Microprocessor Architectures
from VLIW to TTA

Henk Corporaal
Delft University of Technology
The Netherlands

JOHN WILEY & SONS
Chichester • New York • Weinheim • Brisbane • • Singapore • Toronto

Published by John Wiley & Sons Ltd,
Baffins Lane, Chichester,
West Sussex PO19 1UD, England

National 01243 779777
International (+44) 1243 779777

e-mail (for orders and customer service enquires): cs-books@wiley.co.uk

Visit our Home Page on http://www.wiley.co.uk or http://www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons, Inc, 605 Third Avenue,
New York, NY 10158-0012, USA

VCH Verlagsgesellschaft mbH, Pappelallee 3,
D-69469 Weinheim, Germany

Jacaranda Wiley Ltd, 33 Park Road, Milton,
Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01,
Jin Xing Distripark, Singapore 129809

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0 471 97157 X

Produced from camera-ready copy supplied by the author
Printed and bound in Great Britain by Bookcraft (Bath) Ltd
This book is printed on acid-free paper responsibly manufactured from sustainable forestation, for which at least two trees are planted for each one used for paper production.

Contents

Preface

A computer architect is faced with answering the question what functionality has to be supported by a computer system, and in particular by the processor of such a system: is hardware support needed for complex functionality, or for basic (simple) operations only. In the latter case more complex functionality is handled by one or more layers of software, like the operating system, special libraries, and high level languages.

Historically this question has been answered differently. In the seventies there was a strong tendency to support as much functionality as possible, leading to the well-known complex instruction set computers (CISCs). In the eighties computer architects became more aware of taking the whole mapping trajectory from applications to hardware into account, while performing a quantitative analysis of cost and performance. This resulted in reduced instruction set computers (RISCs), which pipelined the execution of instructions. Their simple design soon allowed them to be put on a single die, giving them an extra performance edge.

The nineties again show a tendency towards more complex processors. Superscalar processors, which are now commonly used in PCs and workstations, become very complex because of their support for issuing multiple instructions per cycle and speculative execution. Especially the decoding logic of those superscalars using out of order execution, such as the Pentium Pro, the MIPS R12000, PowerPC 620, HP PA-8500, UltraSPARC-II, and the Alpha 21264, is getting very complex and may require multiple stages of the instruction pipeline. These processors are therefore expensive and less suitable for low cost embedded systems.

Very long instruction word processors (VLIWs) circumvent the complex decoding logic and its associated cost, at the price of not being binary code compatible with predecessor architectures. VLIWs have better scaling properties. In addition they are more flexible; their functionality can easily be adapted to specific applications. Therefore they are good candidates for high performance low cost embedded systems. Several interesting and very powerful VLIWs hit the (multi-media) market: the TriMedia of Philips, the Mpact of Chromatic and the TMS320C6x of Texas Instruments. Despite their good properties, the data path of VLIWs is still too complex, in particular when they are scaled to high performance. This makes it interesting to look at alternative architectures which avoid this complexity but keep the good properties of VLIWs.

Aims of the book

The complexity of existing architectures, and therefore their long design and implementation trajectory, was one of the major issues which inspired us to start a new architecture project at the Delft University of Technology, called the *MOVE project* (see URL: *http://cs.et.tudelft.nl/MOVE*). Within this project we proposed and researched a different class of architectures, called *Transport Triggered Architectures* (TTAs). This class combines a number of interesting properties, like functional flexibility, high performance scalability, and design modularity, which make these architectures particularly suited for being applied within embedded systems.

The purpose of this book is to demonstrate the feasibility of TTAs. It shows that TTAs have a substantial lower complexity than equal performance traditional architectures. This complexity reduction can be achieved by changing run-time decisions into compile-time decisions. By giving the compiler control about the actual data transports which occur within a processor, and scheduling these transports at compile-time, much hardware can be avoided.

An example TTA will be described together with its implementation and realization. The class of TTAs is very large. Many architecture design parameters of this class and their evaluation will be treated extensively. Furthermore, it is shown how to make these architectures interruptible; this makes them suitable for hard real-time applications as well.

Although the principles underlying TTAs are equally valid for general purpose as application specific computing, we believe that TTAs will first be introduced in the latter computing domain, primarily, because in that domain short design times and low cost are very important, and binary code compatibility is of less concern, especially when good compilers are available. Actually, a number of embedded systems has already been realized using TTA based processors. To shorten the design trajectory of these processors, while still enabling a quantitative search of a large design space, tools are needed which explore this space automatically for any given application. At the end, the book touches upon this subject, and explains how we automatically synthesize processors using the *MOVE framework*, a set of design automation tools based on a parametric TTA template.

Audience

This book is meant for all people interested in computer architecture beyond the introductory level; especially for those who are interested in new trends in computer architecture. In order to enhance readability a large introductory part has been included containing an extensive description of the current trends in computer architecture, and treating the complexity of instruction level parallel processors. The book may therefore be used in post-graduate curricula of computer science and electronic engineering, where new directions in computer architecture, and corresponding code generation techniques, are treated.

