993] Marwedel, P. and Schenk, W. (1993). Cooperation of synthesis, retargetable code generation and testgeneration in the MSS. *EDAC-EUROASIC'93*, pages 63–69.

[Marzano and Aarts, 2003] Marzano, S. and Aarts, E. (2003). *The New Everyday*. 010 Publishers.

[McLaughlin and Moore, 1998] McLaughlin, M. and Moore, A. (1998). Real-Time Extensions to UML. *http://www.ddj.com/articles/1998/9812g/9812.htm*.

[Menard and Sentieys, 2002] Menard, D. and Sentieys, O. (2002). Automatic evaluation of the accuracy of fixed-point algorithms. *Design, Automation and Test in Europe (DATE)*, pages 529–535.

[Mermet et al., 1998] Mermet, J., Marwedel, P., Ramming, F. J., Newton, C., Borrione, D., and Lefaou, C. (1998). Three decades of hardware description languages in europe. *Journal of Electrical Engineering and Information Science*, 3:106pp.

[Microsoft Inc., 2003] Microsoft Inc. (2003). Microsoft windows embedded. *microsoft.com/windows/embedded/default.mspx*.

[Miranda et al., 2004] Miranda, M., Brockmeyer, E., Meeuwen, T. V., Ghez, C., and Catthoor, F. (2004). Low power data transfer and communication for SoC. *in: C. Piquet (ed.): Lower Power Electronics and Design, CRC Press, (to be published)*.

[Miranda et al., 1998] Miranda, M., F.Catthoor, M.Janssen, and Man, H. (1998). High-level address optimisation and synthesis techniques for data-transfer intensive applications. *IEEE Trans. on VLSI Systems*, pages 677–686.

[MPI/RT forum, 2001] MPI/RT forum (2001). Document for the real-time message passing interface (mpi/rt-1.1. *http://www.mpirt.org/drafts/mpirt-report-18dec01.pdf*.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, Inc.

[Müller et al., 2003] Müller, W., Rosenstiel, W., and Ruf, J. (2003). *SystemC – Methodologies and Applications*. Kluwer Academic Publications.

[Neumann, 1995] Neumann, P. G. (1995). *Computer Related Risks*. Addison Wesley.

[Niemann, 1998] Niemann, R. (1998). *Hardware/Software Co-Design for Data-Flow Dominated Embedded Systems*. Kluwer Academic Publishers.

[Nilsen, 2004] Nilsen, K. (2004). Real-Time Java. *http://www.newmonics.com/about/tech/realjava.shtml*.

[Object Management Group (OMG), 2002] Object Management Group (OMG) (2002). Real-time CORBA specification, version 1.1, august 2002. *Object Management Group, http://www.omg.org/docs/formal/02-08-02.ps*.

[Object Management Group (OMG), 2003] Object Management Group (OMG) (2003). CORBA®basics. *http://www.omg.org/gettingstarted/corbafaq.htm*.

[OMG, 2005] OMG (2005). UML^{TM} resource page. *http://www.uml.org*.

[Oppenheim et al., 1999] Oppenheim, A. V., Schafer, R., and Buck, J. R. (1999). *Digital Signal Processing*. Pearson Higher Education.

[Österling et al., 1997] Österling, A., Benner, T., Ernst, R., Herrmann, D., Scholz, T., and Ye, W. (1997). The COSYMA system. *http://www.ida.ing.tu-bs.de/research/projects/cosyma/overview/node1.html*.

[Palkovic et al., 2002] Palkovic, M., Miranda, M., and Catthoor, F. (2002). Systematic power-performance trade-off in MPEG-4 by means of selective function inlining steered by address optimisation opportunities. *Design, Automation, and Test in Europe (DATE)*, pages 1072–1079.

[palowireless, 2003] palowireless (2003). HomeRF Resource Center. *http://www.homerf.org*.

[Parker, 1992] Parker, K. P. (1992). *The Boundary Scan Handbook*. Kluwer Academic Press.

[Pino and Lee, 1995] Pino, J. L. and Lee, E. (1995). Hierarchical static scheduling of dataflow graphs onto multiple processors. *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*.

[Poseidon, 2003] Poseidon (2003). Documentation for Poseidon for UML. *http://www.gentleware.com/index.php?id=documentation*.

