Hades

—

Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor

Diss. ETH No. 12276

Hades

—

Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Stefan Hans-Melchior Ludwig
Dipl. Informatik-Ing. ETH
born May 21, 1966
citizen of Schiers, Graubünden

accepted on the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. H. Eberle, co-examiner

1997

Für Irene, Vanessa und Cyril.
Für meine Eltern.

Acknowledgments

I would like to express my appreciation and gratitude to my advisor Prof. Niklaus Wirth. His striving for simplicity and understandability are unparalleled. If Hades is fast, it is because of the constant "fear" of spending a cycle too much here or a byte too much there. He is a fabulous teacher and it was a pleasure to work under his supervision.

I thank Prof. Hans Eberle for being my co-examiner. His knowledge in hardware design was very welcomed during the development of the Hades hardware and his constant skepticism of the feasibility of reconfigurable coprocessors was a driving force behind this work.

Many thanks go to my colleague, collaborator and office mate for the past four years, Stephan Gehring, for his excellent Trianus framework, for the discussions and feedback, for the criticism and for his willingness to enhance or alter Trianus almost instantly. It was fun to work with you!

Thanks to Immo Noack for many things, especially for being who you are.

Thanks and acknowledgments go to

- all of my colleagues at the Institute for Computer Systems for a stimulating and enjoyable working environment. Remember, there is only one true "Ludwig-of-the-day".
- Erwin Oertli for tips regarding the details of the Hades board and many other things.
- Marco Sanvido for the provision of a timing analyzer.
- Beat Heeb for helping with the Ceres-2 and Cuno Pfister for having produced CALLAS, a good tool to measure my own work against.
- Wolfgang Weck and Clemens Szyperski for discussions on various issues.
- Tom Kean for the chip and much more.
- Bill Wilkie for answering my endless questions and for never stopping to appreciate my corrections to the data sheet.
- the remaining people at Xilinx Development Corp., Scotland, for their openness and generosity.
- Virtual Computer Corporation for distributing Hades with their board.
- Dr. Roger Woods and Jean-Paul Heron of the DSP Laboratory, Queen's University of Belfast, for their willingness to work with Hades and for their humor.
- Patrick Müller and Reto Zimmermann for betting a term project on Hades, and for succeeding.
- Chuck Thacker, Dave Conroy and Mark Shand of the Digital Systems Research Center for discussions about FPGAs and high-performance memory systems.
- Monty Brekke and Steve Atkins for the Verilog code and Martin Radetzki for the VHDL code.

- Peter Alfke for background information on the XC4000 and the FPGA market.
- David Hofmann for the logic minimizer.
- my proofreaders Stephan Gehring, Taylor Hutt, Cheryl Lins and Nels Vander Zanden. I owe you a dinner in Silicon Valley.

This work would have been impossible without the love and support of my beloved wife, Irene. Thank you for everything during the past nine years. I love you!

Contents

List of Figures

List of Tables

List of Programs

Kurzfassung

Das Aufkommen von benutzerprogrammierbarer Hardware entfachte ein Interesse an konfigurierbaren Koprozessoren, welche zur Beschleunigung von zeitkritischen Softwareteilen benutzt werden können, indem diese in Hardware gegossen werden. Applikationen, welche auf konfigurierbaren Koprozessoren ausgeführt werden, werden mittels Hardwarebeschreibungssprachen oder schematischen Eingabesystemen beschrieben. Diese Beschreibungen werden in Logikgatter übersetzt, für welche ein Layout (Auslegeplan) gefunden werden muss. Logik- und Layoutsynthese sind zeitintensive Vorgänge, für welche heutige Hardwaresynthesewerkzeuge bis zu vier Grössenordnungen mehr Zeit benötigen, als Compiler zur Übersetzung von Softwarebeschreibungen. Heutige Synthesewerkzeuge benutzen stochastische Algorithmen, um ihre Resultate zu erzielen und das Wissen des Benutzers über ein Design kann nur schwer in den Entwicklungszyklus eingebracht werden.

In dieser Dissertation wurde ein komplettes Hardwarebeschreibungssystem entwickelt. Es besteht aus einem konfigurierbaren Koprozessor und entsprechender Layoutsynthesesoftware.

Der konfigurierbare Koprozessor von Hades besteht aus einem einzelnen XC6216 FPGA und lokalem Speicher in der Form von statischem RAM. Der Koprozessor ist mittels einer Speicherkartenschnittstelle mit einer Arbeitsstation verbunden.

Die Hades-Software besteht aus einem Layoutsynthese-Backend für die XC6200 Architektur. Als Frontend zu unserer Software dient Trianus, ein Gerüst zur Entwicklung von FPGA Designs. Die Hardwarebeschreibungssprache Lola dient zur Beschreibung der Algorithmen.

Die Hades Software besteht aus

- einem Technologie-Mapper,
- einem deterministischen und konstruktiven Plazieralgorithmus, der sich auf Plaziervorgaben des Benutzers verlässt, um dichte Layouts zu erzielen,
- einem Labyrinth-basierten Router, der durch den Benutzer in verschiedener Weise beeinflusst werden kann,
- einem Generator von Konfigurationsinformation, und
- einem Schnittstellengenerator, welcher zu einer Hardwareapplikation automatisch eine Softwareschnittstelle generiert.

Das resultierende System hat auf heutigen Rechnern sehr schnelle Übersetzungszeiten im Bereich von Sekunden. Die Hades Software ist mindestens eine Grössenordnung schneller als die vom Hersteller erhältlichen Werkzeuge für dieselbe FPGA Architektur. Die schnellen Übersetzungszeiten eröffnen eine neue Art der interaktiven Entwicklung von Hardware und erlauben es, auf wirksame Weise das Wissen des Entwicklers in den Entwicklungszyklus einzubringen.

Abstract

The advent of Field-Programmable Gate Arrays has spurred an interest in building reconfigurable coprocessors, which are used to accelerate the time-intensive parts of software by casting them into programmable hardware. Applications running on reconfigurable coprocessors are developed using hardware description languages or schematic capture systems. These descriptions are translated into logic gates, for which a layout has to be found. Logic and layout synthesis is a time-consuming process and turnaround times of traditional hardware synthesis tools are up to four orders of magnitude longer than those of software compilers. Current synthesis tools rely on stochastic algorithms to achieve their results and the user's knowledge about a design can enter the design cycle only with difficulty.

In the course of this thesis, a complete hardware description system has been developed. It consists of a reconfigurable coprocessor based on the Xilinx XC6200 FPGA architecture and corresponding layout synthesis tools.

The Hades reconfigurable coprocessor consists of a single XC6216 FPGA and local memory in the form of static RAM. The coprocessor is attached via a memory-mapped interface to a workstation.

The Hades software is composed of a layout synthesis back-end for the XC6200 architecture. The front-end to our tools is Trianus, a framework for FPGA design. The hardware description language Lola is used to describe the algorithms. The Hades software is composed of

- a technology mapper,
- a deterministic and constructive placement algorithm, which relies on placement hints given by the user to achieve dense layouts,
- a maze-running router, which can be influenced by the user in various ways,
- a configuration bitstream generator, and
- an interface generator, which generates a software interface to a hardware application automatically.

The resulting system achieves very fast turnaround times for layout synthesis on the order of seconds on contemporary hardware. The Hades software is at least an order of magnitude faster than commercially available tools for the same FPGA architecture. The fast turnaround times open up a new way for interactively designing hardware and effectively bring the designer's knowledge into the design cycle.

1 Introduction

Hades — Greek god, brother of Zeus, lord of the underworld, ruler of the dead, god of wealth.

Ever since the conception of the first mechanical calculator (Abacus) several thousand years ago, mankind has striven to speed up the brain straining task of calculating with numbers. Since the introduction of the digital computer in the late 1930s and early 1940s, the speed at which computations can be executed has increased by six orders of magnitude (10^3 additions/s in 1946 vs. 10^9 in 1996). Likewise, power consumption and cost have decreased dramatically.

Gordon Moore's "law" was stated in 1962 and is still valid today. It is not a law, but a prediction saying that the number of transistors on a chip doubles every 18 months. Corollaries to this prediction are that the speed of a circuit doubles every 18 months or that the same performance can be bought for half the price after 18 months. As an example, the Intel 8080 microprocessor introduced in 1975 consisted of 4,500 transistors. The Pentium Pro introduced by the same vendor in 1995 contains 5.5 million transistors.

As a consequence of this development, we are able to buy a computer today (early 1997) clocked at 200 MHz with a 32-bit wide data path, executing half a billion instructions per second, with 32 MB of main memory and 2 GB of disk space for $3,000 and put it on our desktop. The same machine would have been termed a supercomputer only two decades ago.

This dramatic increase in computational power can be attributed to several factors:

- First and foremost, technological advances in circuit manufacturing pushed clock speed and circuit density to levels not imaginable in the beginning of digital computing. Chips with a feature size of 0.35 μm running at 500 MHz are common nowadays and the next shrink down to 0.25 μm is within sight.
- Improvements in the architecture of processors, such as multiple execution units and pipelining helped to lower the ratio of clock cycles per instruction.
- The availability of fast and cheap memory allowed to incorporate cache memory on processors (first and second level caches), larger main memory and caches on disk controllers. The availability of larger main memory allowed a space for time tradeoff, resulting in faster execution of algorithms.
- Improvements in algorithms and data structures reduced the runtime of operations.

Still, this level of performance is barely sufficient for modern applications such as image and sound processing (compression/decompression), three-dimensional graphics rendering, speech recognition and so forth. As computers become faster, new application domains are tackled requiring more computational power and programmers pay less attention to the efficient implementation of algorithms. Therefore, new ways for speeding up algorithms are still a strong driving force in hardware and software research.

1.1 Coprocessors

In a computer system, there is normally one central processing unit (CPU) which can be programmed to execute a task. Computationally intensive tasks are usually accelerated through algorithmic and programming techniques, such as caching, loop unrolling and implementation in assembly language. Often though, these techniques do not achieve the required speedup. If a large enough user base is interested in solving such a task quickly and there is enough economic interest to justify the investment, *special purpose hardware* is built. This hardware solution of a task can consist of a board full of chips or of a single chip. The latter is called an Application Specific Integrated Circuit (ASIC). The production of ASICs is expensive and time consuming, but the resulting chips solve the task much more quickly than a general purpose CPU.

A *coprocessor* is an ASIC, which aids the CPU by speeding up a task and usually requires the presence of the former in a system to function properly. Figure 1.1 shows a schematic view of such a system. The coprocessor is either closely coupled to the CPU with a dedicated interface or it resides on an extension card attached to the system bus.

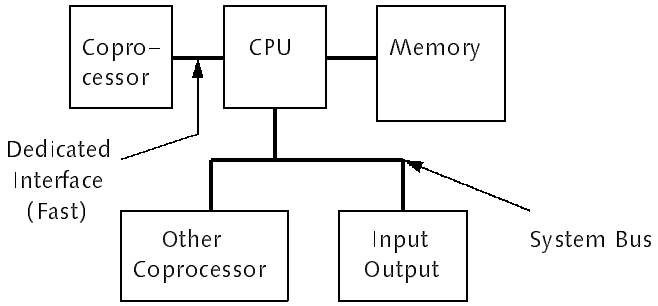

Figure 1.1: CPU and Coprocessor

Well-known examples of early coprocessors are Floating-Point Units (FPU), such as the Intel 8087 and the Motorola 68881. In the past years, as chip area became larger and cheaper, FPUs were integrated into the CPUs resulting in even better performance.

Coprocessors more common today are digital signal processors (DSPs), video decoder chips (for MPEG-encoded video streams) and so-called "multimedia" processors (TriMedia [Phi95], MPact [Chr95]) processing *long data streams*, such as continuous video or audio data. However, as with FPUs, a coming trend in recent months was the introduction of *single instruction multiple data instruction sets* in general purpose CPUs (also called "multimedia" instructions). Examples are Intel's MMX for the Pentium, Sun's VIS for the SPARC and Digital Equipment's media extensions for the Alpha. Typically, these instructions operate on 8 bytes or 4 half-words at a time. For an Add instruction, for instance, the carry path is broken up after every 8 bits. This way, 8 parallel adds can be executed with one instruction. By incorporating some crucial instructions for DSP applications into the CPU, it is possible to achieve the same processing throughput on a general purpose CPU as with a special ASIC.

It seems that as soon as there is a large enough user base, CPU manufacturers will include traditional coprocessor-like abilities in a CPU. The result is often much higher speed than with a conventional coprocessor, as the coupling between processing units is tighter and more parallelism between instruction pipelines can be exploited. The result is that it becomes increasingly more difficult to find applications for which it makes sense to build special purpose hardware.

1.2 User-Configurable Hardware

One consequence of the high transistor count on an integrated circuit is the ability to produce *configurable hardware.* In configurable hardware, some transistors are "wasted" to implement configuration memory and routing switches instead of active logic circuitry. This memory is used to control connections between various transistors on the chip. The immediate advantage of configurable hardware is the ability to implement *different functions* in a chip at very low cost. It is not necessary any more to implement a digital circuit using a silicon process. Instead, configurable hardware brings the silicon foundry to the desktop.

Traditionally, the term configurable hardware is associated with Programmable Logic Devices (PLDs) such as Programmable Array Logic (PAL). PALs consist of a programmable And-matrix and a fixed Or-matrix. One great advantage of PALs are *predictable signal delays*. The configuration memory of these devices is usually implemented by erasable ROM cells. They can be programmed many times, but one often needs special programming machinery and the number of reprogramming steps is limited. Only recently have in-system programmable PALs been introduced [Lat96].

In 1985, Xilinx Inc. introduced a new class of programmable logic devices, the first commercial *Field-Programmable Gate Array* (FPGA). As the name suggests, FPGAs were conceived as a replacement for Mask-Programmable Gate Arrays (MPGAs). The configuration memory is implemented by static RAM (SRAM) cells. Therefore, these devices can be programmed easily, rapidly (in milliseconds), and an arbitrary number of times. No special programming machinery is required. By downloading a *configuration bitstream* to such a device, it is possible to adapt the device to a specific task in a couple of milliseconds. Typically, the contents of the SRAM remain unchanged during an application session. Chapter 2 gives a more detailed description of this technology.

1.3 Reconfigurable Coprocessors

The main drawback of special purpose hardware as defined in Section 1.1 is that it is *special purpose*. Once built, it is not possible to change the hardware to accommodate slightly different needs. A CPU can be programmed to implement any algorithm, but an ASIC implementing an MPEG decoder will do only that. Another drawback is that for economic reasons it is not sensible to build special purpose hardware for an algorithm that is executed only by one user. For example, if a time consuming task is not "popular" enough to warrant the high design and manufacturing costs of an ASIC, the user who has to solve that task has no other options than optimizing the software code or buying a faster machine, both of which might not suffice to achieve the needed level of performance.

With the advent of FPGA technology, however, it is possible to construct a *general purpose coprocessor*, which can be programmed for a specific task and then reprogrammed for another task within milliseconds. Such a system is usually called a *Custom Computing Machine* (CCM) but we prefer the name *Reconfigurable Coprocessor* (RC), as it better characterizes the close coupling to a CPU.

When compared with the use of hard-wired ASICs, the inclusion of a reconfigurable coprocessor in a computer system bears several advantages:

- The time-consuming part of an algorithm can be executed at the speed of hardware.
- The design implementing this part of an algorithm can be developed with the flexibility and turnaround time of software.
- The available FPGA hardware can be reused for various algorithms, thereby reducing circuitry that is otherwise unused in a system.

- The hardware can be adapted to changing requirements or new algorithms, as only the SRAM configuration data has to be generated anew.
- When specific parameters to an algorithm are known, the hardware can be specialized for those, thereby reducing the amount of logic (due to constant propagation) and achieving higher speeds.
- The cost and power consumption of a system is reduced, as one reconfigurable coprocessor can fulfill tasks of several separate ASICs, provided that the tasks are separated in time.

There are two main application areas where RCs can be used to speed up an application: algorithms performing *integer operations on large amounts of data* (most DSP applications fall into this area) and the *processing of input and/or output data streams.* To perform well in the first area, the RC should reside as closely as possible to the CPU, where it can execute, for instance, a time-critical loop of some algorithm. To perform well in the second area, the RC should be near the input/output ports or include these ports on the board.

The general architecture of an RC (shown in Figure 1.2) consists of one or more *FPGAs connected to local memory* and an interface to the CPU. As will be seen in Chapter 6, the speed of this interface is essential for achieving good performance. Often, a general purpose connector where extension boards can be mounted is included as well. The connection between the system bus and the local memory of the RC shown in Figure 1.2 is not necessarily present in an RC. It is, however, very convenient to have, as the CPU can then directly access the board's local memory and does not have to go through the FPGA, which might need to be reconfigured to allow for that possibility.

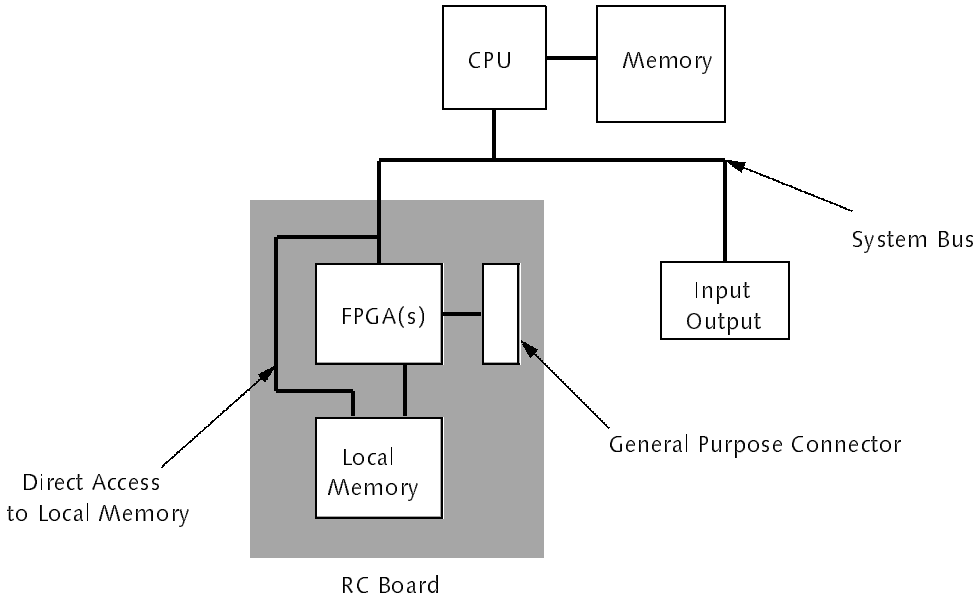

Figure 1.2: Typical Reconfigurable Coprocessor

In the past, most RCs have been realized as big external boards attached to the system bus via an additional interface [ACC95, ABD92, BRV89, Ber93, GHK90]. But as FPGAs get denser, modern RCs are realized as smaller extension boards [LSC96, Sha96], which can be plugged directly into the system bus (such as a PCI bus [PCI93]). Chapter 7 presents some of the RCs that have been developed in the past. The bigger boards described therein achieve or surpass performance levels normally attributed to supercomputers.

Programming an RC is difficult, even more so than programming a CPU. As the execution unit of an RC is one or several FPGAs, *programming an RC means designing hardware*. There are efforts under way to allow for the generation of FPGA designs directly from a "high-level" programming language such as C [AS93, Gal95, IS95], but by and large, the main method for describing an RC application is by means of schematic entry or hardware description languages. This might be one reason why RCs are still not very popular, as few people are trained in hardware design, and as hardware synthesis software is usually big, expensive and often slow.

1.4 Hardware Synthesis

Hardware is usually described using a combination of the following three means:

- Schematic entry
- Hardware description language (HDL)
- Circuit layout

Still many hardware design engineers use schematic entry to describe a hardware circuit. But over the last years, hardware description languages (HDLs) have gained ground and are becoming more popular. For utmost performance and density, using a layout editor is still the preferred way for describing a circuit. Note that any combination of these three methods can be used to describe a complete system. Also, a subcircuit described in the form of a layout might be represented by an HDL description on a higher level.

Figure 1.3 describes the flow from a schema or HDL program to the final circuit, which in this case is implemented using an FPGA. The dark shaded areas represent tools and hardware which are the subject of this thesis.

A *compiler* translates an HDL program or a schema into a device independent *netlist* representing the circuit. The netlist is mapped to the target device by a *technology mapper*. Technology mapping is a hard problem for FPGAs with complex cells (cf. Chapter 2) and takes some time to decide which gates in a netlist are mapped into which cell in an FPGA. The resulting netlist is then *placed and routed*, i.e., the physical location on the FPGA for the gates in a netlist is determined (placement) and the gates are connected together with wires (routing). Both problems are NP-complete - their runtime cannot be bounded by a polynomial function, i.e. they require time exponential in the problem size - and require long run times to achieve good results. Commercial tools often use placement algorithms based on simulated annealing [KGV83]. This algorithm tries many different configurations until it reaches a solution. Once the design is placed and routed, the netlist can be converted into an SRAM configuration for the FPGA and finally be downloaded to the device.

Netlists that cannot be placed or routed, or that do not meet the timing constraints may lead to changes in the design. Therefore, a feedback path as shown in Figure 1.3 exists from several design phases to several other phases and often several iterations are necessary to reach a satisfactory result.

Note in Figure 1.3 that the netlists between different stages may not be in the same format, as they may be produced by software from different vendors. Also, most often, the output from one phase is *written to a file and read in again* in the next phase, leading to inefficiencies. These and other aspects of hardware synthesis software are discussed in Chapter 3.

1.5 Contributions

This thesis deals with the problem of speeding up computationally intensive tasks by means of *specialized hardware*. The goal is the development of a hardware description system, which

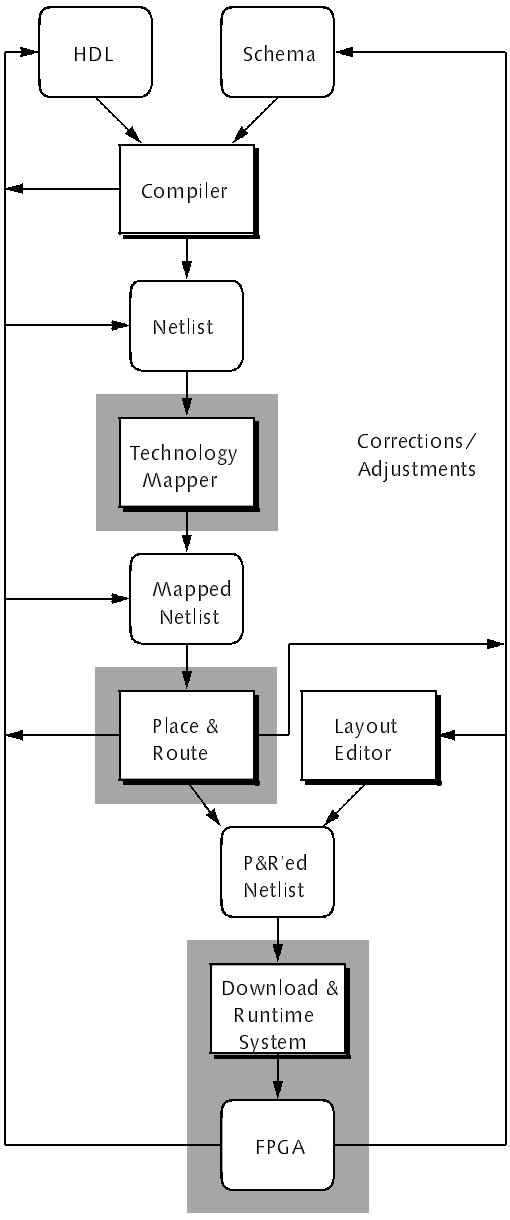

Figure 1.3: Hardware Synthesis Flow

runs on machines with moderate performance, allows for the construction of reconfigurable coprocessor applications and supports a *very fast design cycle*. The resulting system, called *Hades* (HArdware DEscription System), makes use of Field-Programmable Gate Array technology to implement a *reconfigurable coprocessor*. It features *fast, interactive physical design tools* to define and construct this hardware, and a *software interface*, which allows software programmers to make use of the available hardware accelerator.