The book may also interest people involved with research in embedded system design. ASIC designers will be interested, because the transport triggering concept offers an attractive alternative to dedicated, non-programmable, hardware solutions. It also enables very short design times through the automatic design space exploration and generation of hardware and software for arbitrary applications written in a high level language.

Outline

This book has been organized into thirteen chapters and five appendices; a short description of each of them is given below. For convenience, the book is completed with a glossary of symbols and abbreviations, and an index. Each of the chapters and appendices starts with a short introduction and overview and often ends with a summary or evaluation of discussed subjects. To enhance readability, the book is divided into four parts.

Part I: Architectures: overview and complexity

The first part consists of chapters 1 to 4; it serves an introductory purpose and motivates the research into different processor architectures.

Chapter 1 describes the criteria which have to be met when designing processors, in particular when designing processors for specific applications or application domains. It serves as a general motivation for using Transport Triggered Architectures (TTAs) and also gives a quick introduction to these architectures.

Chapter 2 describes general trends in computer architecture. It highlights the main developments which are responsible for the annual performance increase of more than 50% of current microprocessors. At the architectural level several techniques are outlined to speedup processors. From these techniques we derive a classification model which is based on a four-dimensional architecture design space. Besides architectural developments, this chapter also goes into the compiler developments. It is shown that an important design decision to be made by a computer architect is to decide which part of the required functionality should be supported at the architectural level, and must therefore be implemented in hardware, and which part is better handled in software by an optimizing compiler.

Chapter 3 gives an introduction on the complexity of buses. One of the main problems of instruction level parallel processors, which exploit parallelism at the instruction level, is that they require many buses or ports on the register file. In this chapter models are discussed for area and timing characteristics of buses and multi-ported register files. From these models several fundamental area and delay problems are derived. Solving these problems necessitates the reduction of the number of buses and ports.

Chapter 4 analyzes the complexity of instruction level parallel processors. A criterion is introduced to measure the complexity of the data path of these processors. This criterion is applied to VLIWs, whether or not superpipelined, with a shared register file. It is demonstrated that the complexity of the data path of these processors gets out of control when these processors are scaled up, such that they offer high degrees of parallelism.

Part II: Transport triggering concept

In the second part, containing chapters 5 and 6, the operating principles of TTAs are shown and discussed, and an example realization is treated.

Chapter 5 derives the concept of TTAs, taking VLIWs as point of departure. It is shown that traditional VLIWs include hardware functionality which should and can be controlled by the compiler in order to reduce their complexity. This compiler control requires that all data transports become visible at the architectural level. This results in a TTA for which the complexity of both the register file(s) and of the bypassing network can be substantially reduced, thereby alleviating the main bottlenecks of highly parallel VLIWs. As a side effect many software and hardware advantages emerge which are described at the end of this chapter.

Chapter 6 treats an instantiation of a TTA, called *MOVE32INT*, which has been realized at our research laboratory, without using a completely automated design trajectory. The processor has been implemented in a 1.6 μm CMOS Sea-of-Gates technology (i.e. with prefabricated transistors or gates) and designed to run at 80 *MHz*; it contains multiple 32-bit integer type function units. Four transports (and therefore up to four FU operations) can be issued per cycle. The purpose of this chapter is to show how the TTA concept can be realized into a complete processor. *MOVE32INT* will act as a reference processor to which we will refer several times in the following chapters when describing TTA alternatives.

Part III: The design space of transport triggered architectures

Transport triggering is a concept for designing architectures; however, many design options are left open. The resulting large design space of TTAs is described in chapters 7 to 10; many variants are researched and evaluated.

Chapter 7 describes the transport design space of TTAs. The principle type of operation of a TTA is the data transport, or *move*. Different programming models for programming these moves are researched. This includes several options for performing conditional execution. Further, the execution of these moves has to be pipelined in order to achieve a high performance. This results in several transport pipelining models which are also covered, together with their required control.

Chapter 8 presents the design of function units (FUs). In particular it researches and evaluates different methods of controlling the latching of pipeline stages of pipelined FUs. It will be shown that these methods have a large impact on the architectural behavior of TTAs, and therefore also on the compilation techniques employed for TTAs. Two latching methods, hybrid latching and virtual-time latching, turn out to have the most promising characteristics.

Chapter 9 goes into the design of register units (RUs): the units which contain the general purpose register files. It will be shown that high performance TTAs require pipelined RUs.

The pipelining model used for RUs has some impact on the transport pipelining model, as well. Unfortunately, pipelining RUs results in a similar bypass complexity problem, although at a much reduced scale, as the one TTAs are trying to solve. Methods are discussed to alleviate this problem. Finally, this chapter demonstrates that TTAs can be designed with arbitrary connectivity, which encompasses all connectivity schemes shown in existing VLIWs.

Chapter 10 treats the difficult subject of how to support exceptions. Because the main application of TTAs will be for specific applications, exception support should be optional and have less impact on the achievable cycle time. Different exception support mechanisms are proposed; a newly developed method for handling exceptions within TTAs, called *busflashing*, turns out to be of particular interest.