[Press, 2003] Press, D. (2003). *Guidelines for Failure Mode and Effects Analysis for Automotive, Aerospace and General Manufacturing Industries*. CRC Press.

[Ramamritham, 2002] Ramamritham, K. (2002). System support for real-time embedded systems. *in: Tutorial 1, 39th Design Automation Conference (DAC)*.

[Ramamritham et al., 1998] Ramamritham, K., Shen, C., Gonzalez, O., Sen, S., and Shirgurkar, S. B. (1998). Using Windows NT for real-time applications: Experimental observations and recommendations. *IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 102–111.

[Reisig, 1985] Reisig, W. (1985). *Petri nets*. Springer Verlag.

[Rixner et al., 2000] Rixner, S., Dally, W. J., Khailany, B., Mattson, P., Kapasi, U. J., and Owens, J. D. (2000). Register organization for media processing. *6th Intern. Symp. on High-Performance Computer Architecture*, pages 375–386.

[Russell and Jacome, 1998] Russell, T. and Jacome, M. F. (1998). Software power estimation and optimization for high performance, 32-bit embedded processors. *Proc. International Conference on Computer Design (ICCD)*, pages 328–333.

[Ryan, 1995] Ryan, M. (1995). Market focus – insight into markets that are making the news in EE Times. *http://techweb.cmp.com/techweb/eet/embedded/embedded.html(Sept.11)*.

[Sangiovanni-Vincentelli, 2002] Sangiovanni-Vincentelli, A. (2002). The context for platform-based design. *IEEE Design & Test of Computers*, page 120.

[Schmitz et al., 2002] Schmitz, M., Al-Hashimi, B., and Eles, P. (2002). Energy-efficient mapping and scheduling for dvs enabled distributed embedded systems. *Design, Automation and Test in Europe (DATE)*, pages 514–521.

[SDL Forum Society, 2003a] SDL Forum Society (2003a). Home page. *http://www.sdl-forum.org*.

[SDL Forum Society, 2003b] SDL Forum Society (2003b). List of commercial tools. *http://www.sdl-forum.org/Tools/Commercial.htm*.

[Sedgewick, 1988] Sedgewick, R. (1988). *Algorithms*. Addison-Wesley.

[SEMATECH, 2003] SEMATECH (2003). International technology roadmap for semiconductors (ITRS). *http://public.itrs.net*.

[Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Trans. on Computers*, pages 1175–1185.

[Shi and Brodersen, 2003] Shi, C. and Brodersen, R. (2003). An automated floating-point to fixed-point conversion methodology. *Int. Conf. on Audio Speed and Signal Processing (ICASSP)*, pages 529–532.

[Simunic et al., 2000] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and Micheli, G. D. (2000). Energy efficient design of portable wireless devices. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 49–54.

[Simunic et al., 2001] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and Micheli, G. D. (2001). Dynamic voltage scaling and power management for portable systems. *Design Automation Conference (DAC)*, pages 524–529.

[Simunic et al., 1999] Simunic, T., Benini, L., and De Micheli, G. (1999). Cycle-accurate simulation of energy consumption in embedded systems. *Design Automation Conference (DAC)*, pages 876–872.

[Society for Display Technology, 2003] Society for Display Technology (2003). Home page. *http://www.sid.org*.

[Spivey, 1992] Spivey, M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition.

[Stankovic and Ramamritham, 1991] Stankovic, J. and Ramamritham, K. (1991). The Spring kernel: a new paradigm for real-time systems. *IEEE Software*, 8:62–72.

[Stankovic et al., 1998] Stankovic, J., Spuri, M., Ramamritham, K., and Buttazzo, G. (1998). *Deadline Scheduling for Real-Time Systems, EDF and related algorithms*. Kluwer Academic Publishers.

[Steinke et al., 2002a] Steinke, S., Grunwald, N., Wehmeyer, L., Banakar, R., Balakrishnan, M., and Marwedel, P. (2002a). Reducing energy consumption by dynamic copying of instructions onto onchip memory. *Int. Symp. on System Synthesis (ISSS)*, pages 213–218.

[Steinke et al., 2001] Steinke, S., Knauer, M., Wehmeyer, L., and Marwedel, P. (2001). An accurate and fine grain instruction-level energy model supporting software optimizations. *Proc. of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.