1.6 Overview of Thesis

The following Chapter 2 introduces the technology of FPGAs, focusing on the target architecture of this thesis, the Xilinx XC6200. Chapter 3 presents the foundations of our *Hades* system, namely the hardware description language *Lola* and the *Trianus* framework for FPGA design. Chapter 4 introduces the *Hades hardware* — a coprocessor board based on the XC6200 FPGA. Chapter 5 introduces the *Hades software*, featuring a *technology mapper*, *automatic place and route tools*, a *loader* and *driver* for the coprocessor board and a *software interface* presenting a coprocessor application to the software programmer as an accelerated library module. Chapter 6 shows the usefulness of Hades by presenting applications of the board and the software and compares Hades to the commercial tool for the XC6200 FPGA. Chapter 7 presents related work. Chapter 8 summarizes the presented work, draws some conclusions and presents ideas and suggestions for future work.

2 Field-Programmable Gate Arrays

In this chapter, we present the motivation for using field-programmable gate arrays and describe various architectures. In particular, we present the Xilinx XC6200 FPGA and the reasons it was chosen as the target architecture for this thesis.

2.1 Background

FPGAs continue the trend of higher integration and more flexibility in programmable logic devices. Whereas PALs and complex PLDs are used for the replacement of glue logic on a printed circuit board and the implementation of decoders and simple state machines, FPGAs were invented as an alternative to mask-programmable gate arrays. These are mainly used for the implementation of high-volume, complex logic chips in an electronic device, for which no standard off-the-shelf solution exists.

A mask-programmable gate array is, as the name indicates, an array of gates with fixed functionality, such as NAND-gates, where part of the routing network, typically the last metal layer in the silicon process, is determined by the designer. This last layer of metal is manufactured by the gate array vendor in a fabrication facility. The problem with this approach is that the manufacturing process usually takes several days or weeks and that it is quite expensive. Errors in the design lead to a faulty chip and hence to a substantial increase in costs. It is also not easily possible to explore different design alternatives, except using slow simulation.

FPGAs try to alleviate these problems by making the *gates* and the *routing network programmable*. The designer of a circuit can program the functionality of the chip "in the field", by simply *downloading configuration bits* onto the FPGA. FPGAs have a clear advantage over MPGAs, as it is possible to create a new gate array within a few seconds, and — with SRAM- and EEPROM-based FPGAs — to do so an unlimited number of times. Programmability comes at a cost, however. The logic implemented in an FPGA is less dense (usually 10% of an MPGA) and also slower (2-10 times) than its gate array implementation [DeH96]. This is mainly due to the large amount of wiring resources needed (up to 85%) and the on-chip configuration store (up to 10%), leaving only a small fraction of the chip area for active circuitry (as little as 5%).

The feature of reprogrammability makes FPGAs useful in several areas:

- Replacement of glue logic: This continues the trend of other PLDs, such as PALs and CPLDs.
- Replacement of MPGAs: This leads to a reduction of costs, as a design can be produced in shorter time and design changes can be accommodated even when the circuit is already installed in a system.
- Speed of design cycle: The FPGA can be programmed in the system and tested right away, without lengthy simulation cycles.
- Reconfigurable coprocessor: FPGAs can be programmed to implement parts of a time-consuming algorithm in hardware.

- Logic emulation: Instead of simulating a netlist, FPGAs can directly implement a circuit.
- Teaching: Students can implement complex circuits in a short time, verify them using real hardware, and, upon completion, delete the design and move on to the next assignment [GLW94, Wir95, Wir96b].

In 1995, the total size of the programmable logic market was 1.7 billion US Dollars of which the FPGA market was 716 million US Dollars. Several different FPGA architectures from many vendors compete in that market. The leader is Xilinx with its 2000, 3000, 4000, 5200 and 6200 architectures. Other vendors include Actel (MAX), Altera (FLEX), Atmel/Concurrent Logic (6000), AT&T/Lucent (Orca), Lattice (ispLSI), Motorola/Pilkington (MPA) and Quicklogic (pASIC), all of them American.

2.2 General Structure

A field-programmable gate array consists of *programmable logic cells* containing function units and registers, a *programmable routing network*, and *programmable input and output (I/O) cells*. The routing network connects logic cells with each other and with the I/O cells. Figure 2.1 gives an overview of an FPGA.

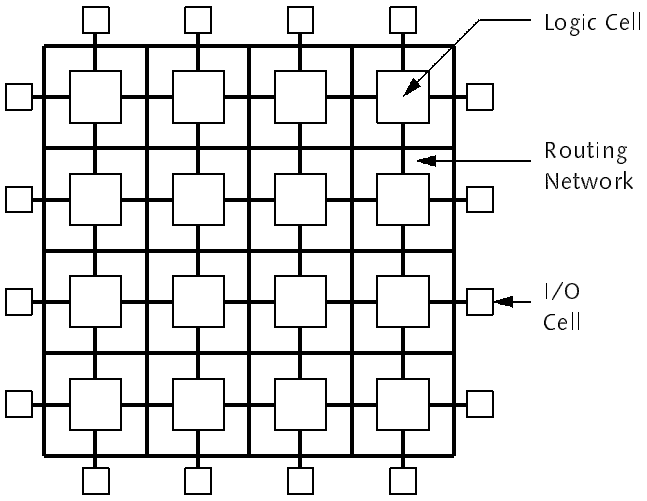

Figure 2.1: General FPGA Structure

In many FPGAs, programmability is achieved through SRAM cells, which are intermixed on the chip with the logic cells and the routing network. SRAM-based FPGAs can be implemented using a standard CMOS process. Conceptually, the SRAM cell layer lies beneath the logic and routing layers, controlling switches in the latter (Figure 2.3). The switches, which are implemented as pass gates using n-type transistors, determine the functionality of the cells and the direction of the signal flow (cf. Figure 2.2). The SRAM cells are implemented as 5 or 6 transistor cells.

Because of the volatility of the SRAM cells, the configuration bits are loaded from an external storage device upon power up, typically from a serial ROM. A serial or parallel programming interface is provided for that task. Protecting intellectual property in the presence of a ROM is problematic, as the ROM can be read out not only by the FPGA, but also by using a probe. Therefore, FPGA vendors try to protect the intellectual property of their customers by keeping the format of the configuration bitstream secret.

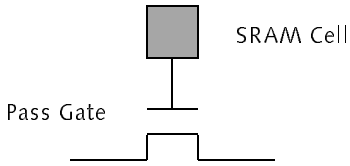

Figure 2.2: Pass Gate

SRAM-based FPGAs can be programmed an unlimited number of times. Depending on the size of the chip and the speed of the programming interface, the time needed for a full reconfiguration is between hundreds of microseconds and hundreds of milliseconds.

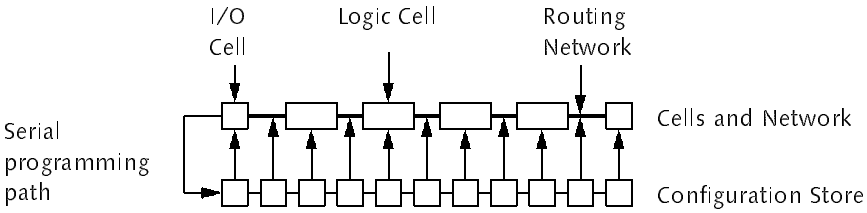

Figure 2.3: Configuration Store

Other technologies for storing configuration information and implementing switches are based on *antifuses* [Act95, Qui94] and EEPROM cells [Alt96]. In this thesis we focus on FPGAs based on SRAM cells, as they are the only alternative for implementing a reconfigurable coprocessor. Further information on configuration store technology can be found in [BFR92].

Among different FPGAs, there is a wide architectural variety in the functionality of the logic cells and the structure of the routing network. If a logic cell provides only *simple functionality* (e.g. any function of two inputs) then it is called a *fine-grained* architecture. If it provides *rich functionality* (e.g. any function of four or more inputs and one or several optional registers), then it is called a *coarse-grained* architecture. More complex logic cells often also require a more complex routing network.

2.3 The Xilinx XC6200

As our first example of a commercial FPGA, we examine the XC6200 architecture from Xilinx [Xil96], the successor of the CAL architecture from Algotronix [Alg90, Kea89], which is described in Section 2.4. It is an SRAM-based FPGA with an array of identical, fine-grained cells and a hierarchical routing network. Two novel features of the architecture are a *processor interface* and very *fast reconfiguration times*, which makes the chip suitable for coprocessor applications. The first implementation of the architecture, the XC6216, consists of an array of 64 by 64 cells.

2.3.1 Logic Cell

An XC6200 logic cell implements *any logic function of two inputs* or a *multiplexer*, possibly followed by a *register.* As shown in Figure 2.4, the *Dynamic Mux* to the right of *Y2* is the only multiplexer that is controlled by a dynamic signal *(X1)*. All other multiplexers are controlled by SRAM configuration bits, denoted by shaded boxes attached to the bottom. The signals on *X1*, *X2* and *X3* are determined by multiplexers selecting from eight possible inputs: any of the

four neighboring cell outputs or any of the four length-4 FastLANE signals running along the cell (see Section 2.3.2).

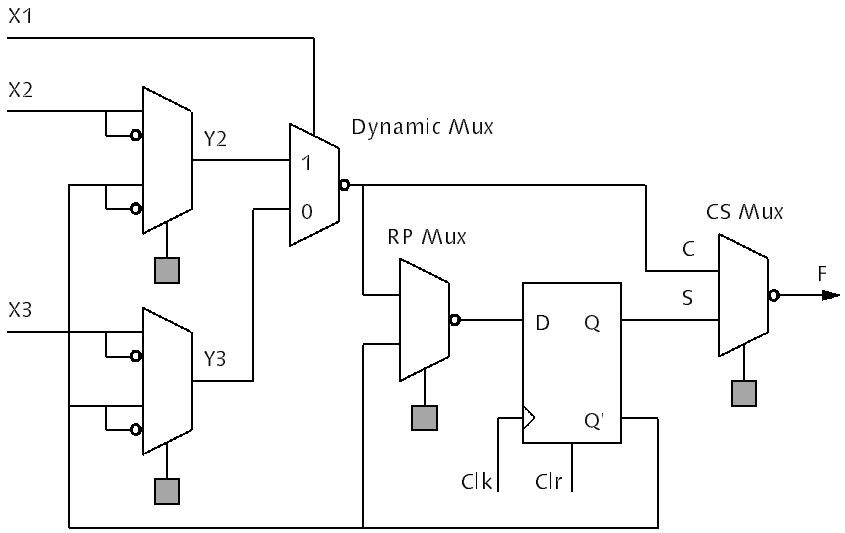

Figure 2.4: XC6200 Function Unit

The feedback from *Q'* to the *Y2* and *Y3* multiplexers in Figure 2.4 gives additional flexibility, for instance to implement the register of a counter ($F := REG(F \oplus cin)$). The *RP Mux* can be used to *protect* the register. When a register is protected, it is only writable through the processor interface of the FPGA (Section 2.3.4). This can be useful for implementing constants or for writing parameter values into a circuit without having to connect the register to an I/O pad (also called padless IO).

To achieve higher transmission speeds, the multiplexers in Figure 2.4 have *inverted outputs*. These can be a problem for the CAD software. Not only do these inversions exist in the multiplexers within the cell, but also on routing multiplexers. A process called *inversion compensation* (cf. Section 5.7) is required to determine the polarity of input signals to a cell. The reader should keep the presence of these inversions in mind when reading subsequent chapters.

All Boolean functions of two or less inputs can be implemented by a single multiplexer with optional inversions at its inputs. Figure 5.4 in Chapter 5 shows all cell functions supported by Hades. As an example, we show the implementation of an AND- and an XOR-gate; other functions can be implemented accordingly. The true value of the function is taken at point *C* in Figure 2.4, hence the inversion on the output of the central multiplexer must be taken into account.

To implement $F := a \wedge b$, we connect *a* to *X1* and *X3* and *b* to *X2*. *Y2* and *Y3* select the inverted value of *X2* and *X3*, respectively. $F := a \wedge \overline{b}$ can be implemented by selecting the true value of *X2* at *Y2*. To implement $F := a \oplus b$, we connect *a* to *X1* and *b* to *X2* and *X3*, inverting *X3* at *Y3*.

Figure 2.5 shows the resulting circuit and Table 2.1 shows the truth table of the functions. We use the Lola notation for describing a multiplexer (Chapter 3), where *MUX(sel: in0, in1)* indicates that when *sel* is zero, the multiplexer selects *in0*, and when it is one, it selects *in1*.

2.3.2 Routing Network

The routing network of the XC6200 consists of *connections between neighboring cells* and a *hierarchy of longer connections* between switches located at 4 and 16 cell boundaries and at

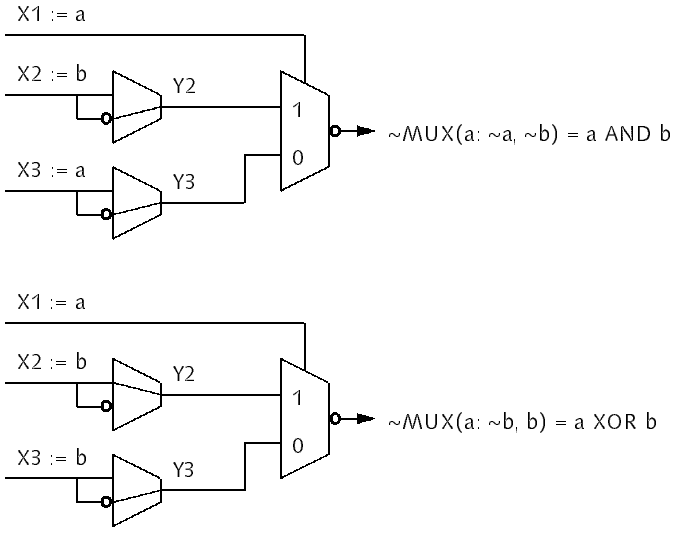

Figure 2.5: Mux Implementation of the AND and XOR Functions

a	b	a AND b	˜MUX(a: ˜a, ˜b)	a XOR b	˜MUX(a: ˜b, b)
0	0	0	a = 0	0	b = 0
0	1	0	a = 0	1	b = 1
1	0	0	b = 0	1	˜b = 1
1	1	1	b = 1	0	˜b = 0

Table 2.1: Truth Table for AND and XOR Functions

the array's perimeter. The longer connections are termed *FastLANE* connections. This hierarchy of routing resources provides shorter signal delays, as the delays scale logarithmically with the distance between cells, instead of linearly. All signals are uni-directional, i.e. they have one source only. Tri-state buses must be emulated by using multiplexers.

On the first level of the routing hierarchy are neighbor connections (Figure 2.6). The signal source of a neighbor connection can be either the *F* output of the function unit or one of the three inputs from the other cells. For instance, the *north* output of a cell is *MUX(sel0, sel1: F, SIn, EIn, WIn)*. Hence, a cell can simultaneously be used for function generation and routing. The selector signals *sel0* and *sel1* are determined by configuration bits. At 4x4 block boundaries, the neighbor connection can be driven by a length-4 FastLANE as well.

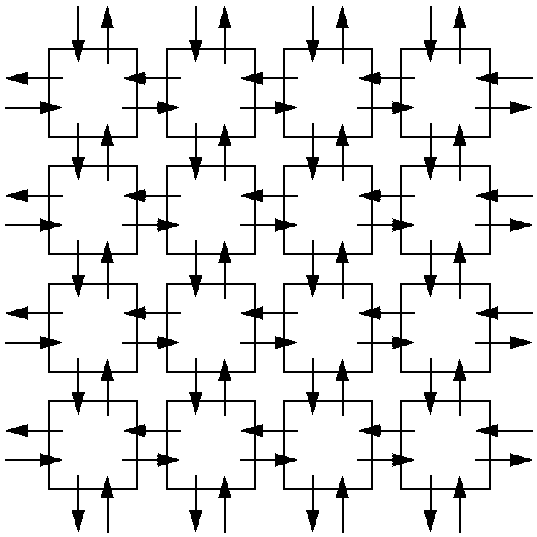

Figure 2.6: XC6200 Neighbor Routing

The next level of the routing hierarchy is composed of length-4 FastLANEs. They run across four cells and are connected together through switches, which are spaced 4 cells apart (Figure 2.7). A length-4 FastLANE can be driven by another length-4 FastLANE or by the signals on the previous or next level.

Accordingly, there are length-16 FastLANE signals driven by switches spaced 16 cells apart and chip-length FastLANE signals driven by switches located at the perimeter of the cell array.

Four *global signals* (G1, G2, GClk, GClr) are provided for low skew, low delay signals such as register clear and clock. [Xil96] presents the routing architecture in more detail. It also describes the topology of the *magic routing resources*, which are not supported by our tools due to their irregular structure with respect to hierarchy. They allow to connect the signal of the *X2* or *X3* input of a cell to two 4x4 switches.

2.3.3 Input/Output Blocks

Surrounding the array of cells, configurable input/output blocks (IOBs) are located at every cell location. An IOB can be configured to act as an input, an output or a bi-directional driver controlled by a tri-state enable signal. Not every IOB is connected to a pad (padless IOB). A novel feature of the XC6200 is that padless IOBs can be connected to additional inputs of IOBs with pads. This gives additional flexibility for the routing of signals. Also, it is possible that an IOB can drive a control signal which goes into the array, for instance the chip select signal. This feature makes it possible for a chip to generate its own control signals, thereby

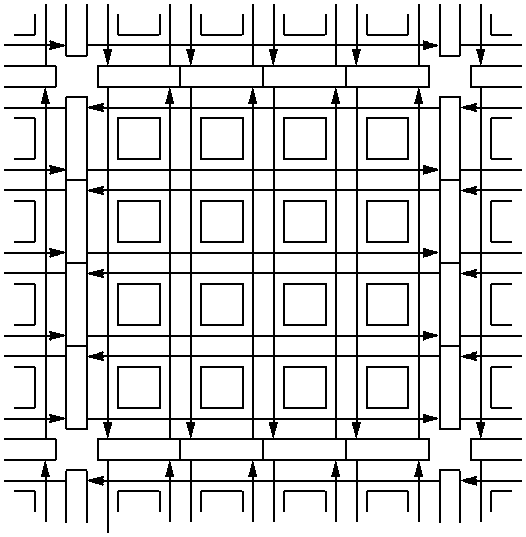

Figure 2.7: XC6200 Length-4 FastLANEs

making external control logic obsolete. The input/output architecture is discussed in more detail in [Xil96].

2.3.4 Programming Interface

From a host processor, the XC6200 FPGA can be accessed like a conventional SRAM using data, address, and control signals (Figure 2.8).

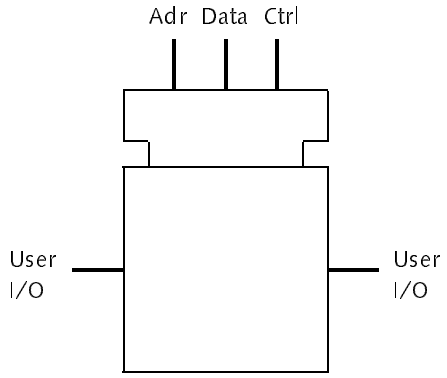

Figure 2.8: XC6200 Logic Symbol

Simple memory-mapped reads and writes using an up to 32-bit wide data bus are used to configure the chip and to access the values of cells. Using this fast interface at a clock rate of 33 MHz, it is possible to fully configure an XC6216 (with 64 x 64 cells) in 270 μs. The chip can also be partially reconfigured down to a single configuration bit. Applications of this partial reconfigurability will be discussed in Chapter 6.

Special registers are provided for the quick configuration of multiple rows and columns of cells and routing resources: The *wildcard register* is used to configure regular structures which occur, for example, in every second column of every fourth row of the chip. The *mask register* is used to change only the relevant bits, for example, of a north multiplexer within a

cell. For state access, the *map register* is used to map bits on the data-bus to individual cells in a column (e.g. bit 0 of the data bus is mapped to the cell in row 1, bit 1 to the cell in row 3, etc.). [CKW95] and [Xil96] treat this subject in more detail.

2.3.5 Summary

The XC6200 has a simple, regular structure with simple logic cells and a hierarchical, unidirectional routing network. A fast programming and access interface can be used for rapid, partial reconfiguration and for accessing (reading and writing) user registers without using routing resources.

On the downside, inversions on routing and logic multiplexers complicate the implementation of various software tools and the magic routing resources break the symmetry of the other routing resources.

2.4 Other Architectures

In this section we give an overview of three other FPGA architectures, namely the CAL by Algotronix, the XC4000EX by Xilinx and the AT6000 by Atmel. There are many more FPGAs on the market and discussing them all would be beyond the scope of this introduction. A more detailed, if somewhat older presentation of different architectures can be found in [BFR92].

2.4.1 CAL

The predecessor of the XC6200, the CAL (Configurable Array Logic) by Algotronix is not available any more but is presented here for historical reasons [Alg90, Kea89]. It was one of the first *fine-grained* SRAM-based FPGAs, featuring 32 by 32 cells, each of which could implement *any function of two inputs or a latch*. Figure 2.9 shows the function unit of the chip. It had *neighbor to neighbor connections*, just as the XC6200, but these were the only routing resources (Figure 2.6).

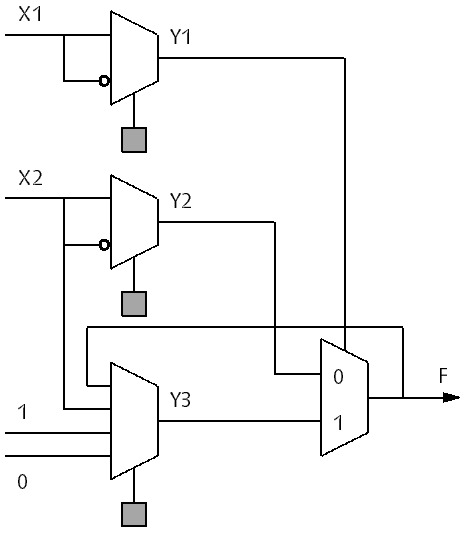

Figure 2.9: CAL Function Unit

The CAL was used in reconfigurable coprocessors, such as the CHS2x4 from Algotronix [Alg91] and the Chameleon computer developed at the Institute for Computer Systems at ETH

Zürich [Hee93, HP92]. The CAL may be regarded as a pioneering work on fine-grained architectures. Its main drawbacks were long propagation delays due to the lack of long connections and the presence of level-sensitive latches instead of edge-sensitive registers.