Part IV: Architecture evaluation and synthesis

The final part, containing chapters 11 to 13 quantifies the capabilities of TTAs, demonstrates their automatic synthesis, and concludes this work.

Chapter 11 demonstrates how well TTAs perform with respect to the described problems of traditional VLIWs by evaluating results of code generation for numerous benchmarks. It will be shown that indeed TTAs require fewer register ports, and that many other TTA specific features result in enhanced performance. This chapter further evaluates several architectural issues; while chapters 7 to 10 performed a qualitative analysis of many TTA design variations, this chapter does a quantitative analysis, based on code measurements, of those variations which proved to be most interesting.

Chapter 12 explains how TTAs can be synthesized automatically. It describes a set of tools, called the *MOVE framework*, which enable the designer to do an automatic search of large parts of the TTA design space. This search is based on a parametric TTA. Using these tools the chapter demonstrates that the inter-FU and RU connectivity can be substantially reduced without any performance loss. This is one of the key aspects of TTAs.

Chapter 13 provides a summary of this work, lists the main contributions, and suggests lines of continuation for future research.

Appendices

In addition to above chapters five appendices have been added which contain more detailed information about a number of discussed topics.

Appendix A gives a derivation of an area model of multi-ported memory cells, based on an implementation of such a cell in a Sea-of-Gates image.

Appendix B details the control logic used to control the pipelining used in *MOVE32INT*. This control logic largely determined the achieved cycle time of 12.5 ns.

Appendix C gives an in depth analysis of hybrid pipelined FUs. Both the scheduling restrictions and the implementation of the hybrid pipelining control logic are treated. A number of control optimizations are developed in order to release the impact of this control logic on the critical timing path.

Appendix D presents an analysis of virtual-time latched FUs. These FUs are shown to have more scheduling constraints than hybrid pipelined ones, but they constrain much less the achievable cycle time of a processor employing these type of FUs.

Appendix E describes related work. In particular it looks at other approaches of reducing the high demands on the register file and the high complexity of the interconnection network between FUs.

Reading order

This book can be read in different orders. Although the best way to read it is to go through all the chapters in sequence, some readers may find it convenient to read first those chapters which have their specific interest. For those readers the strong and weak dependences between chapters and appendices are given in the figure below; it shows for example that in order to understand the architectural evaluation given in chapter 11 it is advisable to read and understand at least chapters 2, 4, and 5 first.

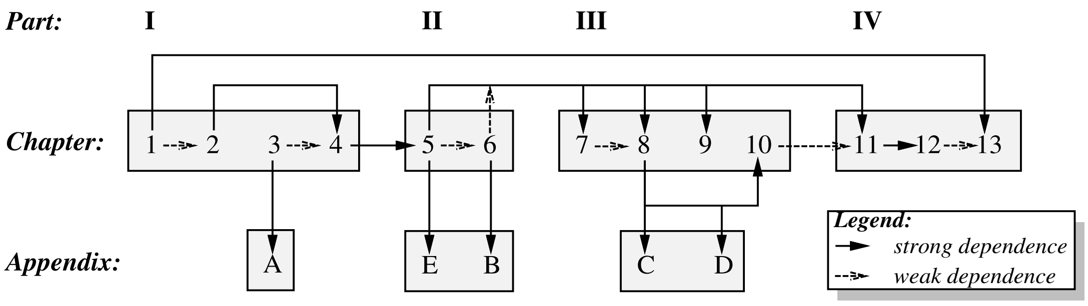


Figure: Dependences between chapters and appendices.

Chapter 3 can be skipped by readers less interested in bus design and bus complexity details; its major goal is to stress the importance and criticalness of buses within (instruction level) parallel processors. Chapter 6, which describes an example implementation of a transport triggered prototype processor, can be read on its own; although it builds upon knowledge from earlier chapters, an attempt has been made to make it self contained, offering readers an opportunity to quickly learn about architectural, implementation and realization aspects of TTAs.

Occasionally footnotes are added to the text; they usually elaborate on some specific detail, or give some more precise understanding of discussed matters. Footnotes may be skipped by less experienced readers without loosing the main thread.

Part I

Architectures: overview and complexity

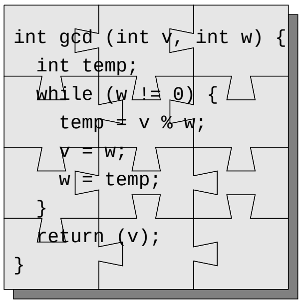
The first part provides background information on computer architecture in general and on instruction level parallel architectures in particular. It also serves as a motivation of why it is necessary to explore different architectures which do not suffer from the complexity problems of current instruction level parallel architectures.

Chapter 1 gives a short overview of requirements relevant to processor design, especially when tailoring processors for specific applications. It indicates that traditional architectures have trouble in meeting all these requirements simultaneously; our proposed solution, the use of transport triggered architectures, is briefly outlined.

Chapter 2 explores current trends in computer architecture, including developments in the compiler and VLSI area. As a side effect of this exploration an architecture classification scheme is presented.