[Steinke et al., 2002b] Steinke, S., L.Wehmeyer, Lee, B.-S., and Marwedel, P. (2002b). Assigning program and data objects to scratchpad for energy reduction. *Design, Automation and Test in Europe (DATE)*, pages 409–417.

[Stiller, 2000] Stiller, A. (2000). New processors (in German). *c't*, 22:52.

[Storey, 1996] Storey, N. (1996). *Safety-critical Computer Systems*. Addison Wesley.

[Stritter and Gunter, 1979] Stritter, E. and Gunter, T. (1979). Microprocessor architecture for a changing world: The Motorola 68000. *IEEE Computer*, 12:43–52.

[Sudarsanam, 1997] Sudarsanam, A. (1997). SPAM compiler release. *http://www.ee.princeton.edu/~spam/release.html*.

[Sudarsanam et al., 1997] Sudarsanam, A., Liao, S., and Devadas, S. (1997). Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. *Design Automation Conference (DAC)*, pages 287–292.

[Sudarsanam and Malik, 1995] Sudarsanam, A. and Malik, S. (1995). Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392.

[Synopsys, 2005] Synopsys (2005). System studio. *http://www.synopsys.com/products/cocentric_studio*.

[SystemC, 2002] SystemC (2002). Home page. *http://www.SystemC.org*.

[Takada, 2001] Takada, H. (2001). Real-time operating system for embedded systems. *in: M. Imai and N. Yoshida (eds.): Tutorial 2 – Software Development Methods for Embedded Systems, Asia South-Pacific Design Automation Conference (ASP-DAC)*.

[Tan et al., 2003] Tan, T. K., Raghunathan, A., and Jha, N. K. (2003). Software architectural transformations: A new approach to low energy embedded software. *Design, Automation and Test in Europe (DATE)*, pages 11046–11051.

[Teich et al., 1999] Teich, J., Zitzler, E., and Bhattacharyya, S. (1999). 3D exploration of software schedules for DSP algorithms. *CODES'99*, page 168pp.

[Telelogic, 1999] Telelogic (1999). Real-Time Extensions to UML. *http://www.telelogic.com/help/search/index.cfm*.

[Telelogic AB, 2003] Telelogic AB (2003). Home page. *http://www.telelogic.com*.

[Tensilica Inc., 2003] Tensilica Inc. (2003). Home page. *http://www.tensilica.com*.

[Tewari, 2001] Tewari, A. (2001). *Modern Control Design with MATLAB and SIMULINK*. John Wiley and Sons Ltd.

[The Dobelle Institute, 2003] The Dobelle Institute (2003). Home page. *http://www.dobelle.com*.

[The SUIF group, 2003] The SUIF group (2003). SUIF compiler system. *http://suif.stanford.edu*.

[Thiébaut, 1995] Thiébaut, D. (1995). Parallel programming in C for the transputer. *http://maven.smith.edu/~thiebaut/transputer/descript.html*.

[Thoen and Catthoor, 2000] Thoen, F. and Catthoor, F. (2000). *Modelling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*. Kluwer Academic Publishers.

[Thomas and Moorby, 1991] Thomas, D. E. and Moorby, P. (1991). The verilog hardware description language. *Kluwer Academic Publishers*.

[TimeSys Inc., 2003] TimeSys Inc. (2003). Home page. *http://www.timesys.com*.

[Tiwari et al., 1994] Tiwari, V., Malik, S., and Wolfe, A. (1994). Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. On VLSI Systems*, pages 437–445.

[Transmeta, 2005] Transmeta, I. (2005). Support: Technical documentations. *http://www.transmeta.com/developers/techdocs.html*.

[Vaandrager, 1998] Vaandrager, F. (1998). Lectures on embedded systems. *in Rozenberg, Vaandrager (eds), LNCS, Vol. 1494*.

[Vahid, 1995] Vahid, F. (1995). Procedure exlining. *Int. Symp. on System Synthesis (ISSS)*, pages 84–89.

[Vahid, 2002] Vahid, F. (2002). *Embedded System Design*. John Wiley& Sons.

[van de Wiel, 2002] van de Wiel, R. (2002). The code compaction bibliography. *This source has become unavailable, check http://www.iro.umontreal.ca/~latendre/codeCompression/codeCompression/node1.html instead*.