2.4.2 AT6000

The AT6000 architecture from Atmel is a slight variant of the Concurrent Logic CLi6000 architecture [Atm95]. It is used in a laboratory for a digital design course for computer science students at ETH Zürich [GLW94, Wir95]. The AT6000 is a *fine-grained* SRAM-based FPGA, although less fine-grained than the XC6200. Each cell has three inputs, two can come from any of the four neighboring cells and one can come from one of four local buses. It drives two outputs, which are connected to all four neighboring cells. One output can be connected with pass gates to the local buses. A logic cell contains an XOR-gate, a (N)AND-gate and a register, plus two additional AND-gates reading from the local bus input (Figures 2.10 and 2.11). A cell can, for example, implement a half-adder, a counter element, a multiplexer or a loadable register.

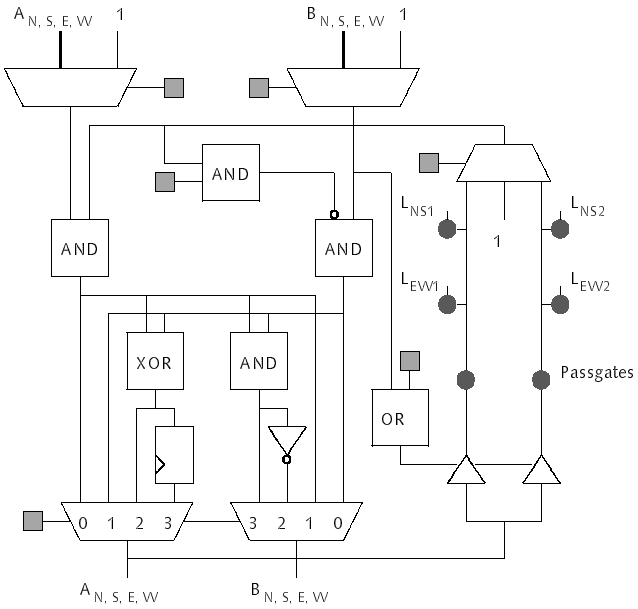

Figure 2.10: AT6000 Function Unit

For routing, the two neighbor inputs can be passed to the neighbor outputs, straight-through or crossed-over. This means that a cell is either used for logic or for routing, but never for both. On each side of the cell there is a *local bus*, which runs along 8 cells. A local bus can be driven by multiple cells, thus implementing a tri-state bus. A vertical and a horizontal local bus can be connected together, implementing a *corner turn*. At 8x8 block boundaries, so-called *repeaters* are used to connect local buses to each other or to *express buses*. The latter cannot be read by logic cells and are therefore faster.

The serial or 8-bit parallel programming interface can be clocked at 10 MHz. Partial reconfiguration is possible down to a single cell and the state of all user registers can be read.

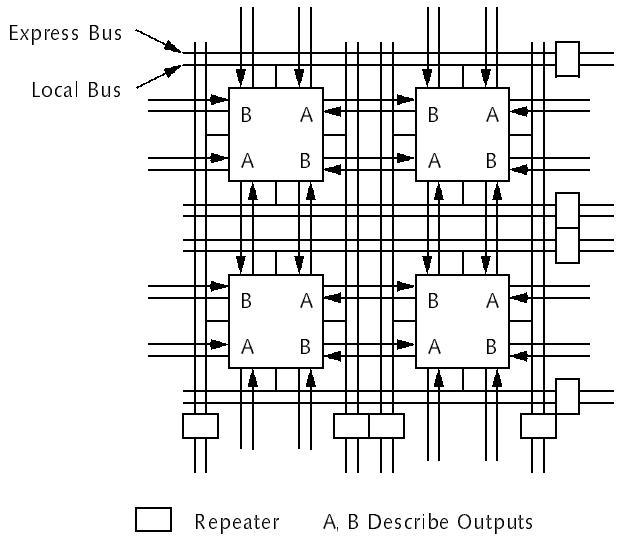

Figure 2.11: AT6000 Routing Network

2.4.3 XC4000EX

The XC4000EX is the third version of the market-leader architecture from Xilinx [Xil96]. It is a coarse-grained SRAM-based FPGA, based on the architectures of the 2000 and 3000 series. Figure 2.12 shows a diagram of a logic cell. It has three function generators (*F, G, H*), plus two registers (*XQ, YQ*). *F* and *G* have four, *H* has three inputs, two of which can be the outputs of *F* and *G*. The function generators are implemented as lookup-tables and *F* and *G* can be used for implementing *distributed SRAM*. In addition, the XC4000 features *fast carry logic* for speeding up adder and counter structures.

The routing resources are intricate, if not astounding. A single cell has 12 input signals and generates 4 outputs signals. It has 4 direct connections to its neighboring cells and access to a dedicated carry chain. Between cells, switch matrices have access to 16 single, 8 double, 24 quadruple length lines, 16 long lines and 8 global signals. This gives a total of 45 vertical and 32 horizontal lines near each cell. Figure 2.13 shows a simplified diagram of the routing resources. The switch matrix implements a sparse connection of the input and output signals. Apart from the switch matrix, two signals can be connected together at most cross points in Figure 2.13. Some of the longer lines can be used to implement *tri-state buses*, other signals are uni-directional in principle, although multiple sources can be connected to a signal.

The XC4000 is programmed through a serial interface (or 8-bit parallel in the case of the XC4000EX). Partial reconfiguration and the setting of user registers is not possible. However, it is possible to read the state of all logic cells.

2.5 Evaluation

Comparing FPGA architectures against each other is difficult. Each one of them has its strengths in certain application domains and its weaknesses in others. A simple fact is that Xilinx's 3000, 4000 and 5200 architectures (all lookup table based, coarse-grained FPGAs) hold roughly 70% of the FPGA market. This seems to indicate that these architectures are well suited for different application domains. (Or it could indicate that customers play it safe and go with the market leader.)

Table 2.2 and the following sections summarize the differences and similarities of the

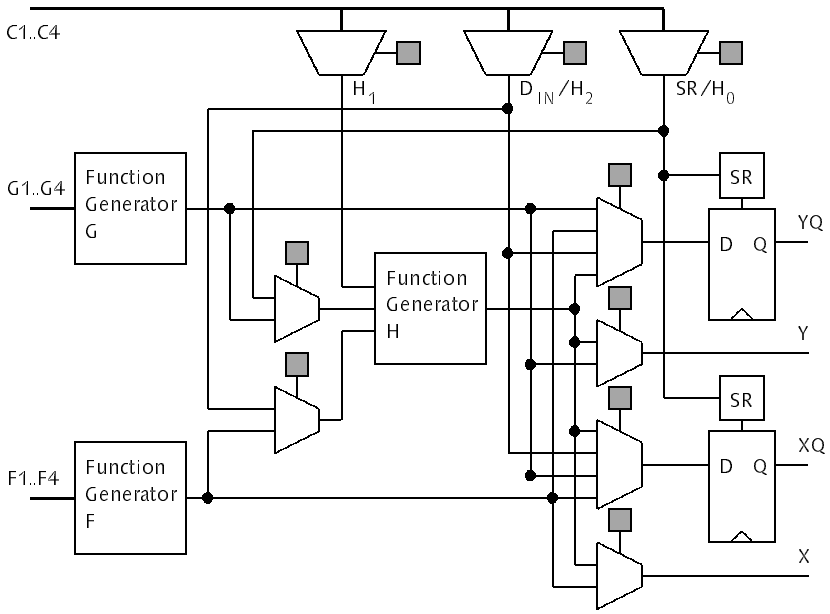

Figure 2.12: XC4000EX Function Unit (Simplified)

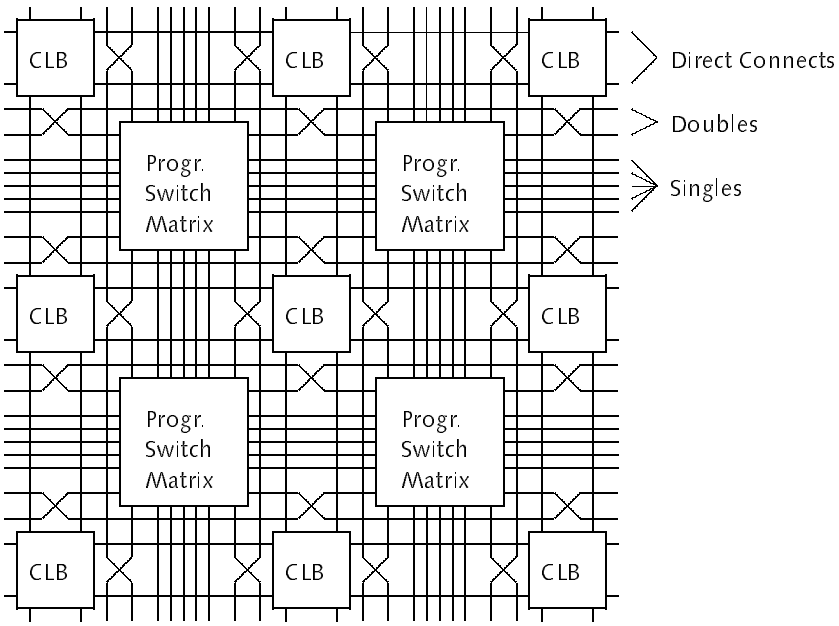

Figure 2.13: XC4000EX Routing (Simplified)

previously presented FPGAs, using representatives with equivalent gate capacity. Each feature is also evaluated with regard to coprocessor applications and suitability for hardware synthesis. We exclude the CAL architecture and we list the XC4020E instead of the XC4028EX, as even the smallest EX device has a higher gate capacity than the other two FPGAs in the comparison.

	XC6216	AT6010	XC4020E
introduced	1995	1994	1994
num. of cells	64x64 (4096)	80x80 (6400)	28x28 (784)
gates (K)	16–24	10–20	20
cell	simple	medium	complex
flipflops	4096 (1/cell)	6400 (1/cell)	2016 (2/cell + IO)
suitability for			
data-path	+	+	+
random logic	–	–	+
synthesis	+	–	+/–
distrib. RAM	partial	no	yes
routing	hierarchical	single, local, express	single, double, long
fast carry	no	no	yes
direction	uni	uni + tri-state	uni + tri-state
reconfiguration			
speed	ns–μs	μs–ms	ms
full	0.27 ms	1.3 ms	33 ms
partial	yes	yes	no
register			
read	ns	ms (all)	ms (all)
write	ns	–	–
bitstream	public	proprietary	proprietary

Table 2.2: Comparison of Different FPGAs

2.5.1 Logic Cell

The XC6200 features a simple cell which can implement any two-input Boolean function or a multiplexer. Technology mapping is easy (cf. Section 5.4). The XC4000 has a very complex cell which can implement from two functions of 4 inputs each up to certain functions of 9 inputs. Technology mapping is complex, as the software has to decide what part of a circuit to combine and put into one cell. The AT6000 cell is a collection of special cases. Technology mapping is complex, as some basic functions, such as the OR-gate, have to be constructed from several cells, or certain parts of a circuit have to be combined and put into one cell.

All FPGAs are register rich (ratio of registers to logic gates). They are well suited for pipelined, data-path intensive designs. In addition, the XC4000 with its high fan-in cells is good for implementing random logic. Also, the dedicated carry logic is a big plus for arithmetic circuits and the distributed RAM capability is useful in many applications.

2.5.2 Routing Resources

The XC6200 has a regular, hierarchical routing structure with few special cases. The XC-4000's abundant routing resources gives good routability at the cost of a complicated software

implementation because of the many special cases to consider. Both the XC6200 and the XC4000 have routing resources, which are independent of the chosen cell function. The absence of this feature is the biggest drawback of the AT6000, as the hardware synthesis software must decide at an early stage whether to use a cell for routing or for logic. If, at a later time, routing is not possible, it has to redo placement to free up some cells for routing.

The tri-state buses in the XC4000 and the AT6000 can be useful for certain applications, but their absence in the XC6200 has its advantages as well, as it is not possible to destroy the chip with a faulty configuration. This can be useful both in education [GLW94] and research [HHC96, Tho96].

2.5.3 Coprocessor Suitability

So far, most reconfigurable coprocessors use either the XC3000 or the XC4000 series from Xilinx (cf. Chapter 7). Reconfiguration times, although faster in newer devices, are quite slow (ms) with regard to coprocessor applications and partial reconfiguration is not possible. Also, reading user registers is not very fast (ms) and it is not possible to set them. Data-path intensive circuits map well onto the logic cell as does random logic. The proprietary bitstream format hinders the development of new tools by third party vendors or universities. Such tools could give better support for coprocessor applications.

The AT6000 has relatively fast reconfiguration times and also supports partial reconfiguration. For a long time being the only FPGA to support these features, the AT6000 and its cousins were a favorite among researchers for exploring partial reconfigurability [EH94, HH95, LD93, WH95]. The cell supports data-path type applications well, while it is less suited for random logic. As with the XC4000, the proprietary bitstream format hinders the development of new tools.

The dedicated memory-mapped processor interface, the possibility of fast, padless I/O and the fast reconfiguration times of the XC6200 make this chip a first choice for coprocessor applications. This is not surprising as the architecture is targeted at that market segment. It is not evident, however, if the fine-grained architecture of the XC6200 can implement arithmetic circuits as efficiently as, for example, the XC4000 with its dedicated carry logic. However, [Mul97] presents a fast adder circuit (also discussed in Chapter 6) and [KNS96] recently described a constant multiplier with similar density and better performance than a XC4000 implementation. The logic cell lends itself to regular, pipelined data-path type applications. The availability of the bitstream format makes it possible to write new tools, exploring various possibilities of generation and reconfiguration of hardware.

2.5.4 Deciding on an Architecture

The simple cell, regular routing architecture, processor interface, fast reconfiguration times, possibility of user register access and availability of the bitstream format made the XC6200 a clear winner for implementing a reconfigurable coprocessor and associated software tools.

3 Foundations: Lola and Trianus

In this chapter, we present the foundations Hades is based on, namely the hardware description language *Lola* and the *Trianus* framework for digital circuit design with FPGAs. Lola is compared to the more popular languages VHDL and Verilog. The structure of Trianus and its data structures are explained in some detail to give the reader background information for the remaining chapters.

Further information on Lola can be found in [Wir95, Wir96b]. An overview of Trianus is given in [GL96] and detailed information in [Geh97].

3.1 Hardware Description Languages

As in most engineering disciplines, traditional hardware design involves a graphical approach using diagrams. Today, many hardware engineers still use *schematic entry* to describe their hardware designs. Hardware synthesis tools are used to translate these schematics (drawings) into *netlists* containing *gates* (such as ANDs, ORs, Registers) and *wires* connecting these gates. The graphical description is well suited to describe the global signal flow in a digital system. However, a schema containing many components is hard to read and understand. Also, entering and altering a schema is often a tedious task, as it involves the placement of graphical components on the drawing plane and connecting these with wires.

Carrying over the methodology used in software programming languages, a textual description of hardware is possible using hardware description languages (HDLs). These textual descriptions are *precise* and the *semantics* of the individual language constructs are *well defined*. Describing repetitive components is accomplished easily and descriptions can be parameterized with certain values, such as the width of a bus. HDL compilers perform the same task as their graphical cousins, but take a textual description of the hardware as input instead of a graphical one. Last, but not least, the demands posed on a computer system are much smaller when text is processed instead of graphics.

3.2 The Hardware Description Language Lola

Lola (logic language) was designed by N. Wirth in 1992 as a simple, easily learned hardware description language for describing synchronous, digital circuits. In addition to its use in a digital design course for second year computer science students at ETH Zürich [GLW94], the Institute for Computer Systems uses it as an HDL for describing hardware designs in general and coprocessor applications in particular. The complete syntax is listed in Appendix A.

3.2.1 Overview

The purpose of Lola is to *statically* describe the structure and functionality of hardware components and of the connections between them. A Lola text (or program) is composed of *declarations* and *statements*. Statements consist of control statements and assignments. A program describes the hardware on the gate level in the form of *signal assignments*. Signals are combined using operators, thus forming expressions. These expressions can be assigned

to other signals. Signals and the respective assignments can be grouped together into *types*. Types can be composed of instances of other types, thereby supporting a hierarchical design style. An instance of a type is a hardware component (such as an adder). Types can be generic, e.g. parameterizable with the word-width of a circuit.

3.2.2 Variables, Signals and Assignments

Variables serve to give a name to an electrical signal (a wire or net), a group of signals or a component. Each variable is either of a signal type, an array type or a composite type. There are three basic signal types in Lola: BIT (single source signal), TS (tri-state bus) and OC (open-collector bus). A signal carries the value zero ('0) or one ('1), or undefined in the case of a tri-state bus.

Signals can be declared in one of four different declaration sections: IN (input), INOUT (input/output), OUT (output) and VAR (variable). Input signals may only be read, input/output signals are tri-state or open-collector buses and may be read and written, output and variable signals may be read and written. IN, INOUT and OUT signals make up the interface of a type (cf. Section 3.2.4). INOUT and OUT signals are visible outside the scope they are declared in, whereas VAR signals are local to the scope.

In an *assignment* var := exp, the expression exp defines the value of variable var. There may only be one assignment to a variable of BIT type. Variables of type TS may have multiple, *conditional assignments*. Variables of type OC may have multiple assignments.

3.2.3 Operators, Expressions, Control and Position Statements

Table 3.1 lists the basic operators of Lola, which are used to combine signals. In addition to unary and binary operators, there are multiplexer, latch, register and set-reset latch operators. Note that the register is not a special signal type, but an operator; it can appear anywhere in an expression. This has the advantage that enable and special clock signals can be associated with the register in an expression instead of in the declaration part.

Operator	Meaning
~ a	not (negation)
a + b	or (conjunction)
a * b	and (disjunction)
a - b	exclusive or
MUX(s: a, b)	multiplexer (~s*a + s*b)
REG(clk: en, d)	register (optional clock and load enable)
LATCH(en, d)	latch, with load enable
SR(s', r')	latch with set and reset (both active low)

Table 3.1: Lola Operators

An *expression* can be composed of operators, variables, numeric values and numeric expressions. If a variable is of type array, a selector may be used to specify an individual signal (e.g. a.i or a[i] specify the i^{th} signal of an array). If a variable is of composite type, a selector may be used to specify an output signal of that type (e.g. a.co would specify the carry-out signal of an adder instance).

The *control statements FOR* and *IF* are typically used to iterate over an array variable, and to treat special cases in generic types, respectively.

Position statements can be used to annotate variable names with positional information. They make a Lola program target dependent and their interpretation is left to the synthesis

back-end. They are typically used to give hints to a placement algorithm or to specify pin locations (see Section 5.5).

3.2.4 Types, Unit Assignments and Modules

Type definitions are used to group related declarations and assignments together into an entity (i.e., a macro) describing a hardware component (e.g. an adder). A type definition consists of an *interface definition* (input, input/output and output signals) and local signal declarations and statements. Types can contain instances of other types, thus building a hierarchy. This is very useful for structuring large designs into smaller, reusable components. Types can also be generic, e.g. the word-width of a circuit need not be known in advance. Types are instantiated (with concrete parameters) in variable declarations. Actual input signals are passed to an instance in a so-called *unit assignment* (see the example in Section 3.2.5).

Signals defined in the OUT section of a type are visible in the scope of the composite variable and are accessed by means of selectors. E.g., if a type A has an OUT signal co, then with a being an instance of type A, a.co can be used in an expression to access this signal.

Modules are the textual unit of compilation. They contain type and variable declarations and statements. Types in modules may be exported and imported by other modules, hence allowing the construction of libraries.

3.2.5 Example: Ripple-Carry Adder Circuit

The two types in Program 3.1 show the definition of a full adder and of a ripple carry adder of unspecified word-width, which is based on the full adder.

Program 3.1 Ripple-Carry Adder Types in Lola

```
TYPE AddElem;                          full adder
    IN x, y, ci: BIT;                  inputs: data, carry
    OUT s, co: BIT;                    outputs: sum, carry
    VAR h: BIT;                        half sum
BEGIN
    h := x-y;                          XOR
    s := h-ci;                         XOR
    co := x*y + h*ci                   two ANDs and one OR
END AddElem;

TYPE Adder(N);                         generic N-bit ripple carry adder
    IN x, y: [N] BIT; ci: BIT;         inputs: 2 data vectors & carry
    OUT s: [N] BIT; co: BIT;           outputs: sum vector & carry
    VAR add: [N] AddElem;              instantiate N full add elements
BEGIN
    add.0(x.0, y.0, ci);               unit assignment: add two bits
    FOR i := 1 .. N-1 DO add.i(x.i, y.i, add[i-1].co) END;
    FOR i := 0 .. N-1 DO s.i := add.i.s END;
    co := add[N-1].co
END Adder;
```

They are used to construct an 8-bit adder circuit shown in Program 3.2. The use of several operators and constant signals is shown, as well as the instantiation of a generic type and the use of unit assignments. The example is split up into two programs to make comparison with

the subsequent VHDL and Verilog descriptions easier. We will refer to this example in this and later chapters.

Program 3.2 Ripple-Carry Adder in Lola

```
MODULE Add;

    types as defined in Program 3.1

    CONST Bits := 8;
    IN ldx, rd: BIT;
    INOUT D: [Bits] TS;
    VAR
        adder: Adder(Bits);          instantiate generic
        x, y: [Bits] BIT;
BEGIN
    FOR i := 0 .. Bits-1 DO          store D bus if not read
        x.i := REG(~rd*ldx, D.i);    into x if ldx
        y.i := REG(~rd*~ldx, D.i)    into y if ~ldx
    END;
    adder(x, y, '0);                 unit assignment (with vectors
                                     x and y and constant zero)
    FOR i := 0 .. Bits-1 DO
        D.i := rd | adder.s.i        rd controlled tri-state assignment
    END
END Add.
```

3.2.6 Compilation of Lola

In analogy to a high-level programming language compiler, the Lola compiler performs a syntax- and type-check on the program and then generates an abstract syntax tree representing it. An interpreter traverses this tree to generate an expanded data structure with signal and expression nodes (cf. Section 3.3). The expansion step is necessary to generate the required number of nodes described in FOR loops and IF statements, and to evaluate position statements. This interpretation step, however, is different from the interpretation step in Verilog and VHDL, which is based on a simulation model to generate the actual hardware (cf. Section 3.2.7). The values of individual signals is not known to the Lola interpreter as only the control statements are interpreted.

After expansion, an optional simplification step may be executed on the resulting data structure. This simplification step propagates constants through the expression tree. An individual instance may thus look quite different from its original type definition. For instance, if the carry input is zero, the lowest full adder in Program 3.1 degenerates into a half-adder.

3.2.7 Other HDLs

HDLs used in industry are *Verilog* and *VHDL* (Very High Speed Integrated Circuits HDL), both developed in the 1980s. Both languages were originally designed for hardware simulation. The dynamic aspects are defined by the way a simulator works. A hardware synthesizer must make an interpretation of the described constructs and map this into hardware with equivalent behavior. Some language features are solely used for simulation and cannot be im-

plemented directly. Therefore, one often speaks of *synthesizable* VHDL or Verilog, which are subsets of the language definition.