Chapters 3 and 4 treat complexity matters. Chapter 3 describes several models for timing and area of buses. These matters are important because bus design has a severe impact on achievable cycle time and area of current processors. Chapter 4 models the data path of instruction parallel processors, thereby demonstrating its large complexity when supporting larger amounts of parallelism.

In passing, some terminology, needed to understand following chapters, is defined in part one. In particular knowledge of chapters 2 and 4 is recommended to understand and fully appreciate chapter 5 and following.



From specification to realization

Chapter contents

1 Problem statement

It is hard to imagine how our current world and way of living would have been if single chip microprocessors were not invented. The microprocessor market is huge. The use of powerful microprocessors is not limited to general purpose computers. Although there are millions of them sold annually, billions of microprocessors show up each year in other equipment, used for all kinds of daily life purposes; this may range from a domestic iron to high definition television sets. These processors are called embedded because they are embedded within special purpose systems. The main applications for these processors are within the control and signal processing domains.

Processors have been designed for about half a century. Within that period their performance has seen a tremendous increase. This increase is driven by an ongoing demand to run larger and more powerful applications. The performance of a processor, when running an application, is determined by several aspects: the quality of the compiler, the architecture of the processor, its implementation, and its realization. All these aspects have seen large improvements. Although all these aspects will have some coverage in this book, its main focus is on one of these aspects: the processor architecture.

The *architecture of a processor* can be viewed as its *functional appearance* to its immediate users [10, 19, 59]. This book proposes a new class of architectures, called *transport triggered architectures*, or TTAs, whose functional appearance is quite different from most architectures designed so far. TTAs have a number of favorable characteristics which make them attractive for being applied as processors used within embedded systems.

The purpose of this chapter is to indicate the requirements and problems encountered when designing processor architectures, to explain the driving-force behind the development of TTAs, and to give the reader a quick introduction into the subject of this book. It is structured as follows. Section 1.1 describes different, and often conflicting, processor design goals. From these goals a number of requirements is derived which architectures should fulfill. Section 1.2 shows that these requirements cannot be met simultaneously by current architectures; a problem exists and a new class of architectures is therefore needed. Finally, section 1.3 presents the proposed solution; it explains the main operating principles of TTAs and indicates that TTAs are more likely to satisfy the requirements.

1.1 Processor design goals

By far the most important market for processors is the embedded system market. Annually, billions of microprocessors are sold, which are almost exclusively used within embedded systems. The design of such a system is a complex process with many degrees of freedom. When designing a processor for an embedded system one may have several optimizations in mind:

1. *Cost optimization.* Factors influencing cost are divided into recurrent and non-recurrent costs. Recurrent costs are influenced by: chip area, package and number of external connections, and the used process technology; using a well established process technology is cheaper and gives a higher yield. The design time and the production of layout masks, if a full custom realization is used, largely determines the non-recurrent costs.
2. *Power optimization.* Low power consumption will have a high priority for battery powered equipment. Usage of this type of equipment will assume enormous proportions.
3. *Application optimization.* One may design dedicated processors which are highly customized for a given application, or one may try to design a more general applicable processor which can handle a larger range of applications. It will be clear that a dedicated processor may achieve more performance, and have lower cost.
4. *Performance optimization.* If this is the main goal then the processor must give the highest performance for a specific application, while costs and power constraints are less important.

The above optimization requirements are often contradictory. It is up to marketing to target, given the technological and architectural possibilities, the right market spot, i.e. the right combination of design time, VLSI technology, design method (e.g. the use of full custom chips, Sea-of-Gates or programmable gate arrays), performance, area, application domain, etc.

Instead of designing an *application specific processor* (ASP), specially designed for an embedded system, one could buy an existing *general purpose processor* (GPP), which is designed to execute many different applications with reasonable performance, or an available *digital signal processor* (DSP), whose design is tuned towards signal processing applications. These processors provide a cost-effective solution for applications with typical (i.e., not too divergent) computational requirements. For example, a standard micro-controller type processor is well suited for managing the user interface (buttons and display) of video equipment. On the other hand, applications or application domains with requirements that do not map efficiently onto standard components call for ASPs.

Let us make a comparison of the characteristics of these three processor classes. Table 1.1 shows those characteristics which are related to above mentioned optimizations. The non-recurrent costs of GPPs and DSPs hardly exist, because they can be taken off-the-shelf; for ASPs they can be very high. Recurrent costs may be low for ASPs, if the market volume is large enough, because they include only that functionality which is required by the appli-

Characteristics	Processor class		
	GPP	**DSP**	**ASP**
Costs:			
• Non-recurrent	zero	zero	very high
• Recurrent	high	medium	low
Power consumption	high	medium	low
Functionality	high	medium	low
Performance	medium	high	very high

Table 1.1 Comparing characteristics of different processor classes.

cation. This also holds for the embedded system itself, because an ASP is often cheaper to interface with the surrounding logic. Power consumption of high performance GPPs can be very high; ASPs may consume less power because they do not include superfluous functionality. GPPs score much better on functionality; they are more general applicable. Also, their programming environment is usually much better than for DSPs and ASPs. Finally, ASPs may include special functionality which gives them a performance edge for those applications for which they are designed. Note that table 1.1 only gives a first impression of the characteristics; individual processors within each class may deviate.