[Vladimirescu, 1987] Vladimirescu, A. (1987). SPICE user's guide. *Northwest Laboratory for Integrated Systems, Seattle*.

[Vogels and Gielen, 2003] Vogels, M. and Gielen, G. (2003). Figure of merit based selection of A/D converters. *Design, Automation and Test in Europe (DATE)*, pages 1190–1191.

[Wagner and Leupers, 2002] Wagner, J. and Leupers, R. (2002). Advanced code generation for network processors with bit packet addressing. *Workshop on Network Processors (NP1)*.

[Wedde and Lind, 1998] Wedde, H. and Lind, J. (1998). Integration of task scheduling and file services in the safety-critical system MELODY. *EUROMICRO '98 Workshop on Real-Time Systems, IEEE Computer Society Press*, page 18pp.

[Wegener, 2000] Wegener, I. (2000). *Branching programs and binary decision diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications.

[Weiser, 2003] Weiser, M. (2003). Ubiquitous computing. *http://www.ubiq.com/hypertext/weiser/UbiHome.html*.

[Weste et al., 2000] Weste, N. H. H., Eshraghian, K., Michael, S., Michael, J. S., and Smith, J. S. (2000). *Principles of CMOS VLSI Design: A Systems Perspective*. Addision-Wesley.

[Willems et al., 1997] Willems, M., Bürsgens, V., Keding, H., Grötker, T., and Meyr, H. (1997). System level fixed-point design based on an interpolative approach. *Design Automation Conference (DAC)*, pages 293–298.

[Wilton and Jouppi, 1996] Wilton, S. and Jouppi, N. (1996). CACTI: An enhanced access and cycle time model. *Int. Journal on Solid State Circuits*, 31(5):677–688.

[Wind River Systems, 2003] Wind River Systems (2003). Web pages. *http://www.windriver.com*.

[Winkler, 2002] Winkler, J. (2002). The CHILL homepage. *http://www1.informatik.uni-jena.de/languages/chill/chill.htm*.

[Wolf, 2001] Wolf, W. (2001). *Computers as Components*. Morgan Kaufmann Publishers.

[Wolsey, 1998] Wolsey, L. (1998). *Integer Programming*. Jon Wiley & Sons.

[Wong et al., 2001] Wong, C., Marchal, P., Yang, P., Prayati, A., Catthoor, F., Lauwereins, R., Verkest, D., and Man, H. D. (2001). Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. *9th Intern. Symp. on Hardware/Software Codesign (CODES)*, pages 170–177.

[Xue, 2000] Xue, J. (2000). *Loop tiling for parallelism*. Kluwer Academic Publishers.

[Young, 1982] Young, S. (1982). *Real Time Languages –design and development–*. Ellis Horwood.

About the Author

Peter Marwedel

Peter Marwedel was born in Hamburg, Germany. He received his PhD in Physics from the University of Kiel, Germany, in 1974. From 1974 to 1989, he was a faculty member of the Institute for Computer Science and Applied Mathematics at the same University. He has been a professor at the University of Dortmund, Germany, since 1989. He is heading the embedded systems group at the computer science department and is also chairing ICD e.V., a local company specializing in technology transfer. Through ICD, the knowledge about compilers for embedded processors is incorporated into commercial products and is made available to external customers. Dr. Marwedel was a visiting professor of the University of Paderborn in 1985/1986 and of the University of California at Irvine in 1995. He served as Dean of the Computer Science Department from 1992 to 1995. Dr. Marwedel has been active in making the DATE conference successful and in initiating the SCOPES series of workshops. He started to work on high-level synthesis in 1975 (in the context of the MIMOLA project) and focused on the synthesis of very long instruction word (VLIW) machines. Later, he added compilation for embedded processors (with emphasis on retargetability) to his scope. His projects also include synthesis of self-test programs for processors. His work comprises codesign, energy-aware compilation. Recent work on multimedia-based training led to the design of the RaVi multimedia units (see //ls12-www.cs.uni-dortmund.de/ravi).

Dr. Marwedel is a member of ACM, IEEE CS, and Gesellschaft für Informatik (GI).

He is married and has two daughters and a son. His hobbies include skiing, model railways and photography.

E-mail: peter.marwedel@udo.edu

Web-site: http://ls12-www.cs.uni-dortmund.de/∼marwedel

List of Figures

Index