Compared to Lola, both languages are much more complicated and support more features. VHDL and Verilog support operator overloading, so the adder in Program 3.2 might be written as `D := x + y`, and the actual implementation of the adder is left to the hardware synthesizer. Also, as both languages are used for simulation, a signal may carry more values than just zero and one, which sometimes leads to code that can be simulated but not synthesized.

In VHDL, the interface (entity) and the implementation (architecture) of a type are textually separated. An entity may have multiple architectures, e.g. an adder circuit may be implemented using a ripple-carry or a carry-look-ahead scheme. The types from Program 3.1 would look like Program 3.3 in VHDL.

Due to the separation of interface and implementation, the designer has the possibility to provide multiple implementations for the same interface. For libraries, this is a welcome feature, but for application code, this approach leads to verbosity and more code has to be written. Another problem with this approach is that the declarations of interface and private signals are textually separated. When writing the implementation, the designer always has to consult the entity declaration to see what input and output variables are available.

Verilog has no generic construct. One has to use a preprocessor which then generates the corresponding code. An 8-bit adder in Verilog is shown in Program 3.4.

An interesting note is that unlike in Lola, it is not possible in VHDL or in Verilog to use the output signal of an instance directly. In the `Adder` type, we need an auxiliary variable `c` and a *map statement* to transfer the carry-out of the previous `AddElem` to the carry-in of the next one. This may seem like a trivial limitation, but it ultimately decides whether a language is "handy" or not. Table 3.2 compares Lola with VHDL and Verilog and lists some of the major differences.

Feature	Lola	VHDL	Verilog
Simple	+	-	+/-
Generics	+	+	+/-
Overloading	-	+	-
Structural	+	+	+
Behavioral	-	+	+
Synthesizable	+	+/-	+/-
Position Hints	+	+/-	+/-
Standard	-	+	+
Fast Compilers	+	-	-
Inspired by	Oberon	Ada	C

Table 3.2: Lola vs. VHDL vs. Verilog

3.3 Trianus

Trianus is the name of the companion project of Hades, carried out by Stephan Gehring [Geh97]. It serves as the base and foundation for the Hades software. The principal architecture was designed by S. Gehring and then fine-tuned based on experience with the Hades tools. It has proven to be a very robust base for Hades and its implementation is sound and well tested.

Program 3.3 Ripple-Carry Adder in VHDL

```
entity AddElem is                          interface
    port (x, y, ci: in bit;
          s, co: out bit);
end;

architecture behavior of AddElem is        one implementation
    signal h: bit;
begin
    h <= x xor y;
    s <= h xor ci;
    co <= (x and y) or (h and ci);
end behavior;

entity Adder is                            interface
    generic (n: natural := 4);
    port (x, y: in bit-vector(n-1 downto 0);
          ci: in bit;
          s: out bit-vector(n-1 downto 0);
          co: out bit);
end;

architecture structure of Adder is         one implementation

    component AddElem                      declare used entities
        port(x, y, ci: in bit;             interface must be repeated
             s, co: out bit);
    end component;

    signal c: bit-vector(n downto 0);      auxiliary carries
begin
    c(0) <= cin;
    gen: for i in 0 to n-1 generate
        ae: AddElem port map(              unit assignment
            x => x(i),
            y => y(i),
            ci => c(i),
            s => s(i),
            co => c(i+1));
    end generate;
    cout <= c(n);
end structure;
```

Program 3.4 Ripple-Carry Adder in Verilog

```
module AddElem(x, y, ci, s, co);
    input x, y, ci;
    output s, co;

    wire s, co;                          define output pin types
    wire h;

    assign h = x ^ y;
    assign s = h ^ ci;
    assign co = (x & y) | (h & ci);
endmodule

module Adder(x, y, ci, s, co);
    input [7:0] x, y;
    input ci;
    output [7:0] s;
    output co;

    wire [7:0] s;                        define output pin types
    wire co;                             define output pin types
    wire [6:0] c;                        auxiliary carries

    instantiate eight adder elements
    AddElem bit0(.x(x[0]), .y(y[0]), .s(s[0]), .ci(ci), .co(c[0]));
    AddElem bit1(.x(x[1]), .y(y[1]), .s(s[1]), .ci(c[0]), .co(c[1]));
    AddElem bit2(.x(x[2]), .y(y[2]), .s(s[2]), .ci(c[1]), .co(c[2]));
    AddElem bit3(.x(x[3]), .y(y[3]), .s(s[3]), .ci(c[2]), .co(c[3]));
    AddElem bit4(.x(x[4]), .y(y[4]), .s(s[4]), .ci(c[3]), .co(c[4]));
    AddElem bit5(.x(x[5]), .y(y[5]), .s(s[5]), .ci(c[4]), .co(c[5]));
    AddElem bit6(.x(x[6]), .y(y[6]), .s(s[6]), .ci(c[5]), .co(c[6]));
    AddElem bit7(.x(x[7]), .y(y[7]), .s(s[7]), .ci(c[6]), .co(co));
endmodule
```

3.3.1 Motivation and Structure

The *Trianus* project tries to improve performance of hardware design tools by tightly integrating them through *one common data structure*. It decomposes the tools into a *front-end* and several *back-ends*, all integrated through a *framework*. Trianus features a *circuit checker*, with which different representations of the same circuit can be checked for equivalence (e.g. a hand layout can be compared with a Lola specification), and a *browser*, which can extract a textual view from a layout or a schematic.

The result is the Trianus framework for FPGA design [Geh97, GL96]. It consists of an *architecture independent front-end* for which *several architecture dependent back-ends* can be developed. The front-end encapsulates common operations on design data (Section 3.3.2), an HDL interpreter back-end (Section 3.3.4), a circuit checker (Section 3.3.6), a browser (Section 3.3.6), and a graphical user interface framework for editors.

The separation into a front-end and several back-ends bears the advantage that based on the framework, new back-ends (for a new FPGA architecture, for instance) can efficiently be developed and integrated. For the user of the system, this results in a uniform interface and consistent behavior of the tools. For the programmer, the use of a shared front-end and a common data structure reduces the amount of code to be written and tested when implementing a new back-end. This reduction in code complexity results in a more reliable and smaller system.

The name Trianus is a derivation from the name of the two-headed Greek god Janus and stems from the fact that the framework supports *three different views* onto the same circuit, namely a *textual view* by means of a Lola program, a *schematic view* and a *layout view* in a layout editor. Figure 3.1 gives a graphical representation of possible transitions between views.

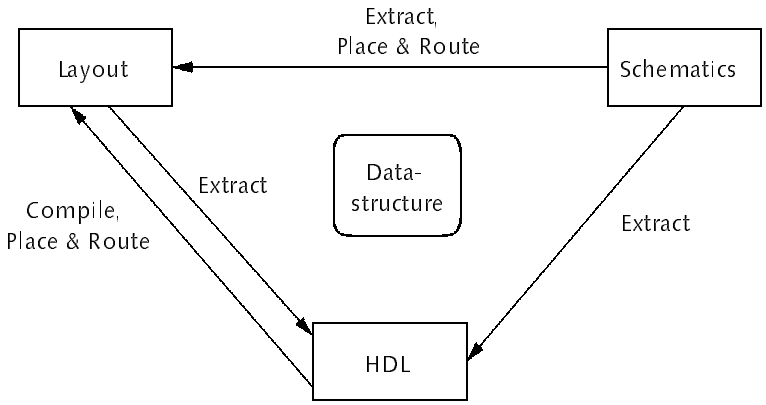

Figure 3.1: Different Views in Trianus

The three views in Trianus give users a choice between different representations of a design. Some prefer describing their design with a program, some with a schematic and some with a layout. Using the circuit checker and extractor, consistency between the three views can always be verified (manually or automatically).

By using only one common data structure for representing design data, intermediary file input and output as shown in Figure 1.3 is avoided. Hence the design flow shown in that figure can be simplified to the one shown in Figure 3.2. The shaded areas correspond to the Lola HDL and the Trianus software described in this chapter.

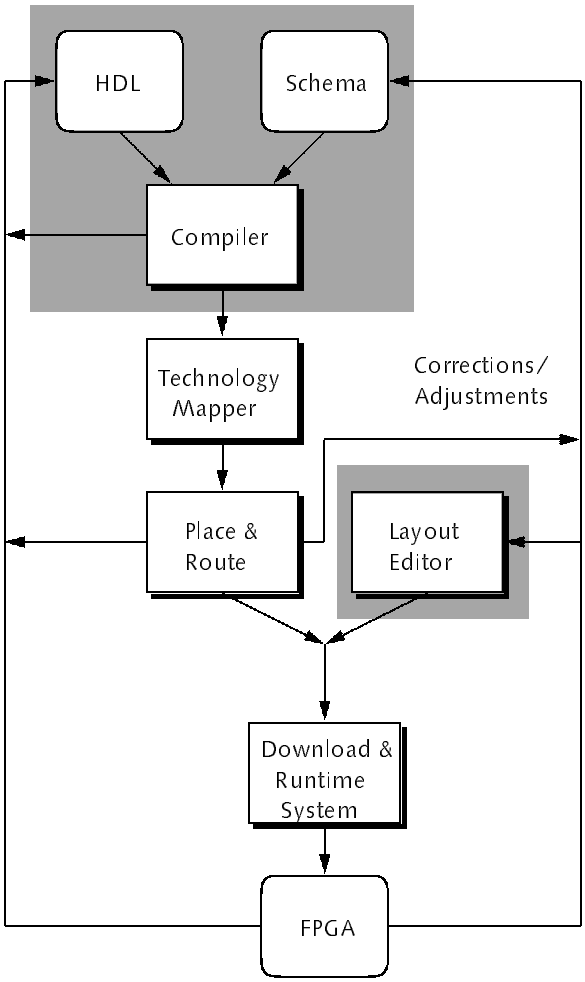

Figure 3.2: Lola and Trianus Part in Design Flow from Fig. 1.3

3.3.2 Data Structures

In the following, we give a brief overview of the basic data structure used for design representation in Trianus. We refrain from introducing every detail and leave out those aspects which are not relevant in this thesis's context. In later chapters we give more information on the data structures should the need arise. For a more detailed discussion of Trianus and its underlying design, we refer the reader to [Geh97].

The central data structure describes a hardware circuit in a general, compact, device independent manner. *Generality* and *compactness* of the data structure are important attributes, as they make it possible to keep the data in memory between different phases in the design cycle. No translation steps are necessary when switching between different tools and, especially, no files need be written and read again. Also, from a software engineering point of view, using only a single data structure instead of a multitude reduces system complexity and learning time.

The data structure is based on the constructs offered by Lola (cf. Figure 3.3). Hierarchical information is available and maintained across all tools. Operators, signal names, types, instances and modules exist in a Trianus data structure, as well as wires which are used to connect operators. Other than basic geometric information (u, v, w, h, to in Program 3.5), no device specific information is stored. Specifically, additional temporary data needed by a back-end tool (e.g. a placement algorithm) has to be managed by that tool itself.

Type-Based

One fundamental principle in Trianus is that *all instances of a type* have exactly the *same (relative) placement* information and the *same wiring.* That is, the placement and wiring of a type determines the placement and wiring of all its instances. This concept is ensured by the system and has to be ensured by all tools as well. Trianus provides algorithms (cf. Section 3.3.3) for distributing information from a type to all its instances. We call tools or algorithms *type-based*, if they have this property of propagating information from a type to all its instances. It implies that an algorithm makes only direct changes to the data structure of a type, and never to that of an instance.

Type Definitions

The following sections describe the types occurring in a Trianus data structure, as they are defined in the Oberon programming language [RW92]. In Trianus, Oberon's type extension is used for specializing behavior and describing the semantic difference between the types. When describing extended types, those inherited fields which have a different meaning than in the base type are listed in parentheses and an explanatory comment is added.

Node

Node is the basic type for building a data structure describing a circuit. Its definition is shown in Program 3.5.

The fct field describes the function of a node. For a "pure" node, this is one of the basic operators of Lola (e.g. not, and, mux, register). For extensions of Node, the function is listed in the respective paragraphs below. A node, per se, exists in the data structure only to describe operators and abstract syntax trees (e.g. the - operator in type AddElem of the Add example of Program 3.1). See Figure 3.4 for a graphical representation of an operator and operand tree, and Section 3.3.4 for an explanation of the use of abstract syntax trees.

Program 3.5 Definition of Node

```
Node = POINTER TO NodeDesc;
NodeDesc = RECORD
    fct: SHORTINT;          operator (Zero, Not, And, Reg, ...)
    x, y: Node;             operands
    link: Node;             next in same instance
    outer: Instance;        outer instance/type
    wire: Wire;             list of wires to all destinations
                            of this node
    u, v, w, h: INTEGER;    physical location and dimension
    to: SHORTINT;           additional physical information
    id: INTEGER             general purpose field
END
```

Object

Variables of a signal type are represented by an Object type, shown in Program 3.6, which is derived from Node.

Program 3.6 Definition of Object

```
Object = POINTER TO ObjectDesc;
ObjectDesc = RECORD (NodeDesc)
    (fct)                   BIT, TS, OC
    (x)                     expression tree defining this signal
    (y)                     always NIL
    name: Name;             name of signal/instance/type
    type: Type;             type of signal/instance
    mode: SHORTINT;         IN, INOUT, OUT, VAR
    next: Object            next in declaration sequence
END
```

The name of an object is the same as the one in the Lola declaration. mode indicates the type of the signal (input, tri-state/open-collector, output, variable). type points to the BIT, TS or OC type structure and fct (from Node) is either BIT, TS or OC. (E.g. x, s, h in type AddElem of the Add example.)

Elements of a signal array are also represented as objects. The only way to determine that an object is indeed an array element is through its name. Individual elements of a declared array variable x: [3] BIT are represented as objects with names x.0, x.1, x.2.

Instance

Instances of composite and generic types are represented by the Instance type, shown in Program 3.7, which is derived from Object.

The name of an instance is the same as the one in the Lola declaration. mode is always VAR, as instances may only be declared in VAR sections. type points to the type structure of which this instance is an instantiation. (E.g. add: [N] AddElem; in type Adder of the Add example).

Program 3.7 Definition of Instance

```
Instance = POINTER TO InstanceDesc;
InstanceDesc = RECORD (ObjectDesc)
    (fct)                       Inst
    (x)                         list of unplaced nodes
    (y)                         list of placed nodes
    (mode)                      VAR
    dsc: Object;                list of interface signals and
                                local variables/instances
    open: BOOLEAN               are contained objects visible?
END
```

Type

Composite and generic types are represented by the Type type. For each instance of a generic type, a concrete composite type exists which contains the actual parameters. Type is derived from Instance. Its definition is shown in Program 3.8.

Program 3.8 Definition of Type

```
Type = POINTER TO TypeDesc;
TypeDesc = RECORD (InstanceDesc)
    (fct)                       Typ/Module
    (mode)                      Expanded, Parameterized
    code: Object;               syntax tree for interpretation
    marked: BOOLEAN;            for export
    color: SHORTINT
END
```

The name of a type is the same as the one in the Lola declaration. mode indicates, whether it is a generic or a normal composite type. code points to the syntax tree, which is used for interpretation and for generation of expanded types out of generic ones. The marked field is TRUE when the type is marked for export. color is used by visualization tools. Since a Lola module is both a description and the only instantiation of a circuit, a type is used to represent the outermost scope of a program. (E.g. type AddElem, type Adder(N) in Program 3.1, module Add in Program 3.2.)

Wire

The last type left to describe is Wire. It represents a physical connection in a circuit, whereas logical connections are represented by the operand tree of Node. For example, an OR-gate reading from two AND-gates as shown in the full adder in Program 3.1 has two logical connections to the AND-gates represented by the x and y pointer in the Node data structure, while the actual connections made in an XC6200 FPGA, for instance using neighbor connections, would be represented by Wires. The definition of Wire is listed in Program 3.9.

The specific values for from and to are determined by the respective back-ends. The back-end for the XC6200 FPGA, for instance, stores information on the routing multiplexers into from and to (e.g. from = Function Unit output, to = West neighbor routing multiplexer).

Program 3.9 Definition of Wire

```
Wire = POINTER TO WireDesc;
WireDesc = RECORD
    from, to: SHORTINT;          physical source and destination
    u, v, w, h: INTEGER;         physical location and dimension
    next: Wire;                  next in same net
    link: Wire;                  next in same instance
    outer: Instance;             outer instance/type
    id: INTEGER
END
```

Summary and Example

Figure 3.3 summarizes the principal types of a Trianus data structure and shows the corresponding Lola constructs.

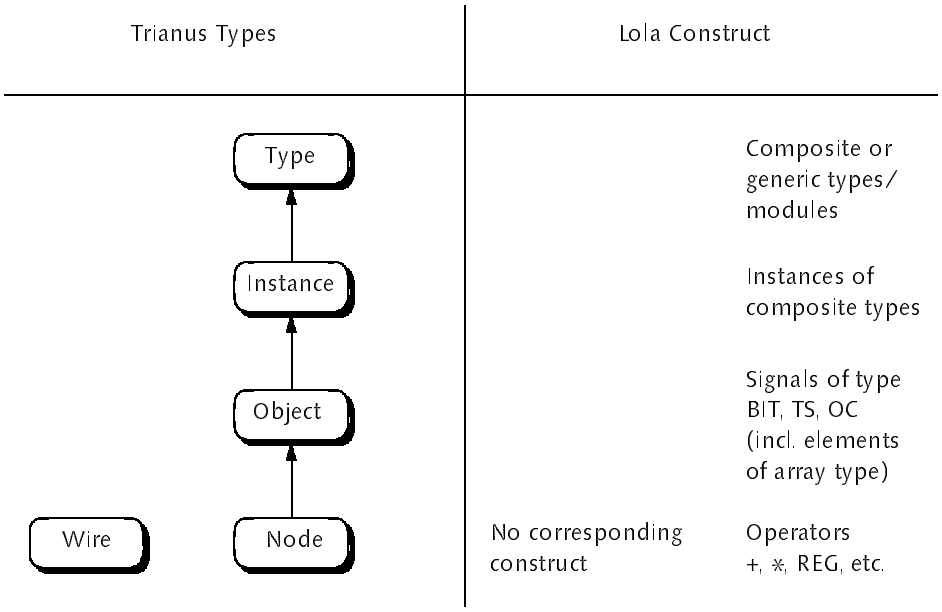

Figure 3.3: Trianus Types and Lola Constructs

The possible values and their meaning for the function field, which are relevant to Hades, are listed in Table 3.3. Additional values are possible and are used for representing the abstract syntax tree of a Lola program (Section 3.3.4).

As an example, we describe in Figure 3.4 the data structure generated for the AddElem type in the Add example from Program 3.1.

A similar data structure exists for every instance of such an AddElem. The dsc (descender) list of an 8-bit instantiation (adder) of the generic Adder type contains 8 instances of AddElem (add.0, add.1, etc.), each containing the descender list (x, y, etc.) shown in Figure 3.4.

Value of fct	Meaning
BIT	uni-directional signal
TS	tri-state bus
OC	open-collector bus
Inst	instance of a type
Typ	type (generic or expanded)
Module	outermost scope
Zero	constant zero
One	constant one
Buf	buffer
Not	inversion
And	and
Or	or
Xor	exclusive or
Latch	D-latch
SR	set-reset-latch
Reg + Reg1	register
Mux + Mux1	multiplexer
Tri + List	tri-state assignments

Table 3.3: fct Values and their Meaning

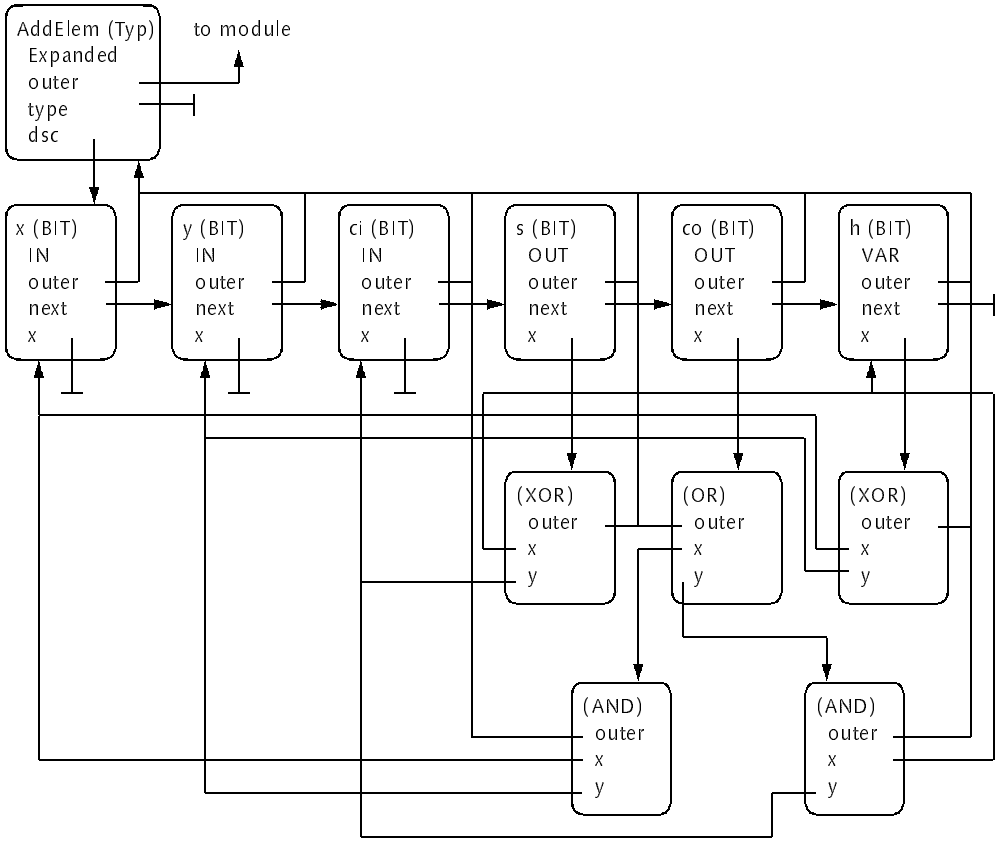

Figure 3.4: Data Structure for AddElem Type

3.3.3 Algorithms

The core of Trianus provides various algorithms which operate on the data structures just described. The most important ones for the Hades tools are algorithms for *placing nodes*, which are used by the placement algorithm, *inserting wires*, which are used by the router, and *broadcasts*, which are used by all back-end tools (such as the mapper, placer, router and loader).

The broadcasting mechanism is of special interest, as it is used to guarantee the aforementioned consistency between instances and their type. The relevant constants, types and procedures are shown in Program 3.10. Depending on a selector, a message based on the base type Message is sent to the nodes and wires in a Trianus data structure, including all extensions of nodes. For instance, it is possible to send a message to all instances of the same type by using the selector SelType. This can be used to distribute placement and routing information to all instances of a type, after the type has been placed and routed.

Note: the terminology used herein for message sending is not to be confused with the one from object-oriented programming. The broadcast mechanism is a generic *data structure iterator* construct. It applies operations to the nodes and wires of a data structure. The nodes and wires do not react to the message sent to them, it is rather the iterator mechanism that invokes the operation on the nodes and wires. The next paragraph explains this process in more detail.