The previous discussion suggests that ASPs are interesting in several aspects. However, doing a dedicated design for some application comes at a cost. Using a traditional design trajectory takes significant investment and effort. This may induce the following ASP design problems:

- Long time-to-market, and therefore use of outdated technology. The long design trajectory causes ASP performance, as compared to available GPPs and DSPs, to be far lower than expected, because the latter currently see an annual performance improvement of more than 50% (see section 2.2).
- High non-recurrent engineering costs: no reuse of developed hardware and software.
- Only limited design space exploration possible, and therefore limited opportunity for applying the discussed optimizations.
- High design risk: the design may never achieve correctness or the required level of performance.
- Tendency towards limited programmability due to design constraints. This leads to difficulty with running complete applications besides the performance-critical routines, and inability to adapt to changes of algorithms or application requirements.

To overcome these problems we would like to define a class of architectures from which ASPs can be automatically generated. This class should cover a large range of applications with very different performance and functionality demands. This class should therefore satisfy the following five requirements:

1. *Scalability*, such that the performance of the processor can be increased to high performance levels without changing the architecture concept underlying the processor.
2. *Programmability*, to handle large and possibly irregular applications and become immune towards algorithmic changes. Note that during the design and life-time of an ASP the program(s) it has to execute may change quite a bit.
3. *Flexibility*, which means that the functionality of the processor can be easily adapted to changing application and market requirements.
4. *Modularity*, making automatic synthesis of many different instantiations tractable. Building blocks have to be designed and implemented once, but can be reused many times.
5. *Software compatibility*, allowing the reuse of software. Software compatibility can be achieved either at the architectural level, or at the high level language (HLL) level. The former is almost impossible when designing ASPs for completely different applications. However, it should be possible to compile HLL code, without first having to write a new compiler.

In the remainder we will concentrate on the first four requirements. The fifth requirement can be met by having a *retargetable compiler*; such a compiler can be designed if the modularity requirement has been fulfilled [35].

1.2 Problem analysis

In this section we discuss possible architectures which potentially satisfy the scalability, programmability, flexibility, and modularity requirements, as stated in the former section.

Scalability

The computer architect has several techniques at his disposal to increase the processor performance (chapter 2 discusses several of those techniques). One of the most important techniques, which is currently generally applied, is to add more, concurrently operating, (FUs). FUs are responsible for the execution of the operations; for example one may have separate FUs for integer operations, floating point operations, and for loads and stores. Two main classes of architectures use this technique: *superscalars* and *very long instruction word* architectures (VLIWs). The general organization of these architectures is drawn in figure 1.1; the central processing unit (CPU) shown contains five FUs of which one is dedicated to perform load and store operations on the data memory. Instructions are read by the instruction fetch unit from the instruction memory, decoded and send to the FUs. The FUs communicate data via a shared register file.

In order to keep the FUs busy the parallelism contained in a program has to be detected and exploited. Superscalars use hardware to detect this parallelism at run-time; the hardware tries each clock cycle to fetch, decode and execute as many successive instructions as possible while satisfying the dependences between these instructions. VLIWs on the other hand rely

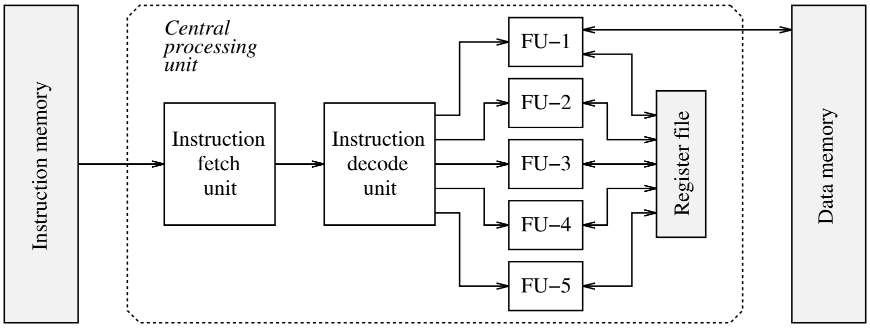


Figure 1.1 General organization of superscalar and VLIW architectures.

on compile-time detection of parallelism; its compiler packs multiple operations into each instruction, and guarantees their independence. This has consequences for scalability. Superscalars require much hardware for the run-time detection of parallelism, especially when many FUs are added. VLIWs do not need that hardware and are therefore much more scalable. However, as will be explained in chapter 4, even VLIWs run into problems when many FUs are added: they need too much data transport capacity (i.e. a large number of buses) for mutual FU traffic and for transports between FUs and the register file(s). In the worst case situation this high capacity may be needed, however, in practice bus utilization is much lower. This suggest that there are opportunities for improvement.