The broadcast mechanism starts at an instance and iterates over all nodes and wires in that instance, proceeding recursively into sub-instances contained in that instance. It invokes (if present) the procedures stored in the procedure variables doNode for each node and extensions thereof, and doWire for each wire encountered. It is possible to send a message based on Message to

- all nodes and wires,
- all instances of a certain type,
- all visible nodes and wires within a given rectangle, and
- all placed nodes and wires.

3.3.4 Lola Compiler Back-End

A Lola program is translated by the Lola compiler into an abstract syntax tree [Wir96b], which can then be interpreted by a Lola interpreter to generate an expanded data structure representing the circuit. Compiling, type checking and generating the syntax tree is very fast, as only a single pass over the source code is needed. No logic minimization is performed, since this would possibly break the correspondence between an instance and its type, violating an invariant in Trianus.

The Trianus Lola back-end translates the produced syntax tree of the Lola compiler into a syntax tree based on the Trianus data structure, which is the same as for representing a circuit, namely Node objects with special values in the function (fct) field.

The data structure representing the syntax tree is interpreted and the circuit is expanded (or generated) into another Trianus data structure. The expanded data structure can be passed to back-end tools, such as a placer and router, for further processing.

As an example, consider the code for the adder in Program 3.2. The first two FOR loops in the body of module Add will generate, when interpreted, a number of registers for the x and y vectors and some negations and AND-gates for the load enable signals. Likewise, interpretation of the unit assignment of adder will connect the x and y vectors with the input vectors of an instantiated 8-bit adder, which is generated by interpreting the code in the generic

Program 3.10 Message Broadcast

```
CONST
    message broadcast selectors
    SelAll = 0;                 to all nodes/wires
    SelType = 1;                to all instances of type msg.type
    SelRect = 2;                to all nodes/wires (partially)
                                visible within msg.r
    SelVisible = 3;             to all nodes/wires (partially)
                                visible in open instances within
                                msg.r
    SelTop = 4;                 to all nodes/wires completely
                                within msg.r, not going into
                                sub-instances
    SelPlaced = 5;              to all placed nodes/wires

TYPE
    MessageBase = RECORD END;

    NodeProc = PROCEDURE(node: Node; VAR msg: MessageBase);
    WireProc = PROCEDURE(wire: Wire; VAR msg: MessageBase);

    Message = RECORD (MessageBase)
        r: Rect;
        type: Type;
        doNode: NodeProc;       called for each node
        doWire: WireProc        called for each wire
    END;

PROCEDURE Broadcast(inst: Instance; VAR msg: Message; sel: SHORTINT);
    send msg to all nodes and wires in inst, sel is the broadcast selector
```

Adder type shown in Program 3.1. Within that instance, 8 full adders (AddElem) are generated, each consisting of the data structure shown in Figure 3.4.

Compilation combined with expansion is several orders of magnitude faster than with traditional VHDL or Verilog compilers (which, notably, also perform more work). See Section 6 for performance data. For a more detailed discussion of the translation and expansion steps see [Geh97].

3.3.5 XC6200 Layout Editor

The layout editor is one of many back-end tools in Trianus and is built upon the generic editor framework. It provides a low-level view of XC6200 designs as a matrix of cells with routing switches in between. Since the XC6200 FPGA features a very simple cell, its function is easily displayed by the editor as a gate (possibly in conjunction with a register). The design hierarchy and the signal names are also visualized to reflect the design structure. This almost schematic-like view helps the designer identify parts of a design quickly. A displayed design is manipulated using the mouse. Figure 3.5 shows an instance of an AddElem, with three gates and the wires between the gates. Note that the carry is implemented using the half-sum and a multiplexer, instead of two AND- and an OR-gate. The squares represent logic cells and the rectangles between the squares represent switches at 4x4 boundaries.

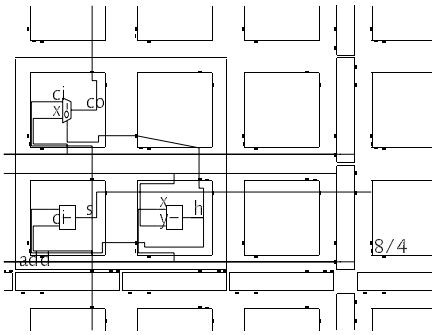

Figure 3.5: XC6200 Layout of AddElem

In addition to just displaying synthesized layouts, the layout editor can be used to create circuits from scratch. It is possible to set each cell's functionality separately using popup-menus. Connections are drawn with the mouse or with the aid of popup-menus. Groups of cells and interconnections can be combined into instances of a type (hard-macro) and stored in libraries for later use. Thus, a circuit can be constructed interactively by plugging together instances of prefabricated and tested types.

The editor is a type-based tool, i.e. when a design contains several instances of the same type, any change made to a single instance is broadcast to all instances of that same type. This feature allows for the rapid manual construction of bit-sliced designs. Furthermore, the editor can be used for floor planning by laying out empty instances and filling in functionality only later.

Quick View Updates are Essential

The editor framework of Trianus supports quick view updates. Especially in design automation tools, where hundreds or thousands of gates and wires have to be drawn, it is crucial for interactive performance that local changes to a design only cause local screen updates. Also, when the designer shifts the view onto a circuit, only those parts should be redrawn which enter the field of view; the unchanged portion of the screen should be moved using a bitblock

transfer. It is a sad fact that many commercial layout editors simply redraw the whole screen when something is changed or the view is shifted. This results in slow screen updates and flicker, and it unnecessarily slows down the editing and layout process.

3.3.6 Other Tools

Trianus offers additional tools, of which only the circuit extractor is used by the Hades router. They are briefly described in the following sections. More detailed information can be found in [Geh97].

Checker and Extractor

Rather than synthesizing circuits, it is sometimes desirable or even necessary to hand-craft a circuit with the aid of a layout editor. Such manual circuit implementation is inherently error-prone. To support manual implementation, Trianus provides a *circuit checker* which compares a design with a circuit specification. The design may be entered with the layout editor and the specification may be obtained through a Lola program. Mismatches between specification and implementation are detected and denote errors in the laid-out design, assuming a correct specification. The checker is type-based and therefore very fast.

Matching is based on ordered binary decision diagrams (OBDDs) [Bry86, Bry92, Geh97]. Each variable can be either one or zero and thus represents a decision node. OBDDs represent Boolean expressions in a memory efficient and canonical form and are thus well suited for comparison. The main drawback of OBDDs is that their size heavily depends on the ordering of the input variables. The Trianus system, however, does not suffer from this potential problem in practice, as expressions are usually small and contain only few variables. The OBDD for the carry out (co) signal of type AddElem from Program 3.1 is shown in Figure 3.6.

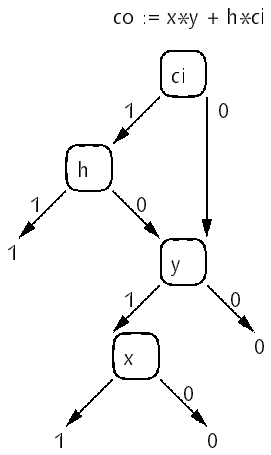

Figure 3.6: OBDD for Carry-Out of AddElem

A *circuit extractor* constructs connectivity information from a design edited with the layout editor. This information is used by the circuit checker to verify the correctness of a circuit. A circuit extractor is needed, because the layout editor does not keep connectivity information consistent when a design is being changed. Like other tools it operates on types and calculates net connection information only for a type, before propagating it to instances.

The extractor and checker are also used to allow the mixing of manual and automatic routing and for checking the correctness of the routed result (see Section 5.6).

Browser

The browser is used to translate a Trianus data structure back into textual information. Also, it serves to show the interface (hence its name) of Lola types and is an indispensable tool for browsing libraries. By using the browser, it is possible to obtain a textual description of a laid-out circuit or sub-circuit. This can sometimes be useful for viewing a circuit in a different representation, for instance, during manual layout when many wires are shown on the screen.

Schematics Editor

As a third possible view onto a circuit or as a third means for design entry, Trianus offers a schematics editor called *Schem* together with a circuit extractor and checker. It, too, is based on the editor framework and is a type-based tool. Using the browser, it is possible to extract a textual view of a schematic and pass this text to the Lola compiler. The resulting data structure can then be placed and routed using Hades. Hence, Schem is a different input possibility to describe (part of) a coprocessor application. Figure 3.7 shows an AddElem in a schematic view.

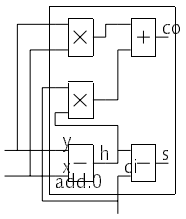

Figure 3.7: Schema Showing an AddElem

3.4 Discussion

Lola is a simple, easy to learn hardware description language for describing digital circuits on a structural level. Module libraries can be built with Lola, which in turn can be used to construct coprocessor applications on the basis of small, composable components. Position statements can be used to guide a placement algorithm to obtain a good layout. The language is small, synthesizable and has clear semantics such that it can be learned and put to work within a week.

Trianus is a fast, device independent framework for circuit design offering a general, yet simple data structure. A flexible broadcast mechanism is used to alter the data structure and to keep a consistent one-to-one correspondence between types and their instances. Type-based tools guarantee efficiency even for large designs. The layout editor is comfortable to use and provides immediate response for most operations. It can be used for manual layout and for floor-planning of large designs.

On the downside, Trianus shows quadratic run-time behavior on unfavorable input due to the simple implementation of the data structure (a linked list of nodes). This implementation will become a limiting factor of the data structure's performance, as FPGA devices get bigger. A more efficient data structure for representing the geometric relationship of nodes would be advantageous, such as quad trees [FB72] or a simple hash table as presented in Section 5.9. Also, support for libraries and for the copying of data structure elements such as types is missing. The latter is a prerequisite for the implementation of libraries, which are essential for the development of reconfigurable coprocessor applications.

4 Hades Hardware

In this chapter, we present the hardware part of this thesis, the *Hades reconfigurable coprocessor* (cf. Figure 4.1). A tutorial-style introduction to the hardware is presented in [Lud96].

Throughout this chapter, names printed in Sans Serif represent labels in figures or identifiers in programs. Names followed by an apostrophe (') represent signals which are active low.

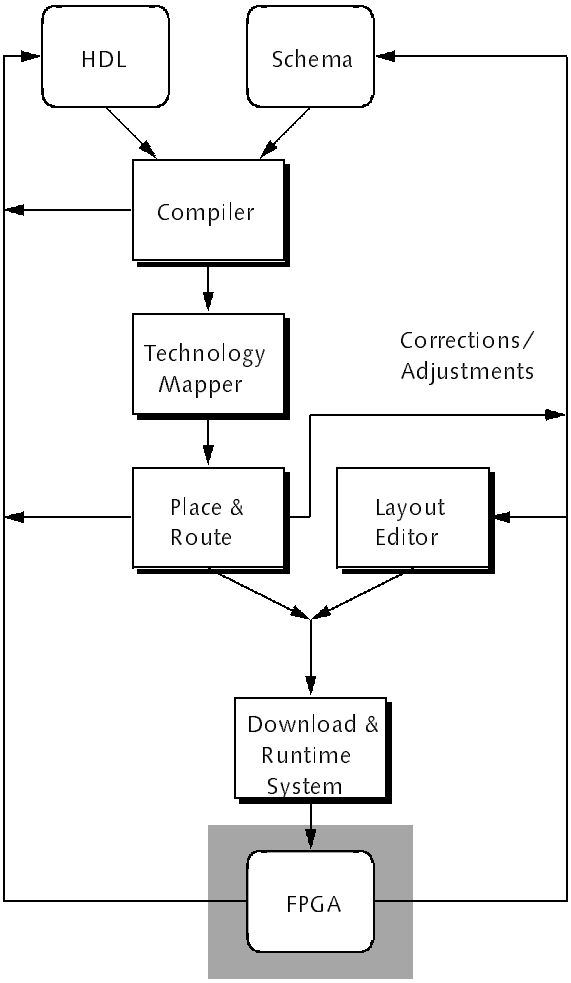

Figure 4.1: Hades Hardware Part within the Design Flow of Fig. 1.3

4.1 Motivation

Many reconfigurable coprocessor (RC) boards based on FPGAs exist today. Guccione lists over 50 different designs [Guc94]. In Chapter 7, we give an overview of related work. The architecture and structure of these boards are quite similar and the community working in the field of custom computing has a good understanding of what a reconfigurable coprocessor should look like. Most boards contain one or more FPGAs and interface hardware, which connects the board to a host computer. These boards may contain some local memory, which can be accessed by the FPGA(s) and sometimes by the host. This memory is mostly used for caching to overcome the limited communication bandwidth between the host computer and the coprocessor board [Ber93]. One deficiency of most current systems is that they are accessed through a special interface over a system bus. Compared to the (usually) very efficient protocol on a processor/memory bus, the protocol on a system bus often introduces undesired communication overhead (multiplexed address and data buses, arbitration).

With the *Hades RC board*, we wanted to overcome this problem by providing a coprocessor board with local memory that *looks like and behaves like conventional memory to the host computer*, especially in terms of latency. The Hades board makes use of the memory interface of the XC6200 FPGA. At the time of writing, the Hades board for the Ceres-2 workstation together with the driver software is the only system presenting a memory-card interface to the application, which allows fast access to the FPGA and the local memory.

An architecturally similar board for the PCI bus [LSC96, VCC97] does not yet support access through memory operations, because the driver software makes it necessary to move data using slow I/O commands. This slows down communication with the RC considerably (cf. Chapter 6 for a quantitative analysis).

4.2 Design Alternatives

As described above, our reconfigurable coprocessor should consist of one or more FPGAs attached to local memory. We wanted to build a simple system for evaluating ideas and as we only had access to two FPGA engineering samples we opted for an RC with a single FPGA. There are several alternatives to implement such an RC:

- an FPGA attached directly to the CPU, with the possibility to directly access main memory,
- an extension card connected to the system bus, containing an FPGA, with DMA capability to access main memory,
- an extension card connected to the system bus, containing an FPGA and local memory, possibly with DMA capability.

We now discuss these alternatives, presenting their advantages and disadvantages.

4.2.1 FPGA Attached to the CPU

Conceptually, the first alternative from the list above is the cleanest, as it most closely resembles the traditional definition of a coprocessor (cf. Figure 4.2). If the FPGA is attached to the CPU via a coprocessor interface (Alt. 1), then it can access memory only through the CPU. If it is attached to the memory/processor bus (Alt. 2), then it can access memory directly.

Both alternatives have one major advantage, namely, that the latency of data transfers between the CPU and the RC is as small as possible. Lower latency can only be achieved by incorporating the FPGA directly on the CPU chip [BRA96, DeH96, ECF96, Raz94]. It is a good setup when an RC is used to implement small statement sequences (within a software

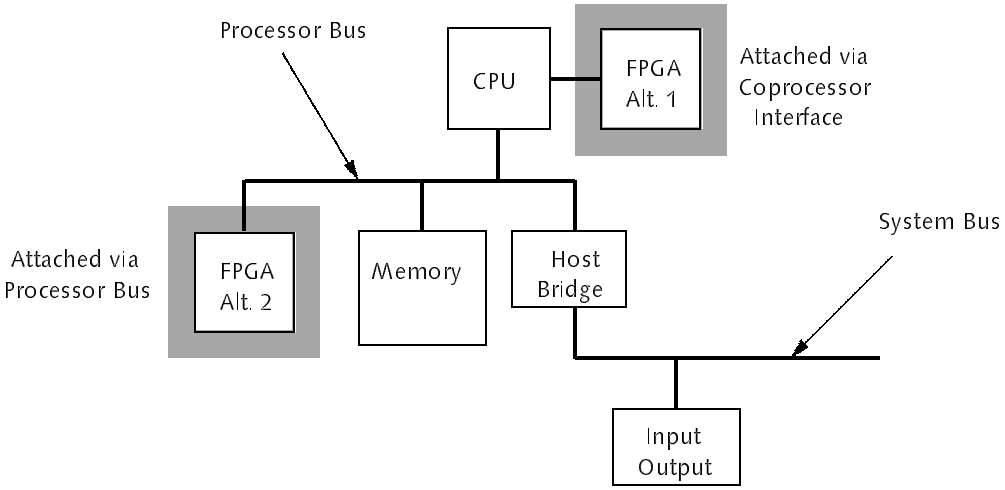

Figure 4.2: FPGA Attached to the CPU (Two Alternatives)

loop), where just a few words of data need to be exchanged between the CPU and the RC. If required, the RC has access to main memory, just as the CPU, and can easily share data with a software application.

But there are also several problems with the two alternatives:

- Alt. 1: If the FPGA is attached to the CPU via a coprocessor interface, then a speedup gained by using the FPGA could be offset by the lack of parallelism between the CPU and the FPGA, as the former must be used to move data in and out of the FPGA.
- Alt. 1: The design of the RC is not portable to different CPUs, and not even to different memory systems using the same CPU.
- Alt. 1: Such a setup would require the construction of a new processor board, a task which was beyond the scope of this work.
- Alt. 2: If the FPGA is attached to the processor bus like normal memory, then arbitration logic must guarantee that only one master is active and memory would not be available at all times due to refreshes. Also, modern memory systems tend to be very complicated for achieving the high speeds needed to fill processor cache lines quickly. Hence, they do not perform equally well when smaller amounts of data are transferred or when the memory access pattern is random.
- Alt. 2: Like any multiprocessor system, concurrency between the RC part and the CPU part of an application is hindered by the additional competition on the memory bus. Depending on the access pattern and the presence of a second level cache, this problem can be very severe, i.e. the memory bus can be saturated quickly. The speed of applications executed on fast CPUs are often dominated by the memory access time, not by the time of computation. That is, the CPU spends most of its time waiting for the memory system instead of computing [Con96].
- Alt. 2: Another problem when lacking a dedicated coprocessor interface is that logic in the FPGA would have to be used to implement the communication signals between the CPU and the FPGA. This logic might be too slow when interfacing to a fast CPU.

Despite these problems, the idea of an FPGA tightly coupled to the CPU looks attractive, and we would like to pursue this in the future (cf. Chapter 8), specifically by using a slower (and cheaper) CPU with a simple memory system.

4.2.2 Extension Card with DMA Only

The second alternative from the list above is an RC board that contains an FPGA and DMA logic for quick data transfers from and to memory. Such a setup is shown in Figure 4.3 — the local memory shown has to be ignored for the discussion in this section. The DMA option allows an RC application to access main memory. One limitation of this setup, however, is that all data transfers have to use the system bus. If only small transfers are made, the control overhead deteriorates performance.

Depending on the target system bus, such a card is relatively easy to build, as there is a well defined interface. On the downside, building a board, which can act as a *bus master* can be quite complex, since bus arbitration is needed and additional control signals have to be generated.

4.2.3 Extension Card with Local Memory

The third alternative overcomes the shortcomings of the solution above in that the RC board is equipped with local memory. The data to be processed by the FPGA can be transferred from the source (main memory or IO board) into local memory on the RC in one large chunk using programmed IO or DMA. Hence, the RC and the CPU can pursue work concurrently after the transfer and the RC is not loading the system bus with read requests.

This setup is very common for RC boards nowadays. It decouples the RC from the rest of the system. Thereby, the control part of an application on the host side can remain small. Additionally, since the card has local memory, it can act as a smart input preprocessor, performing some filtering operations on incoming data, before sending the data to the host [VBR96]. The data might be transferred to the RC via DMA from an IO board or there might be a connector for hardware extensions included on the RC.

As with the solution in Section 4.2.2 the hardware for this card is relatively easy to build due to the well-defined interface to the system bus. The DMA control in Figure 4.3 could be left out, as the FPGA with its local memory can act as a stand-alone computational unit, which has to communicate with the host only rarely.

4.3 Overview of the Hades Reconfigurable Coprocessor

The Hades hardware consists of an *extension card* for the *Ceres-2 workstation* (cf. Section 4.4). It features *one XC6216 FPGA* and *256 KB of fast SRAM*. An *address decoder* is realized with *three 22V10 PALs* [AMD95, Cyp95]. The card is *accessed like conventional memory* using address and data buses and read/write control signals. A schema of the coprocessor board is shown in Figure 4.4. The complete board layout and a picture of the board is shown in Appendix C.

4.4 Choice of Host Workstation

When we had to decide on a host computer for the Hades RC board, the choice was between a commercial architecture (like a PC, Sun or Macintosh) and the Ceres workstation, which was developed at the Institute for Computer Systems. Our Institute has a long tradition in building its own workstation hardware [Ebe87, Hee88, HN91, Ohr84]. The Ceres-1, Ceres-2 and Ceres-3 computers have been used since 1987 in research and education. An experimental workstation named Chameleon used an array of six CAL-1 [Alg90, Kea89] chips for a reconfigurable coprocessor and a single CAL-1 for all control logic [Hee93, HP92]. For a digital design course, a small extension board for the Ceres-3 (the machine used in education) based on the Concurrent Logic 6000 FPGA was developed [GLW94].

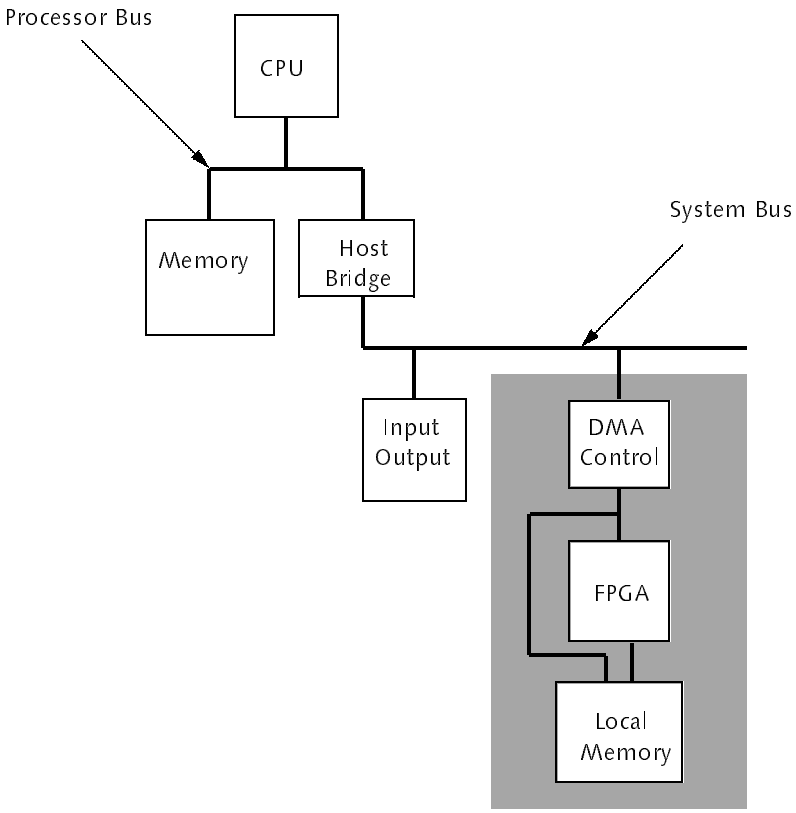

Figure 4.3: FPGA with Local Memory on Extension Card

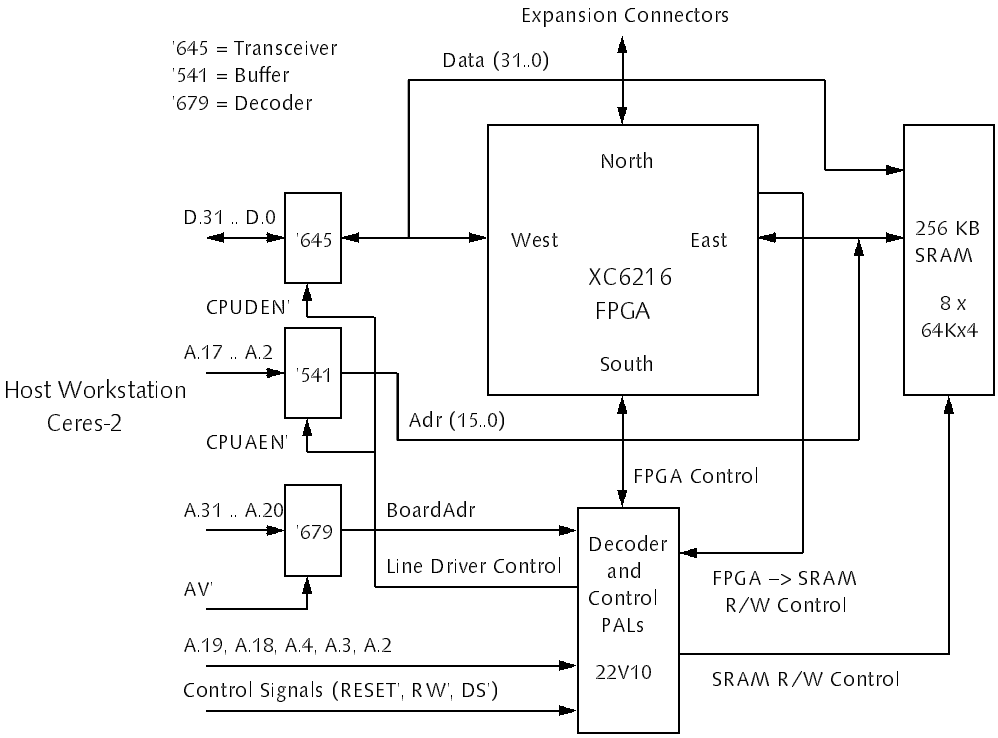

Figure 4.4: Hades Reconfigurable Coprocessor

We developed the Hades RC board for the Ceres-2 [Ebe87, Hee88], the second generation Ceres workstation, which has the following characteristics:

- 1985/1988 technology
- National Semiconductor NS32532 CPU @ 25 MHz (40 ns cycle time) with 512 byte of shared instruction and data cache [NS88]
- proprietary bus with arbitration
- 4 or 8 MB DRAM
- 1 memory cycle equals 6 processor cycles (240 ns)
- memory transfer rate of 12.5 MB/s (as seen by applications)
- 80 MB harddisk
- disk transfer rate of 120 KB/s (as seen by applications)
- Oberon operating system with all source code available

The reasons for choosing the Ceres-2 as our host platform were as follows:

- the Hades RC board looks like a normal memory card to the Ceres-2,
- the Ceres-2 has a 32-bit wide data bus, which delivers satisfactory performance for our purposes,
- it is used in our group as the main computing platform,

- its architecture is well understood,
- numerous extension boards have already been produced (color display controller, Ethernet interface, audio/video interface),
- all maintenance can be accomplished at the Institute,
- the Oberon operating system was written by members of the Institute, and
- it has a simple bus protocol.

At the time we started our project, no RC board existed that used the XC6200. To gain experience with the Trianus/Hades tool set as soon as possible, availability of such a board was more important than performance or compatibility with commercial systems. Therefore, we chose the Ceres-2 as our target platform rather than a commercial architecture. Furthermore, we knew of other people working on a PCI board [LSC96]. Clearly, to gain wide-spread acceptance, a second generation Hades RC should be implemented as a PCI card with corresponding driver software or the Hades software should be ported to a commercial board. See Chapter 6 for a comparison of the PCI card [LSC96] with our Hades RC.

4.5 Architecture of the Hades Board

The Hades RC board shown in Figure 4.4 consists of a single XC6216 FPGA in a 299 pin grid array package [Xil96] and 256 KB of local memory implemented with fast SRAM (8 Motorola 64Kx4 bit SRAMs [Mot95]). The memory-card interface is realized with three 22V10 PALs [AMD95, Cyp95], which generate the necessary control signals for the XC6216 and the SRAMs. See Appendix D for a complete list of hardware components used on the board.

The programming and access interface to the XC6216 is that of a conventional SRAM (cf. Section 2.3). Therefore, the coprocessor board looks like a memory board to the CPU. Accessing the XC6216 takes the same amount of time as accessing "normal" memory (240 ns).

The board features a 16-bit wide address and a 32-bit wide data bus to access the XC6216 and the local memory (Data and Adr in Figure 4.4). 16-bit addresses and 32-bit data can address 256 KB of local memory. This memory could hold, for instance, three 320x256x8-bit images or 1.5 seconds of stereo CD-quality audio data.

Although we could have used the XC6216's capability to generate the control signals needed by the CPU to access the XC6216 [Xil96], we decided against this option as we did not have any software to generate the configuration bits nor hardware to test the decoding circuitry itself at the time. Instead, we use three 22V10 PALs for interface and control logic (Decoder and Control PALs in Figure 4.4).

Two expansion ports are provided, to allow for hardware extensions. Series resistors protect the FPGA pins connected to these ports from possible damage caused by high currents.

4.5.1 Host Interface

The Ceres-2 bus is clocked at 25 MHz. One memory cycle takes six clock cycles. The data bus is 32-bit wide, therefore a peak throughput of 25/6 MHz * 4 Bytes = 16.7 MB/s could be achieved. Using the CPU only, a transfer rate of 12.5 MB/s can be achieved in practice, by today's standards a truly antique value.

Program 4.1 lists the Lola code for the decoder chip controlling the XC6216 FPGA on the Hades RC board. Subsequent sections refer to the variable names defined in this program. Appendix E additionally lists the Lola code for the RAM control PAL and the code for the PAL implementing communication ports.

Program 4.1 Lola Code for FPGA Control PAL

```
TYPE DecoderXCCtrl;                          PAL22V10
    IN
        Clk: BIT;
        BoardAdr': BIT;                      board is selected
        A19, A18, A4, A3, A2: BIT;           address lines needed for decoding
        CPURW', CPUDS': BIT;                 CPU: read/write, data strobe
        RESET': BIT;                         master reset
    OUT
        XCCS', XCOE': BIT;                   XC: is selected, may drive pins
        XCAOE', XCDOE': BIT;                 XC: may drive A/D buses
        XCReset', XCGClr: BIT;               XC: reset, global clear
        CPUAEN', CPUDEN': BIT;               CPU drives the A/D-bus
        XCStep: BIT;                         single stepping
    VAR
        select, write, XCSel, RAMSel, PortSel: BIT;
        oe: BIT;                             register
BEGIN
    select := ~CPUDS' * ~BoardAdr';
    write := select * ~CPURW';
    XCSel := select * ~A19 * ~A18;           00'xxx
    RAMSel := select * ~A19 * A18;           01'xxx
    PortSel := select * A19 * A18 * ~A4;     11'0xx

    XCCS' := ~XCSel;
    10'000 disable, 10'001 enable OE'
    oe := REG(write * A19 * ~A18 * ~A4 * ~A3, A2));
    XCOE' := ~oe;

    10'010
    XCReset' := ~(write * A19 * ~A18 * ~A4 * A3 * ~A2) * RESET';
    10'011
    XCGClr := write * A19 * ~A18 * ~A4 * A3 * A2;
    10'100, generate one clock pulse, high -> low -> high
    XCStep := ~(write * A19 * ~A18 * A4 * ~A3 * ~A2);

    CPU uses data bus
    CPUDEN' := ~(XCSel + RAMSel + PortSel);
    CPU drives address bus
    CPUAEN' := ~(XCSel + RAMSel);

    XC may drive data bus
    XCDOE' := ~(oe * ~XCSel * ~RAMSel * ~(PortSel * ~CPURW'));
    XC may drive address bus
    XCAOE' := ~(oe * ~XCSel * ~RAMSel)
END DecoderXCCtrl;
```

Line Drivers and Address Comparator

The host interface comprises four bi-directional line drivers ('645 in Figure 4.4) for interfacing the 32-bit data bus, two uni-directional line drivers ('541) for driving the 16-bit address bus, a 12-bit address comparator ('679) and three PALs (22V10) realizing decoding and control logic. Five address lines (A.19, A.18, A.4, A.3, A.2) are connected to the decoder directly, defining different *regions in the address space* as defined in Table 4.1. The listed addresses are relative to a base address, which is determined by the address comparator.

The direction of the data line drivers is controlled by signal CPURW' of the Ceres bus (see Figure 4.5 for timing information). The enable signals for the address and data line drivers are generated by the decoder (CPUAEN', CPUDEN'). The address comparator requires the 12 address lines to be in a specific sequence, namely address lines that are one followed by lines that are zero when active. In our case, the RC board lies between FEC00000H and FECFFFFFH, so the highest 12 bits of the address to decode are FECH. The sequence A.31 .. A.25, A.23 .. A.20, A.24 is used to decode this address range. A problem resulting from this requirement is described in Section 4.6.4. The comparator's enable signal is the AV' (address valid) signal of the Ceres bus (see Figure 4.5). The comparator will assert signal BoardAdr' whenever a correct address is on the address bus and signal AV' is asserted.

The host is given priority over the XC6216 when accessing the address and data buses of the RC board. For this purpose, the decoder drives two output enable signals read by the XC6216 (XCAOE' and XCDOE' for the address and data buses, respectively). A user configuration in the XC6216 may not drive the address- and data-buses when the respective enable signals are not active.

Memory Map

The different memory regions selected by the decoder are shown in Table 4.1. The function-

No.	Address (A.19..A.0)	Functionality
1	00000 - 3FFFF	FPGA (configuration + register access)
2	40000 - 7FFFF	SRAM 256 KB local memory
3	80000 - 80003	FPGA output disable (XCOE' = 1)
4	80004 - 80007	FPGA output enable (XCOE' = 0)
5	80008 - 8000B	FPGA reset (XCReset' = 0)
6	8000C - 8000F	FPGA global clear (XCGClr = 1)
7	80010 - 80013	single step clock (XCStep = pulse)
8	80014 - 80017	Go/Busy flags
9	80018 - 8001F	reserved
10	C0000 - C0003	general purpose port 0 (R/W)
11	C0004 - C0007	general purpose port 1 (R/W)
12	C0008 - C000B	general purpose port 2 (R/W)
13	C000C - C000F	general purpose port 3 (R/W)
14	C0010 - C001F	reserved

Table 4.1: Memory Map of Hades Board (Address is Relative to a Base)

ality of the different regions are used as follows:

1. Programming the XC6216 and access to the registers within the FPGA.
2. Access to on-board memory. Individual byte accesses are controlled by the byte enable signals BE'.0..3 [Hee88] and RAMWE'.0..3 (cf. Appendix E).

3. Disable Output Enable signal of the XC6216. This is useful to regain control over the buses, if a faulty configuration in the FPGA is driving them.
4. Enable Output Enable.
5. Reset the FPGA via the Reset' signal.
6. Clear the register in the FPGA via the GClr signal.
7. Issue a single clock pulse.
8. Asynchronous communication. A write to this region sets or clears the Go flag in the decoder, depending on the value of data bus bit 0. This flag can be read by the FPGA. A read from this region returns the value of the Busy flag on data bus bit 0. The Busy signal is driven by the FPGA.
9. Reserved.
10. General purpose read and write port (GPP). Individual read and write signals are generated from CPURW' and the values of A.3..2. This region is for port 0.
11. GPP 1.
12. GPP 2.
13. GPP 3.
14. Reserved.

4.5.2 XC6216 Interface

As shown in Figure 4.4, the west side (left) of the XC6216 FPGA is connected to the data bus, and the east side (right) to the address bus. This is demanded by the pinout of the chip. The south side (bottom) connects to the control signals and the north side (top) can be used freely via the expansion connectors. The XCCS' (chip select) signal of the FPGA is asserted by the decoder whenever an access to the FPGA is made (address space 1 in Table 4.1). The XC6216's read/write signal is driven by the R/W' signal of the Ceres bus. The XCOE' (global output enable) of the XC6216 is driven by a register in the decoder (oe), which can be set or reset by writing to address regions 3 or 4. This is a last measure against the case where a (faulty) configuration in the XC6216 is driving the data-bus pins, which would prevent the host from being able to reprogram the chip. The XCReset' (global reset) and XCGClr (register clear) signals of the XC6216 can be asserted with writes to address space 5 and 6, respectively. The Serial' and Wait control signals must be connected to power and ground [Xil96], respectively, since the XC6216 is configured in parallel mode only.

There are four global signals on the XC6216. GClk, G1, G2 and GClr. The various clock signals are explained in Section 4.5.4 below. The GClr signal is driven by XCClr' of the decoder.

4.5.3 Interface Timing

Figure 4.5 shows a timing diagram of the Ceres-2 bus signals (top) and, derived from them, the chip select signal (XCCS') for the XC6216 (bottom). In addition, the timings of a data write to the FPGA and a data read from the FPGA are shown. DOut/DIn on the Ceres side correspond to DRead/DAvail on the XC6216 side. Additional timing information for the Ceres-2 can be found in [Hee88] and for the XC6216 in [Xil96].

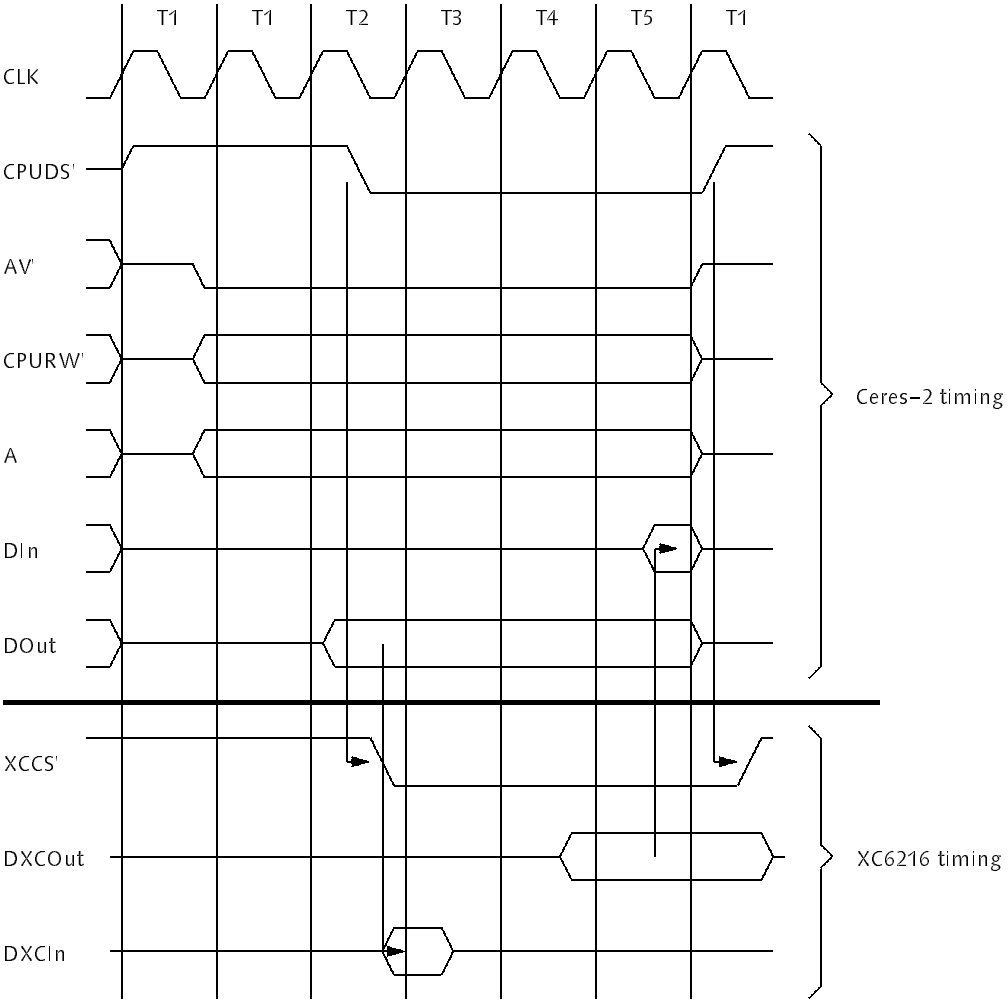

Figure 4.5: Interface Timing

A read or write *access to the XC6216* takes at least *two clock cycles*. The *interface is synchronous* such that XCCS', CPURW', A and DOut are all sampled on a rising clock edge. An access is initiated by XCCS' being sampled low (start of T3 in our case). An access terminates, if XCCS' is sampled high at the second clock cycle. It can be extended by keeping XCCS' low during that cycle, in which case the access terminates asynchronously as soon as XCCS' goes high.

We use these *extended access cycles*, as the CPUDS' signal remains low during three full clock cycles (T3, T4, T5). Signal CPUDS' goes low during T2 and accordingly does XCCS'. XCCS' is sampled synchronously at T3 (together with the other aforementioned signals). This starts the access cycle. At the start of T4, XCCS' is still sampled low, initiating an extended access cycle. The access is terminated asynchronously when XCCS' goes high during the next T1 cycle. This is no problem, as there is enough time before XCCS' could go low again during the next T2 cycle.

When the FPGA is written (DOut -> DXCIn) during a write access, the bus is sampled synchronously at the start of T3. During a read access (DXCOut -> DIn), the FPGA provides the data on the bus during the extended access cycle until XCCS' goes high again (T4, T5, T1). It is sampled by the CPU before the end of T5.

4.5.4 Clocks

Three clocks are provided as global signals in the XC6216. The main clock used by the XC6216 (GClk) is the Ceres clock running at 25 MHz. This guarantees the correct operation of the processor interface and also avoids the need for additional synchronization registers between the Ceres bus and the XC6216. The G1 signal is connected to a 40 MHz oscillator which is mounted on a socket on the board. It can be exchanged with slower or faster oscillators. The G2 signal is connected to the XCStep' output of the decoder PAL. It is used for single stepping and can thus be used to debug an application. A single step can be generated by writing to address space 7.

The three clocks proved sufficient for most applications. Either the main clock GClk or the single stepping mechanism on G2 was used. The fast clock on G1 was only used to test the performance of some basic circuits (counters and adders). If an application demands a slower clock, the main or the fast clock must be divided in the XC6216. This is, however, quite inconvenient, as it introduces additional logic which must be placed at a certain location inside the chip, interfering with other logic and complicating the placement task. In retrospect, a more sophisticated clocking scheme based on an external programmable clock generator would have been advantageous.

Care was taken when routing the clock signals. It was ensured that they run in a single line across the board, without branches which would cause reflections [JG93]. The main clock GClk coming from the Ceres-2 connector first goes to the decoder PALs, then to the FPGA, then to the expansion connector and is parallel-terminated by a 330/270 Ω resistor pair. Likewise, the fast clock comes from the oscillator, goes to the FPGA and then to the expansion connector and is terminated by a similar resistor pair.

4.5.5 Host – Coprocessor Communication

Normally, communication is implemented using *direct register reads and writes* using the XC6200's processor interface. This is the most flexible way, as it allows an arbitrary number of "flags" to be implemented. The pattern matcher application in Chapter 6 is an example using this method, where the state machine controlling the application is modified using register writes.

Two additional, simple schemes were devised to control communication between the host application and the application running on the coprocessor:

The first scheme uses *two status bits in the decoder*, one to initiate a computation (Go) and one to signal its completion (Busy). The Go flag is set or cleared by writing a one or zero to address space 8 (cf. Table 4.1). The Busy flag is inspected by reading from address space 8. Its value is determined by the XCBusy signal generated within the XC6216. These bits free up the data bus and allow for communication with the RC even when the data bus is used by an application (for instance to access local memory).

The second scheme enables *independent operation* of coprocessor and host by *using interrupts*. A computation is initiated with the Go flag, but its completion is signaled with an interrupt. The XC6216 can assert an interrupt line (INT7) and a software handler in the user application can react to this signal.

4.5.6 Local Memory

The 256 KB of local memory are implemented with eight 64K x 4 bit, 15 ns SRAM chips [Mot95]. The chips have separate output and write enable signals, where the write enable overrides the output enable. The decoder derives separate write enable signals for each byte from the host's byte enable and from the XC6216's write enable signals. The high speed of 15 ns is needed to enable a 40 ns memory cycle (25 MHz clock) when accessing the SRAM. The maximum access time to local memory can be calculated as shown in Table 4.2, all values being worst case. The decoder PAL is included in the calculation, as the write and output enable signals for the SRAM are generated by that PAL.

Signal	Delay (ns)
Clock to Function Output	2.5
Neighbor to Pad	6.5
22V10	7
SRAM Access	15
Pad to Neighbor	5
Register Set-Up	3
Clock Skew	0.9
Total	39.9

Table 4.2: Worst Case Access Time to Local SRAM

An access time less than 40 ns is important, such that it is possible to switch between reading and writing the SRAM in consecutive clock cycles. Higher speeds can be achieved when data is only read or only written: during reads, the RAMOE' control signal is kept enabled and only the addresses are changed; during writes the RAMWE' control signal is toggled and address and data are changed.

4.6 Constructing the Board

In the following, we give a description of the construction process of a printed circuit board. Experienced hardware designers may skip this section, but readers new to the field may find some interesting information herein. For a more detailed experience report see also [Lud96].

4.6.1 Describing the Printed Circuit Board

Once the design of the RC was fixed on paper, CadStar [ZR95] was used to describe the board in a schema (see Figure B.1 in Appendix B). Based on this, a printed circuit board (PCB) with

two signal and separate power and ground layers was defined. A Ceres-2 expansion board is an extended double Eurocard (233 x 220 mm^2), so there was ample space to place components. This was done manually and care was taken that all dual-inline packages were facing the same direction (such that power and ground pins were all at the same position, which eases debugging). We decided to use sockets for all chips and no surface mount technology was used. After successful placement, the board was routed automatically. A pin swap optimization step reduced wire length by more than 10%, but also caused a problem on the final board, as described in Section 4.6.4. A PCB manufacturer produced four prototype boards.

4.6.2 Describing and Implementing the Decoder

The decoder's logic equations were defined in the Lola HDL (cf. Section 3.2). We simulated the decoder thoroughly before translating the Lola code into the CUPL language [Log91] used by the tools that generate the Jedec files (PAL fuse maps). Logic partitioning onto the three PALs and pin assignment was done manually. The Lola code for the complete decoder is listed in Program 4.1 and in Appendix E. A PAL burner was used to program the PALs based on the Jedec files.

4.6.3 Assembling the Board

After receiving the boards we tested them for short circuits. The connections to the various chips – especially power and ground – were checked manually using a multimeter. During this testing, we found a net which was not correctly connected (cf. Section 4.6.4).