Programmability

Usually ASPs have a limited programmability; they only support a restricted set of operations. Full programmability requires that besides an instruction fetch unit the regular set of RISC (reduced instruction set computers) operations is supported, such that it is possible to compile HLL programs for these processors; ASPs usually lack this functionality. A standard way to solve this problem and make an embedded system both programmable, such that it can run large compiled HLL programs, and application specific, such that it can run critical routines fast enough, is to use a combination of a GPP and an ASP; the ASP acts as a coprocessor of the GPP. This way the programming environment of the GPP can be used. However, it means that we need an extra component, and that the interface of the ASP should be adapted to the GPP coprocessor interface. This makes the system more complex and expensive. Also, as we will discuss below, the communication overhead between an ASP and GPP may reduce the overall performance. Therefore, a better choice is to use a so-called ASIP, an *application specific instruction set processor*. An ASIP is an ASP which includes the minimal functionality needed to run any HLL program.

Flexibility

ASPs have to be adapted to the requirements of applications. This not only means that it should be easy to scale the architecture by adding FUs which support regular, RISC like operations, but also, that support for more complex operations can be added as well. In the past, the usual way to support complex functionality was to add a tightly coupled coprocessor to an existing GPP [92]; this also guarantees programmability, as described above. However this approach has three disadvantages:

1. Communication of instructions and/or data is required between the GPP and the coprocessor.
2. The coprocessor is used only to speedup some critical routine; it is not used for the other, more serial code.
3. Functionality has to be duplicated. For example, a coprocessor may require addressing and load-store functionality, which is already available in the GPP, for performance purposes.

A far better solution is to use an ASIP and add *special function units* (SFUs) to it. This solution does not suffer from the communication overhead because SFUs have, in contrast to coprocessors, access to the same register set as regular FUs. SFUs can also, although to a lesser extent, be used for serial code parts. For example, an experiment using the MCCD (minimum cost contour detection) algorithm, which draws contours within a graphical image (see [35]), showed that a specific routine could be accelerated with a factor of two by adding a five input adder. Occasionally this adder might also be used within the other code to do regular, two input, additions.

It may also occur that, as soon as we have the capability of adding FUs (because of the scalability requirement), it is more efficient, and easier, to add more regular FUs instead of an SFU, especially when the ASP has to support an application domain instead of just one application; the reasons are well known to people designing processors according to 'RISC design principles' [67, 130] (see also chapter 2 and section 4.2). In this case the serial code part profits even more from these extra FUs.

To gain some insight, the performance of a processor with an extra SFU is modeled and compared to a GPP with a coprocessor. The coprocessor approach is discussed first. Assume a fraction f_{par} of a program, called the parallel part, is down-loaded to and executed by the coprocessor; the execution time of this fraction decreases with a speedup factor S_{par}. The other fraction $1 - f_{par}$, the serial part, is executed on the GPP. The execution time for a processor with coprocessor configuration is then estimated by:

$$t_{coproc} = (1 - f_{par}) \times t_{proc} + \frac{f_{par} \times t_{proc}}{S_{par}} + t_{comm} \tag{1.1}$$

where t_{proc} is the execution time of the program on the same GPP without a coprocessor, and t_{comm} is the communication time required between processor and coprocessor. The latter is modeled to be proportional to execution of the number of instructions within the down-loaded code part on the processor. It is given by:

$$t_{comm} = rct \times f_{par} \times t_{proc} \tag{1.2}$$

where rct (the relative communication time) is a proportionality constant.

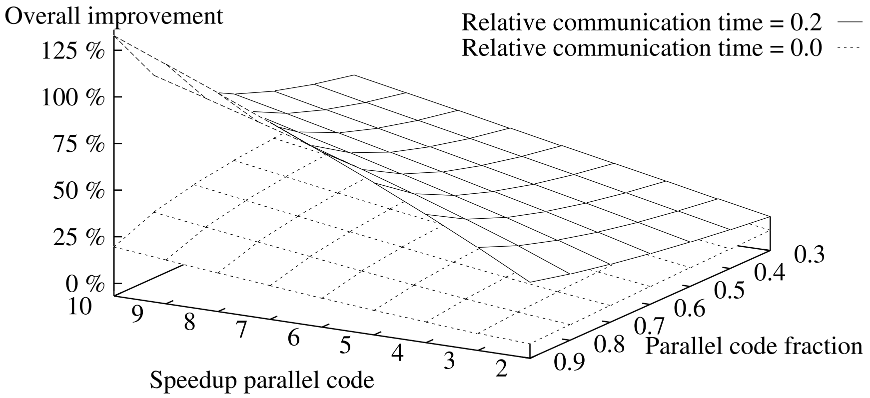


Figure 1.2 Performance improvement of a processor with SFU compared to one with a coprocessor.

If an SFU is incorporated within the processor we assume again the same speedup of the parallel fraction. In this case the serial part also shows a little speedup S_{serial}. There is no communication overhead. The execution time is then given by:

$$t_{SFU} = \frac{(1 - f_{par}) \times t_{proc}}{S_{serial}} + \frac{f_{par} \times t_{proc}}{S_{par}} \tag{1.3}$$

In figure 1.2 the performance improvement when using an SFU instead of a coprocessor is drawn as a function of S_{par} and f_{par} for two rct values: one with zero communication overhead, $rct = 0$, the other with $rct = 0.2$. The improvement of the serial code fraction, in case a SFU is used, is pessimistically taken to be 1/20 of the improvement of the parallel code fraction, i.e. $S_{serial} = 1 + (S_{par} - 1)/20$. The figure shows that, given the assumptions, an SFU easily outperforms a coprocessor, even when we neglect the coprocessor communication overhead.

Modularity

Modularity requires that the generated processors are built out of a restricted number of building blocks. These building blocks are pre-designed and efficiently implemented. Apart from SFUs, which have to be specially designed, any required ASP can be designed by combining more or fewer of these building blocks.

VLIWs are (again) the best candidate for satisfying the modularity requirement; e.g., FUs can be added to a VLIW without change of their architecture concept. However, as already mentioned in the scalability paragraph above, their required transport capacity does not scale well when adding FUs; in fact it is directly related to the number of FUs. This means that buses for data transports and FUs cannot be added independently.

In case the FUs share the same register file, the problem even gets worse: adding an FU means a complete redesign of the register file, because it needs more ports. Furthermore, the addition of an FU may add a field to the instruction format as well, and therefore change the instruction cache and/or the external instruction data path. For the same reason, the addition of an SFU, which is only occasionally (within one routine) used, may lead to sparse instruction encoding. These problems can be solved by adding crossbars between FUs and register files, as done in the TriMedia processor [134], however, this is a very costly solution.

Conclusion
From the above discussion it becomes clear that a new class of architectures is needed from which ASIPs can be automatically synthesized. This class should inherit the favorable characteristics of VLIWs without having their disadvantages. This book proposes such a class, the class of TTAs. This class has been extensively researched within the *MOVE project* at the Delft University of Technology.

1.3 Proposed solution

In this section we discuss the operating principle of TTAs and argue that TTAs may meet the requirements stated in section 1.1. TTAs are organized as a set of FUs and *register units* (RUs), which are connected by some kind of interconnection network. RUs contain general purpose registers. Figure 1.3 displays a picture of this organization. It shows a TTA containing three FUs, of which one communicates with the data memory, and two RUs. Note, as becomes clear from this figure, that there are no prior constraints on the number of inputs and outputs of FUs and the number of ports on the RUs, and on the capacity of the transport network; e.g., the latter may contain only one bus, or be a very complex network. Thus the modularity and flexibility requirements seem to be fulfilled.

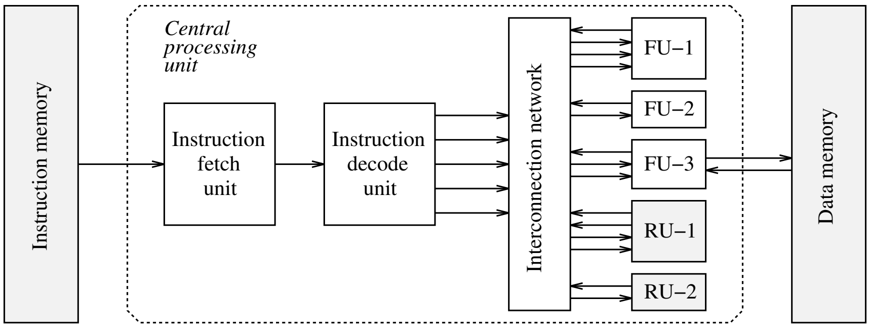


Figure 1.3 Organization of a TTA.

How is this architecture programmed? For most current architectures a program has to specify the FU operations, like `add`, `sub` and `mul`, which have to be executed. A VLIW, for example, may specify several of these FU operations within each instruction. In TTAs these FU operations occur as a side effect of data transports; that is why the architecture is called *transport triggered*. A TTA program only specifies the data transports to be performed by the interconnection network; so it supports only one type of operation: the *move* operation from source to destination; a different name for TTA is therefore *MOVE architecture*, and their realizations are also called *MOVE processors*. A TTA instruction consists of one or more move operations; the number of move operations per instruction is equal to the number of transports which can be handled in parallel by the interconnection network. Sources and destinations of move operations are any register contained in the RUs, and any FU input and output. For example, to perform an addition, just specify two moves to transport the source operands to the FU which supports the `add` operation, and, within one of the following instructions, specify a move which reads the result of the addition, by reading from the output of the same FU, and transports this result to another desired destination (e.g., the input of another FU). Example 1.1 explains the operation of a small TTA program and illustrates several TTA specific code optimizations.