After soldering the sockets for all chips and the connectors, the board was inserted into a Ceres-2, which booted without problems. One after the other, the line drivers, the decoder PALs (which functioned right away) and the SRAMs were added to the board. We tested the relevant signals of each newly added component. Finally, the FPGA was inserted as well and the first design was downloaded.

4.6.4 CAD-Software Pitfalls

To reduce the wire length of nets, board routers perform a process called *pin and gate swapping*. Pins of a chip can be exchanged with each other if they implement the same function. For instance, the outputs of a transceiver chip ('645) can be swapped, as long as the inputs are swapped accordingly. Whether certain pins can be swapped or not is described in a device description file (in the case of CadStar). Due to an error in our description of the '679 address comparator, the router "optimized" address line A.24 by moving it to its "natural" position between A.25 and A.23 (cf. Section 4.5.1). The wire length was reduced, since a crossover was removed, but the decoded address was wrong. Cutting the nets on the PCB by hand and inserting two patch wires solved the problem, however in a not very aesthetic way. The lesson learned was: double- and triple-check the final plots of your board before sending them to the manufacturer!

4.6.5 Power Consumption

A fully populated Hades board with 8 SRAM chips, 3 decoder PALs and an (idle) FPGA draws 980 mA (4.9 W) in standby mode (without being accessed). Accessing the SRAM from the host draws another 230 mA (1.15 W). An FPGA design full of registers (4096) toggling at 25 MHz adds another 300 mA (1.5 W).

For comparison purposes, the Ethernet board for the Ceres-2 draws 850 mA (4.25 W) in standby mode (without being accessed), so the standby current used by Ceres-2 extension cards seems to be quite high.

4.7 System Software

The Hades board occupies 1 MB in the address space of the host's operating system. This address space must not be cached, otherwise, changes in the FPGA or in the on-board memory are not observed by the processor. The Ceres-2 runs the Institute's *Oberon operating system* [WG92], whereby we were able to modify the kernel to provide the needed address space. More details on the bitstream generator and the hardware/software interface is given in Section 5.7 and 5.8.

4.8 Discussion

Adequate Testbed

The board has proved its value for implementing and testing coprocessor applications. See Chapter 6 for a discussion of an application using the board and a presentation of performance data. The performance of the Ceres-2 and its bus was adequate and competitive with a prototype PCI-card we received from Xilinx [LSC96], which had to be accessed using slow input/output commands.

A Software Person Can Build Hardware (With Some Help)

The Hades RC board was our first hardware project. It proved advantageous to have a simple, conservative design, which was intellectually manageable and overseeable at all times. We spent more time learning the CAD tools than actually designing or constructing the board. With some help from hardware experts it was possible for a software-oriented person to conceive, describe and implement an FPGA-based coprocessor board in three months.

Interfacing to the XC6200 FPGA

Interfacing to the XC6216 is quite easy. The control logic for generating the two necessary signals (CS' and RW') of the synchronous interface is simple and easily built (depending on the complexity of the host bus protocol, of course).

Operating System Issue

The availability of an operating system which we could change was an invaluable advantage. When discussing such issues with other researchers, we always heard complaints about the difficulties of writing driver software for the respective host operating system. The downside of the medal, of course, is that such a setup is only possible in a research environment.

Months Later

Just one and a half years later, we have gained enough experience to suggest different design alternatives. One would be to include a programmable clock chip. The other would be the use of synchronous SRAMs to implement local memory. Also, implementing the decoder in one large PAL (like a MACH211 [AMD95]) or an ispGAL [Lat96] would ease maintenance. Today, we would also feel confident to target a more sophisticated bus, such as PCI, which would improve performance of data transfers to and from the host considerably at the cost of more complicated driver software required by commercial operating systems.

5 Hades Software

In this chapter, we present the *Hades software*. It consists of layout synthesis software comprising a *technology mapper*, a *placer*, a *floor planner* and a *router*. In addition, a *bitstream-generator*, a *loader* and a *runtime system* are provided (cf. Figure 5.1). The Hades software is based on the Lola hardware description language and the Trianus framework for digital circuit design (cf. Chapter 3 for an introduction). An overview of the Hades software is given in [GL96].

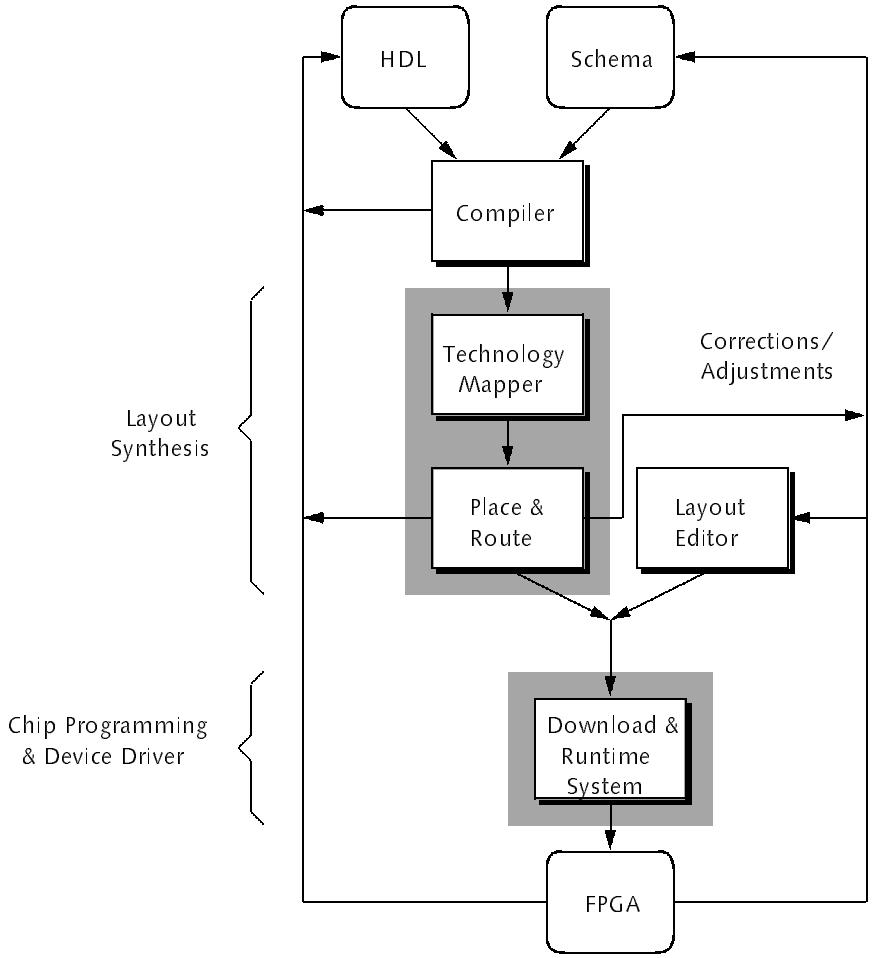

Figure 5.1: Hades Software Part within the Design Flow of Fig. 1.3

5.1 Problem Statement and Motivation

Digital circuit design is a difficult, error-prone task. Like any other engineering discipline, *abstraction and modularization* are the key to a successful completion of a system. Only through abstraction is it possible to intellectually manage a digital circuit containing hundreds to possibly millions of functional units. The aid of a computer during the design and implementation of a digital circuit is mandatory to manage the hierarchical data structures involved (see [Geh97] for a discussion of CAD frameworks).

As was presented in Section 1.4, the term *hardware synthesis* describes the process of generating an implementation of a digital circuit based on a specification, which exists in the form of a schema or a program written in an HDL. Hardware synthesis is composed of *logic synthesis* and *layout synthesis*. The former is the problem of finding functional units (gates), which implement the specification. The latter is the problem of fitting the functional units into the available resources of the target device, be it custom logic, standard cells or an FPGA.

5.1.1 Layout Synthesis

This chapter treats the subject of layout synthesis. Our starting point is a *Trianus data structure*, also called a *netlist*. This netlist has been generated by the Lola HDL compiler or a schema editor. The task is to implement the circuit represented by the netlist on an instance of the XC6200 FPGA architecture. There are several degrees of freedom on how this circuit can be implemented:

- *Technology Mapping*: how are the basic gates of the netlist mapped onto the available cells of the FPGA?
- *Placement*: how are the mapped cells arranged on the rectangular grid of the FPGA?
- *Routing*: how are the placed cells interconnected using the available wiring resources of the FPGA?

5.1.2 Intractability

The problems above are not independent from one another. Routing depends on the placement of cells, which in turn depends on the mapping of gates onto cells. In general, the problem to find the optimum solution for any of the above tasks is *NP-hard* [Coo71, GJ79, Kar86]. Because of the intractability of solving these problems optimally, layout synthesis tools often use *heuristics*, which approximate an optimal solution as closely as possible. Aside: the same statement holds for most optimization problems; for example, logic synthesis is another NP-hard problem.

5.1.3 Current Approaches

We give a short overview over current approaches in all of the three problem domains. A longer introduction can be found in [Pfi92]. All these problems are normally solved using heuristics, which are not guaranteed to find the global optimum.

Technology Mapping

As was stated above, *technology mapping* is the task of assigning the function units in a netlist to the available cells of the FPGA. Consider the problem of mapping the full adder of Section 3.2.5 to the cells of the coarse-grained XC4000EX, and the fine-grained AT6000 and XC6200, respectively. Ideally, the three input, two output full adder fits into one cell (CLB) of the XC4000EX and makes use of the fast-carry logic. On the AT6000 the carry logic is

implemented using NAND-gates, a full adder taking up three logic and one routing cell. On the XC6200, an ideal implementation of a full adder uses two XOR-gates and a multiplexer, hence three cells.

Most current technology mappers use a *graph-based approach* pioneered by [Keu87] and further developed by [DGR87, FRV91, BFR92]. The netlist is represented in a canonical, technology-independent graph structure. The various possible cell configurations are also represented in a similar structure. By using *graph matching*, a minimal cost cover of the netlist graph using cell configuration graphs is calculated. Obviously, as there are many ways how to cover the netlist using subgraphs, this process can take a long time. Many different mappings have to be evaluated to find the "best" one, which is the result of a dynamic programming algorithm or some heuristic. The advantage of this approach is that it is technology-independent. The same mapping algorithm can be used for various FPGA devices and also for other target technologies such as standard cells. To support a new architecture, only a new graph representation has to be defined.

Another approach is to *directly map* the netlist to the available FPGA cell configurations. The mapping of netlist gates to cell configurations of the FPGA is determined by the algorithm itself. The mapping is *constructed*. This approach is *efficient* when the target cells are simple, and more complex for more complicated cells such as the CLBs of the XC4000EX family.

Hence, graph-based mapping is used for coarse-grained cells and direct mapping is used for fine-grained cells. Consequently, the mapper for the XC6200 described in Section 5.4 uses the direct approach.

Placement

Placement is the task of arranging the mapped cells onto the grid in the FPGA. For placing the full adder from the example of Section 3.2.5 onto the XC6200, the task would be to place the two XOR-gates, the two AND-gates and the OR-gate. How are they placed relative to each other? How is a group of full adder elements — making up an N-bit wide adder — arranged? How are bigger functional units, like an ALU or a state machine placed with regard to each other and with regard to I/O pins, connecting the functional units to the outside world? There are several approaches to tackle these problems.

Constructive placement: this is a very fast placement method. An overview of several algorithms is given in [HWA76]. Constructive placement deterministically arranges the cells according to a programmed "recipe". For instance, a two-input cell might be placed where space is found, then the two sources feeding these inputs are placed to the right of the cell, and to the right above. The placer described in Section 5.5 uses this approach.

Min-cut: this is a graph partitioning method, trying to minimize the number of connections (net cut) between two subgraphs while keeping the sizes of the subgraphs approximately the same [KL70, Bre77, FM82, Kri84]. It starts with an arbitrary partition of the netlist and swaps gates between the partitions, if this reduces the net cut. The swapping is repeated until no further reduction can be achieved. A linear time algorithm using a good heuristic is presented in [FM82] and an improved version is presented in [Kri84]. The total runtime of the algorithm depends on the number of partitions the netlist is divided into.

Simulated annealing: this is the most popular — and successful — approach to placement [KGV83]. It is based on a model from physics. Annealing is the process by which molecules in a hot gas arrange themselves as the gas is cooled down. The arrangement minimizes the energy stored in the gas. Simulated annealing cools down the "gas of cells". In the beginning, the cells float around freely and exchange positions with other cells. A grading function, e.g. based on the total length of wiring, is evaluated for a given configuration. If the grade for the current configuration is better than for the previous one, the current configuration is accepted. If the grade is worse than before, however, the configuration is only accepted with a certain probability. This allows the algorithm to leave local minima and is the key to

finding a placement which is close to the global minimum. After each evaluation step the temperature is lowered, i.e. the area within which a cell may float around is made smaller. Simulated annealing gives very good results, but its problems are long runtimes and — due to its randomness — non-deterministic solutions. One placement run may give a completely different result from a previous one, despite being produced from the same input. This can be a real problem when iterative design methods are used or when device utilization becomes very high and manual support is needed. In these cases, non-deterministic placement algorithms make things harder for the designer than necessary.

Min-cut and simulated annealing yield the best results. Constructive placement is the fastest. Certain placement algorithms combine several of the above approaches and use the best result. This process, of course, takes even more runtime.

Routing

Routing is the process where placed cells on a rectangular grid are connected together using the available wires (also called routing resources). Taking the example from Section 3.2.5, the router has to connect the OR-gate of the carry out with the two AND-gates, which in turn have to be connected to the XOR-gate and the carry input coming from the OR-gate in the previous adder element.

Routing resources can be put into two categories: channels and switchboxes. A routing channel is a region free of logic cells. The cells lie on either side of the channel. Wires run from one side of the channel to the other, possibly crossing over each other, connecting cells on one side with cells on the other side (see Figure 5.2). A more general routing structure is the switchbox, which connects wires from different sides. While routing channels are often found in standard cell technology, switchboxes are typical for FPGAs.

In Figure 5.2, a *channel router* connected the top row of cells to the lower row of cells. For a given width, the channel router uses as high a channel as is necessary to perform the routing. These routers perform not so well in FPGAs, as there is a multitude of available routing resources, not just channels.

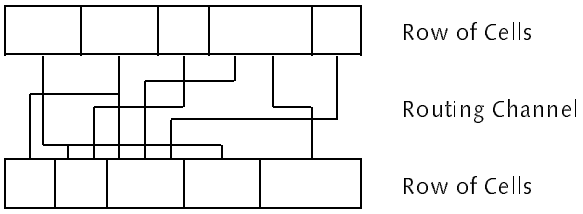

Figure 5.2: Routing Channel

A *maze-running router* [Lee61, Hig69] spreads a wave from the destination (D) of a signal in all directions until the wave reaches the source (S) signal (cf. Figure 5.3). Depending on the implementation, the wave starts from the source or from both ends. Also, the shape of the wave can be influenced in many ways, such that only horizontal or only vertical wires are used, or that expansion in one direction is more “costly” than in another direction. A maze-runner represents the most flexible routing algorithm. Its main problem is the long runtime, as the time required to spread a wave in all directions grows with the square of the distance.

Despite this, the *maze-running router* is still the most popular routing algorithm due to its flexibility. It can be accelerated by several techniques, so that the quadratic behavior affects the routing of only a few nets [Dio87, DM95]. The router described in Section 5.6 is a maze-running router.

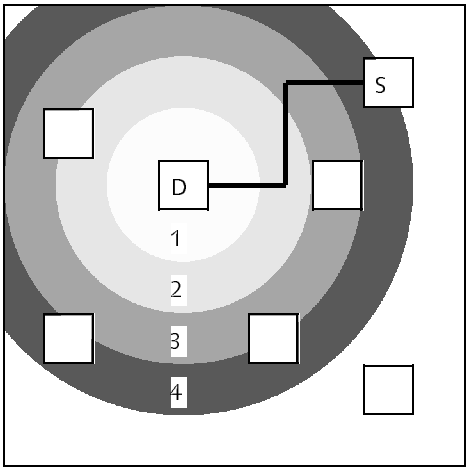

Figure 5.3: Wave Expansion (1, 2, 3, 4) with Resulting Route

5.1.4 Problems with Today's CAD Tools

Speed

Many algorithms used in today's FPGA layout synthesis tools were adapted from packages used for ASIC design. There utmost performance in both time and space of the resulting circuit is mandatory. Long runtimes (minutes to hours) of the tools are widely accepted and also most often the only way to achieve a result in the target technology. This can be tolerated when the target technology's turnaround time is in the range of days to weeks.

However, with the availability of a silicon foundry on the desktop in the form of an FPGA system, this approach is no longer viable. The engineer can now work with and develop hardware in the same way as is common in software development, namely, using iteration and exploration. If the FPGA device can be programmed in a couple of microseconds, then it should be possible to synthesize the necessary configuration bits within seconds or at least a few minutes. Sadly, with current commercial tools this is impossible [Fie95, Woo96a, WLH97].

During one day, many design cycles can be performed in software development. Good programming language compilers have turnaround times in the range of seconds [Wir96a]. With modern operating system concepts such as dynamic linking and loading, a programmer can test a module a few seconds after having made a change to the program text. This statement is at least true for the Oberon System [WG92].

One driving force in the development of the Hades layout synthesis tools was to achieve the same level of speed for the development of hardware. As can be seen in Chapters 6 and 7, this has been achieved for all parts of the synthesis flow except for routing, which is the slowest part of our software. Especially, HDL compilation, mapping and placement are very fast.

Interactivity

Interactive use of layout tools is mandatory to achieve high quality layouts of FPGA circuits [Ber93, BT94], especially with the XC6200, as current placement algorithms perform poorly on fine-grained architectures [HW96, WCG96]. Therefore, we strove for tools that the designer can *influence at all levels*. When describing a circuit with the Lola HDL, this is possible

through *position statements*; during placement, the layout editor can be used to *arrange cells and instances manually*; and during routing, *single nets can be routed* before others, nets can be *routed manually* and the designer has *control over* which *resources* may or may not be used by the router.

Integration

In software development one means for higher productivity are integrated software development tools. Pioneered by Borland in their Turbo Pascal product, today's software tools are all integrated. This means that all steps from the writing of the program code to the debugging of the program can be performed within the same environment.

This is not true for synthesis tools. Various tools from various vendors interoperate only through files. A schematic editor software produces an EDIF file which has to be read by the place and route software. The output of HDL compilers is a netlist in the form of a file, which has to be read and parsed again by back-end tools. These translation steps are costly in terms of time and memory, as files have to be written and read again. Section 3.3 and [Geh97] treat this subject in more detail.

Hades is based on the Trianus framework and achieves integration through using one central data structure.

Transparency

Current hardware description languages and associated synthesis tools often try to abstract too much from the underlying hardware. This is done, among others, for economic reasons. To support many different devices and target technologies, it makes sense to abstract from them in one way or another. *Reuse of design* is an often heard goal. In the software world, this is common practice. Programs written in a high-level programming language are portable across different hardware architectures. In many cases, one is willing to sacrifice a factor of 1.5 to 2 in performance to gain portability.

Likewise, hardware synthesis tools abstract from the target technology and map to them using elaborate algorithms. This results in slow tools even when a lower-level design style is chosen. A good (or bad) example of this is the compilation of a *structural VHDL description* of a one-dimensional discrete cosine transform taking three hours for just synthesizing the netlist. It can then be placed (using hints) and routed using the vendor's tools in just five minutes [WCG96, Woo96a]. We compiled the Lola version using Trianus in under a second.

When hardware is described in *behavioral style*, current synthesis tools often fail to achieve good layout densities or they fail to meet the performance requirements. Therefore, we believe that a *structural*, that is, a fairly *low-level*, *description* of the hardware is necessary. This should result in good performance and especially in transparency for the designer: "What You Describe Is What Is Synthesized". Note that writing target specific HDL code to achieve good performance is also recommended for VHDL, for instance by [GS95]. Using a structural style to describe the basic components, we can then compose our circuit using elements of *libraries*, much like we use procedures from modules to build software systems. The latter style then describes the hardware on a higher level, as only the functional blocks and their interaction is described, rather than the whole circuit.

5.2 Programming Methodology

The Hades software was developed using the Oberon-2 programming language [MW91] and the Oberon System [WG92]. We extensively used preconditions, assertions and (less frequently) postconditions to enhance the quality of our code and to improve the localization of

errors in the software. This proved very useful during the development of the Trianus framework and the Hades back-end. When a precondition fails, it is the user of the service who failed to meet the contract. When a postcondition of the called service or an assertion check based on the assumption that the service performed its duty fails, then it is the implementer of the service who is at fault.

Additional convenient features of the Oberon-2 language and the Oberon System are: checked array indices, NIL checks on pointers and automatic garbage collection (mandatory for extensible systems). These features are suddenly of interest to the rest of the software world — one might even dare to say: humanity — due to the advent and popularity of the Java language. All proved extremely useful and might be evidenced by the lack of a runtime debugger. A few output statements and a comfortable post-mortem trap viewer are all that was necessary to develop the Hades software successfully and efficiently.

5.3 Overview

The Hades tools consist of a technology mapper using a direct approach (Section 5.4), a constructive placer and floorplanner (Section 5.5), a maze-running router (Section 5.6), a bitstream generator and loader (Section 5.7), a hardware monitor and an interface generator (Section 5.8) for providing a software interface to a coprocessor application running on the FPGA. The tools use the Trianus framework [Geh97]. At all times they use and preserve the hierarchical design information given by the Lola HDL description. The tools support an incremental design style by way of fast turnaround times and user control during all phases.

The following sections discuss the various tools and algorithms used in their natural sequence of the design flow as shown in Figure 5.1. Their complexity in source and object code is analyzed and put into perspective. The chapter concludes with a few observations on the programming methodology used and discusses possible improvements. The tools are evaluated in terms of speed in Chapter 6.

When discussing an operation on a Trianus data structure in the following sections, we often list Oberon code to better illustrate an algorithm. This fairly low-level description is meant to serve as a tutorial on using the Trianus data structure and an aid to the reader of our source code. Using and manipulating a Trianus data structure correctly is not trivial and the more examples are given, the better this process can be understood.

5.4 Mapper

A technology mapper is a program that transforms a hardware independent description of a circuit into a form suitable for the technology available in the target device. In our case, this means mapping a Trianus data structure produced by the Lola HDL compiler onto the XC6200 FPGA architecture. Note that although we refer to the Lola compiler as being the tool to produce the data structure, it can also be produced using a schematics editor (cf. Section 3.3.6).

In a Trianus data structure, described in Section 3.3, the logic of a circuit consists of a list of named binary expression trees, where the name corresponds to variables in the Lola program. The expressions can be composed of unary or binary operators of Boolean logic (Not, And, Or, Exclusive-Or), constants (Zero, One), multiplexers, registers, SR-latches and latches. Except for the last two, a cell of the XC6200 can directly implement any of these operators. In fact, the match between the Lola constructs and the possible cell configurations of the XC6200 is almost perfect. One might suspect that Lola was designed for the XC6200, but this is not the case. Rather, the XC6200 was designed to be as flexible and simple as possible — as was Lola — and the resulting basic operators are very similar.