Example 1.1 *TTA programming example.*

Consider the addition of a constant to a vector: $\vec{a} := \vec{a} + 25$. This vector addition can be translated to the following loop for a RISC like architecture (assuming that vector $\vec{a}$ contains 100 4-byte integer elements starting at address zero):

```
   ld   r2,396        // load address of a[99] into r2
L: ld   r1,(r2)       // load a[i] into r1
   add  r1,r1,25      // add constant
   st   r1,(r2)       // store a[i]
   sub  r2,r2,4       // decrement address register
   bgez r2,L          // branch if address greater or equal zero
```

For a particular TTA, supporting at least three data transports (moves) per cycle, the compiler may produce the following code:

```
   396 -> r2                    // load address of a[99] into r2
L: r2  -> Old                   // start load operation
   Rld -> O1add ; 25 -> O2add   // move load result to adder
   Radd-> O1st  ; r2 -> O2st    // perform store of added result
   r2  -> O1sub ; 4  -> O2sub   // decrement address register
   Rsub-> O2bgez; L  -> O1bgez; Rsub-> r2   // repeat if r2 >= 0
```

Destination locations start with an `O` if they are FU operands, or with `O1` and `O2` when the FU requires two operands. Source locations start with an `R` if they contain FU results. We observe that RISC operations are coded as one or more moves. The example code shows 6 instructions with 11 moves; on average we need about two moves per RISC operation. Further, register `r1` is not used anymore; the result of the load operation is directly forwarded to the adder, without being put in one of the register units. The same holds for the result of the addition. If all operations are performed on separate FUs a smart compiler may also recognize that some moves are ‘loop invariant’ and be put in the loop prologue; this gives the following code:

```
   25  -> O2add ; 4 -> O2sub   // prepare constant operands
   396 -> r2    ; L -> O2bgez  // load start address and label L
L: r2  -> Old                  // start load operation
   Rld -> O1add                // move load result to adder
   Radd-> O1st  ; r2-> O2st    // perform store of added result
   r2  -> O1sub                // decrement address register
   Rsub-> O1bgez; Rsub-> r2    // repeat if r2 >= 0
```

In this case the loop body only requires seven transports. It is even possible (for a very smart compiler) to get rid of the use of register `r2`; try yourself to see what the code should look like (the loop body then contains six transports).

The idea of programming only transports is not new; already one of the first commercial computers, the DEUCE, which became operational in 1955, operated that way [70]. The DEUCE was a commercial version of the ACE, a prototype computer designed in the early fifties [3, 161][1]. After that time it seems that this idea has been reinvented several times, see for example: [13, 20, 87, 106, 115, 140]. However, these approaches were restricted in several ways. They all fix the interconnection network to either one external memory bus (which is of course relatively slow[2]), or to two internal buses. Apparently, no one got the idea that the transport capacity is an independent scalable parameter. Further, several of these approaches do not allow direct communication between FUs; they miss therefore one of the key properties of TTAs, namely the ability to *bypass* values directly from FU to FU without consulting the RUs. An architecture also related to TTAs is the FPS-164 [150]. This architectures allows operations to read directly from FU outputs, and therefore support software bypassing. Since this architecture used an irregular instruction format and so-called push-pull pipelines for most function units (i.e. results have to be pushed explicitly out of the FU pipeline), code generation turned out to be very difficult.

The TTA programming method may seem clumsy when seen the first time: instead of one (dyadic) FU operation we now have to specify three move operations. However, as will be shown in this book, this approach has many advantages. You may already guess that, because all transports can now be controlled by the programmer or compiler, unnecessary transports, which occur quite frequently in current architectures, can be avoided; example 1.1 demonstrated already several of these optimizations. This is the key to solving the scalability problem of VLIWs.

As shown, the operating principle of TTAs is very simple. In this respect the development of TTAs can be viewed as the next step in the evolution which went from CISCs (Complex instruction set computers) to RISCs; RISCs streamlined their instruction set such that it could be pipelined. As a result a compiler for RISCs needs to put more effort into producing efficient code. TTAs are even more simpler than RISCs, and their compilers correspondingly more complex. A whole set of TTA specific compiler optimizations emerges, and

[1]This prototype seems still to be within the British National Science Museum.

[2]It means that each transport requires a load of the value of a source location into an internal register, and a store of that register value into the desired destination location.

has to be exploited. For a thorough discussion of TTA compilation techniques the reader is referred to [71].

We conclude that TTAs seem to be modular, flexible and scalable. The final requirement of programmability can be met by demanding that a TTA should contain a fully fledged instruction fetch unit, and that the FUs support a minimal set of operations (the RISC like operations), including conditional operations and support for immediates. Together, these properties make TTAs particularly interesting for embedded system design. Within the MOVE project a set of tools has been developed, called the *MOVE framework*, which allows for the automatic generation of embedded TTA based processors given an arbitrary application written in high level programming language.

From the above discussion it will be clear that the main focus of this book will be on the following topics:

- Determination of the transport bottlenecks of current instruction level parallel processors.
- Derivation of the TTA concept by elimination of these transport complexity bottlenecks.
- The design of a fully programmable TTA. The TTA concept opens up a very large design space of largely different TTAs. This design space is researched and evaluated.
- Demonstrate that TTAs outperform current architectures and that they can easily be synthesized and realized.

Each of the four parts of this book is dedicated to one of these topics.