A mapper for the XC6200 should therefore be quite easy to develop, as almost no work has to be performed to do the mapping. It is for this reason that we have chosen a direct

approach to technology mapping, rather than writing a generic mapper using the widely used graph matching technique described in [Keu87]. Normally, a mapper determines what logic gates are implemented by what cells — hence a preplacement step is performed. Since in our case the mapping is almost one-to-one, we defer this step to the placement phase described in Section 5.5.

In the following, we will often use the terms instance and type. A type is a Trianus data structure describing a type definition in Lola. An expanded type is an instantiated type, i.e. a generic type with actual parameters. An instance is an actual hardware component of an expanded type, such as an adder element. See Section 3.2.4 and [Geh97] for further details on the difference between generic and expanded types.

5.4.1 The Algorithm

The mapper takes as input a Trianus data structure representing a compiled Lola module. It first initializes the id field of all nodes in the data structure. The id field is used to mark nodes which have been mapped already, such that already visited nodes are not visited again. The algorithm then sorts the type hierarchy *topologically*, with types not containing instances of other types occurring first. This is necessary for the correct treatment of an instance's input variables, as will be seen later. All instances of every expanded type are mapped in the sequence defined by the topological order. The mapping is accomplished by invoking a procedure for every instance and for the type itself using a *type broadcast* (cf. Section 3.3.3). The mapper is the first of the back-end tools making use of this broadcasting mechanism. Recall that the type broadcast invokes the procedure stored in the doNode field of the message for all instances of the type stored in the type field, and finally for type itself. The used procedure MapType calls MapDescendants, which traverses all signals defined in an instance or type and invokes a map procedure for them. After the broadcast, the signals occurring in the module itself are mapped. Finally, unused signals and instances are removed from the data structure. This could be done by the compiler, but was not considered during its development. It came as a natural addition to the mapping process, as every node is visited by the algorithm and its use can be determined. Program 5.1 presents the pseudocode for the process just described.

Program 5.1 Overview of Mapping Algorithm

```
PROCEDURE MapModule(module);

TriBase.InitID(module);                    unmark all nodes (and wires)
TriBase2.TopoSort(module, list);           topological sort
msg.doNode := MapType;
WHILE list # NIL DO
    msg.type := list.type;
    call msg.doNode for every instance of msg.type
    and for msg.type itself
    TriBase.Broadcast(module, msg, TriBase.SelType);
    list := list.next
END;
MapDescendants(module);
RemoveUnused(module)
```

We will now describe the necessary steps to actually map the operators of the Trianus data structure onto the XC6200. Table 3.1 listed the operators available in Lola and Figure 2.4 showed the structure of the logic cell of the XC6200 FPGA. Figure 5.4 shows the possible cell

configurations (without the optional register) as represented in the XC editor. Inputs to the cells come from left, right and below.

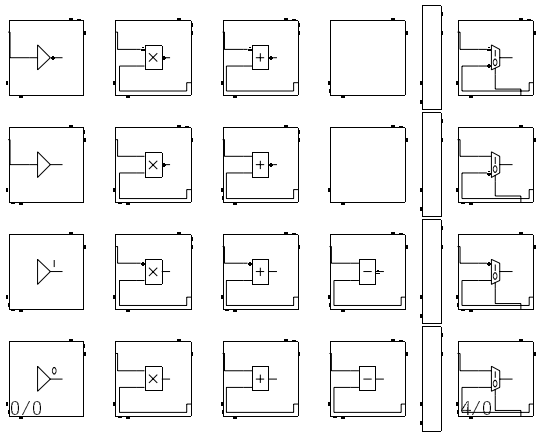

Figure 5.4: XC6200 Cell Configurations (without Registers)

For every variable or output signal node in an instance, the expression tree is traversed *depth-first* and the nodes encountered are marked as mapped. Each *operator node* requiring manipulation *is changed in-place*, that is, the data structure is changed on-the-fly. For instance, if an AND-node has to be transformed into an OR-node, the fct field of the node is changed accordingly. This saves space and time, as no additional nodes have to be allocated.

Implementation note: On some occasions (input variable and constant node duplication), a new node has to be allocated and in those cases we appreciated the existence of reference parameters in the Oberon language very much. It made the mapping procedure more elegant and easier to use as the pointer field being changed could simply be passed as a reference parameter. In a language without this feature (such as Java), an additional parameter would have to be passed indicating which field needs to be updated, or a function procedure would have to be used.

Constants One and Zero

To conserve memory space, the constants One ('1) and Zero ('0) occur only once as global variables in a compiled Trianus data structure. But since they must be represented as actual gates at several locations of the FPGA, a new node representing the constant must be generated for every occurrence in the expression tree. The pointer field pointing to the unique constant node is updated to point to a new copy of such a constant node.

Not

NOT-operators associated with a signal name (label) or which are feeding a register occur as proper cells in the layout. All others can be merged into the operators with more than one input (see the NOT in expression y below). Two NOTs in sequence are replaced by a buffer. See Figure 5.5 for some code examples and their associated mappings.

And, Or and Exclusive Or

As can be seen in Figure 5.4, the XC layout editor does not allow to arbitrarily invert the inputs and outputs of AND- and OR-gates. For instance, an AND with two inverted inputs must be represented as a NOR-gate. Table 5.1 lists the AND-, OR- and XOR-gates with all possible

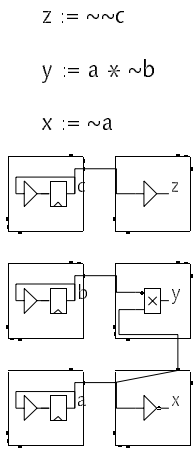

Figure 5.5: Mapping of NOT-Gate

negations at their inputs and output in the left columns, and their transformed representation in the XC layout editor in the right columns.

AND		OR		XOR	
a * b	a * b	a + b	a + b	a - b	a - b
~a * b	~a * b	~a + b	~a + b	~a - b	~(a - b)
a * ~b	~b * a	a + ~b	~b + a	a - ~b	~(a - b)
~a * ~b	~(a + b)	~a + ~b	~(a * b)	~a - ~b	a - b
~(a * b)	~(a * b)	~(a + b)	~(a + b)	~(a - b)	~(a - b)
~(~a * b)	~b + a	~(~a + b)	~b * a	~(~a - b)	a - b
~(a * ~b)	~a + b	~(a + ~b)	~a * b	~(a - ~b)	a - b
~(~a * ~b)	a + b	~(~a + ~b)	a * b	~(~a - ~b)	~(a - b)

Table 5.1: Binary Operators (Left) and Their Mapping (Right)

As an example, Program 5.2 lists the code to map an AND-gate. Note that since we traversed the data structure depth-first, the expression in the left and right subtrees have already been mapped and are in their correct form. This is important as the handling of negations would not work otherwise. Similar algorithms are used to map OR- and XOR-gates. Figure 5.6 presents the algorithm in graphical form.

D-Latch

The latch has to be represented with a multiplexer, where the output is fed back to the zero-input. In Lola, the transformation can be described as

x := LATCH(enable, data) => x := MUX(enable: x, data).

As an XC6200 logic cell cannot have a combinational feedback loop, an additional buffer node has to be inserted between the output of the multiplexer and the feedback input. We also require that the latch is named, that is, that the latch node is rooted in a signal node. This is necessary for ensuring the proper working of other tools, such as the extractor and the checker. It is not a major restriction, but rather enforces a good design style. Figure 5.7 shows a graphical representation of the transformation step and Program 5.3 lists the code necessary for doing the transformation.

In the code, we generate two new nodes and change one existing node into a new form.

Program 5.2 Mapping of AND-Gate

And:

```
prev points to previous node
n points to current node

IF (n.x.fct # TriBase.Not) & (n.y.fct = TriBase.Not) THEN
    Swap(n.x, n.y)                          a * ~b => ~b * a
END;
n.x * n.y OR ~n.x * n.y OR ~n.x * ~n.y
IF (n.x.fct = TriBase.Not) & (n.y.fct = TriBase.Not) THEN
    IF prev.fct # TriBase.Not THEN          proper And
        ~a * ~b => ~(a + b)                 upper part of Figure 5.6
        n.fct := TriBase.Not;               change n into Not
        n.x.fct := TriBase.Or;              change n.x into Or
        n.x.y := n.y.x;                     eliminate Not on n.y
        delete node n.y                     superfluous node
    ELSE                                    prev.fct = TriBase.Not => Nand
        ~(~a * ~b) => a + b                 lower part of Figure 5.6
        prev.fct := TriBase.Or;             change prev into Or
        prev.y := n.y.x;                    eliminate Not on n.y
        n := n.x.x;                         change n
        delete former nodes n, n.x, n.y
    END
END
```

Program 5.3 Mapping of Latch

Latch:

```
IF prev is not a label THEN report error
ELSE
    BUF(prev)
    node := NewNode(TriBase.Buf, prev, NIL);
    MUX1(BUF, data)
    node := NewNode(TriBase.Mux1, node, n.y);
    MUX(enable, MUX1)
    n.fct := TriBase.Mux; n.y := node
END
```

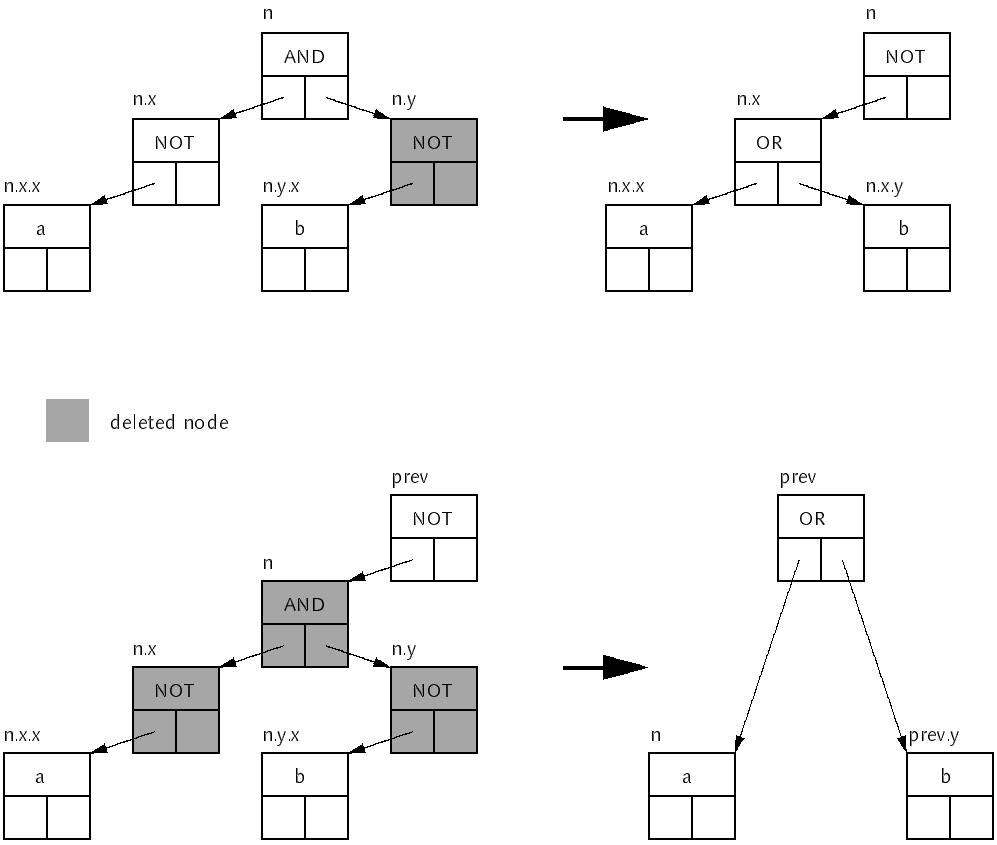

Figure 5.6: Mapping of AND-Gate

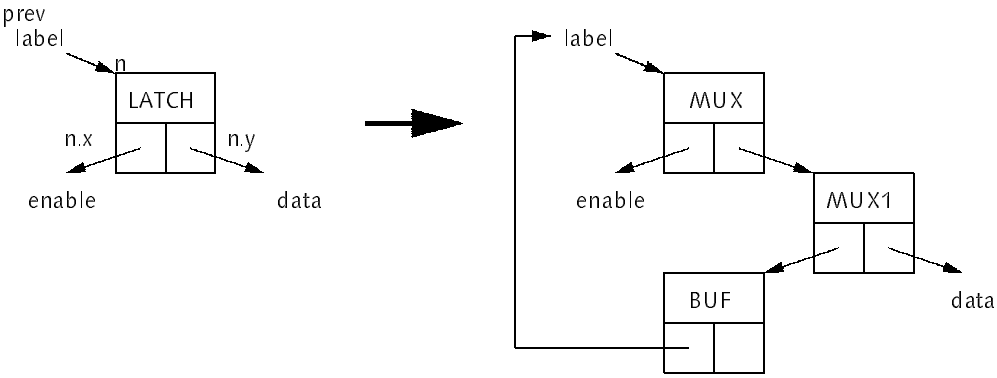

Figure 5.7: Mapping of Latch

SR-Latch

The set-reset-latch with active low control signals has no equivalent in the XC6200 logic cell and is implemented using two cross-coupled NAND-gates. Like the D-latch, the SR-latch has to be rooted in a signal name. We refrain from explaining the transformation in detail and just give the equivalent description in Lola:

```
label := SR(set', reset') => label := ~(~(label * reset') * set').
```

The newly produced expression tree has to be mapped again, as we might have introduced additional negations and AND-gates which should be mapped to OR-gates.

One notable point is that the transformation has to be done in a way that respects the XC editor's limitation given in Table 5.1. The second NOT in the expression above has to be mapped to the left subtree of the AND-gate. Mapping it to the right subtree could not be represented by the XC editor. This is quite a subtle point as one might naturally write down the expression above as

```
label := ~(set' * ~(label * reset')).
```

which would lead to an erroneous expression tree.

Multiplexer

The mapping of a multiplexer node is straight-forward. A negation on the selector input of the multiplexer is mapped by eliminating the negation and swapping the two inputs, i.e.

```
a := MUX(~sel: in0, in1) => a := MUX(sel: in1, in0).
```

A negation occurring at the output of the multiplexer is propagated to the inputs. If the input is already negated, the negation is eliminated, otherwise, a new negation is inserted:

```
a := ~MUX(sel: in0, ~in1) => a := MUX(sel: ~in0, in1).
```

Register

A register with an enable input is implemented using a multiplexer in front of the register, with the selector signal being the enable and one feedback path being the register's output:

```
a := REG(enable, b) => a := REG(MUX(enable: a, b)).
```

Since the enable signal might be negated, the generated multiplexer has to be mapped again to eliminate the negation and swap the inputs.

When a register is reading directly from a signal name, an additional buffer between the register and the signal has to be inserted, as this is mandated for by the XC layout editor. The same has to be done if both inputs of a gate before the register read the register's value. This can occur with enabled registers, reading their negated value. Other cases should be considered as bad designs, but the software must handle these cases nonetheless. Figure 5.8 summarizes the mapping of registers.

Duplication of Input Variables

Input variables declared in type declarations can occur more than once in expression trees of signal assignments. Since the XC editor displays input variables at the point of use, they must be duplicated to allow for multiple displays of the same input variable name in a design. For instance, considering the Adder example shown in Program 5.4, the carry input variable ci occurs more than once in the expressions of type AddElem and must therefore be duplicated. For duplication, the aforementioned id field used for marking mapped nodes comes in handy. We can use it to determine, whether an input variable is referenced more than once. For every further reference, a copy of the variable is created.

A tricky point in the development of this duplication code is the presence of scopes. Since instances of other types can occur within a type and input variables can be passed as parameters to other input variables in inner instances, the duplication of inner input variables has to be

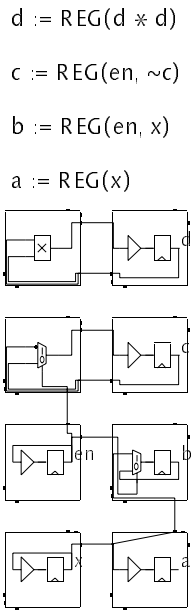

Figure 5.8: Mapping of Register

propagated upwards in the inclusion hierarchy. This is accomplished by a separate pass over all input variables of instances declared in a type; after these, inner instances have been mapped. If an input variable of an inner instance refers to an input variable in the current instance, the latter must be duplicated. Program 5.4 illustrates this point. Note in the example that the carry output of the previous adder element, which is passed to the carry input of the next adder element, is an output variable and does not have to be duplicated.

Anonymous Expressions

Another problem occurring with nested instances is that the actual parameter in a unit assignment can be an arbitrary expression, not only a signal name. These expressions are not anchored in a signal variable and are therefore *anonymous* in the current scope. Their roots are the respective input variables of the instances to which they are passed. Program 5.5 illustrates this problem. Again, as with the duplication of input variables explained above, these anonymous expressions are mapped during the same pass over all instances and their input variables.

Global and Buried Inputs and Outputs

The mapper is also used for transferring positional information found in a *device description file* to the global input and output signals. The global input and output signals are normally implemented as pins of the FPGA. In the XC6200, however, input signals not occurring in the device description are implemented using *buried input registers*, accessible through the coprocessor interface of the XC6200. Likewise, outputs without a description are treated as *buried outputs*, only accessible through the coprocessor interface. Note that these do not have to be implemented as registers, as the coprocessor interface can also read the state of the combinational output of a cell. Program 5.6 shows an example of such buried IOs and Figure 5.9 shows the corresponding layout. The interface generator described in Section 5.8 represents these buried inputs and outputs as interface variables to the software programmer.

Program 5.4 Input Variables and Scopes

```
TYPE AddElem;
    IN x, y, ci: BIT;
    OUT s, co: BIT;
    VAR h: BIT;
BEGIN
    h := x-y;                          first reference to x
    s := h-ci;                         first reference to ci
    co := MUX(h: x, ci)                duplicate x, ci
END AddElem;

TYPE Adder;
    IN x, y: [8] BIT; ci: BIT;
    OUT s: [8] BIT; co: BIT;
    VAR add: [8] AddElem;
BEGIN
    add.0(x.0, y.0, ci);               x.0, ci have to be duplicated
    FOR i := 1 .. 7 DO
        add.i(x.i, y.i, add[i-1].co)   x.i has to be duplicated, co not!
    END;
    FOR i := 0 .. 7 DO s.i := add.i.s END
END Adder;
```

Program 5.5 Anonymous Expressions

```
TYPE AddElem;
    IN x, y, ci: BIT;
    rest as before
END AddElem;

VAR
    x, y, sub: BIT;
    add: AddElem;
BEGIN
    add(x, y-sub, sub);                add or subtract
                                       XOR is anonymous and
                                       only accessible through add.y
    ...
```

Program 5.6 Buried Inputs and Outputs

```
MODULE Buried;
    IMPORT Adders;
    IN x, y: [2] BIT;                  buried inputs
    OUT s: [2] BIT;                    buried outputs
    VAR a: Adder(2);                   ripple-carry adder
BEGIN
    a(x, y, '0);
    FOR i := 0 .. 1 DO s.i := a.s.i END
END Buried.
```

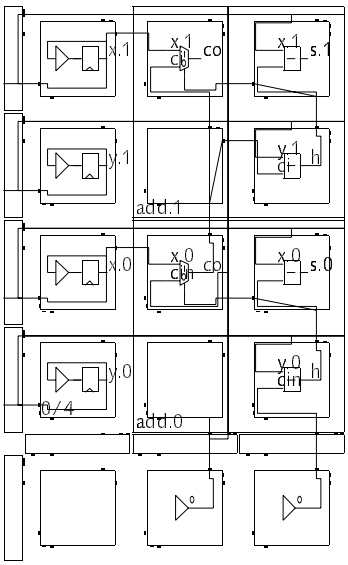

Figure 5.9: Buried Inputs and Outputs

Various Mappings

In addition to the tasks described above, the mapper translates coordinates of Lola position statements given in user coordinates (cell based) into model coordinates used by the Trianus framework.

The mapper also inserts a buffer between an output variable which reads directly an input variable, hence an assignment of the form

```
out := in
```

is translated into

```
out := BUF(in).
```

Additionally, it inserts a buffer between a global output or tri-state signal and its corresponding gate. This is needed for a process called *inversion compensation*, which will be described in Section 5.7. So

```
out := a * b
```

is translated into

```
out := BUF(a * b).
```

5.4.2 Discussion

Although not a difficult problem in principle, the development of the mapping algorithms revealed many subtleties, which were not accounted for in the beginning. In fact, after discovering a problem with the mapping of negations and binary gates in an earlier version, we rewrote the whole mapper from scratch, using the approach described above.

The regular structure of the XC6200 and its simple cell were very advantageous in the development of the mapping algorithm. A direct approach only makes sense for simple cells. In fact, our mapping algorithm does nearly nothing in terms of technology mapping, as most functions of the Lola description are directly implementable by a logic cell. Grouping multiple expressions into one cell is not possible and we can refrain from using a graph-based approach to find a mapping of the compiled netlist onto the target cells.

The complications that occurred during the development of the algorithm, and which were discussed in the preceding paragraphs, were founded more in the requirements of the Trianus

data structure than in the architecture of the XC6200 FPGA.

5.5 Placer and Floor Planner

The placement algorithm used by the Hades placer is *constructive and deterministic*. It produces the same output for the same input, always. The main objectives during its development were quick response and usability for interactive, iterative design. Using a stochastic approach such as simulated annealing [KGV83] was not viable due to the non-deterministic results produced and the long runtimes. Since we rely on the designer to give placement hints — one reason being that the designer will have to give placement hints anyway to achieve a satisfactory layout [WCG96] — we must use a placement algorithm that produces the same result for the same input. The approach taken by the Hades placer is similar to the one described in [MD95].

5.5.1 The Algorithm

The placer takes as input a mapped Trianus data structure, sorts the type hierarchy topologically (innermost first), then places each type and propagates the placement information to its instances using a type broadcast. Each type is placed into an "empty" chip of equal size as the target chip. After the types and instances included therein have been placed, the output buffers inserted by the mapper are placed near the corresponding output pins and all expression trees and instances occurring in the module are placed into the actual chip. The overall structure is given in pseudocode form in Program 5.7.

Program 5.7 Overview of Placement Algorithm

```
PROCEDURE PlaceModule(module);

TriBase.InitID(module);                 unmark all nodes (and wires)
msg.doNode := UpdateInstances;          used to copy placement information
TriBase2.TopoSort(module, list);        topological sort
WHILE list # NIL DO
    type := list.type;
    type.id := Placed;                  mark type as placed
    NewPlacer(p);                       initialize placer data structure
    PlaceDescendants(p, type);          place all signals and instances
    type.w := bounding box width;       calculate bounding box
    type.h := bounding box height;
    msg.type := type;
    msg.placer := p;
    propagate placement information stored in the placer msg.placer
    to all instances of type msg.type
    TriBase.Broadcast(module, msg, TriBase.SelType);
    list := list.next
END;
NewPlacer(p);
PreplaceOutputBufs(p, module);
PlaceDescendants(p, module);
UpdateType(p)                           copy placement information
```

The interesting part of the placement algorithm is the placement of the signals and in-

stances occurring in a type. This is accomplished by procedure PlaceDescendants. We use several heuristics described below to achieve a reasonable placement, which can then be improved using the layout editor and placement hints. The sequence of objects being placed is as follows:

1. instances and signals with their associated expression tree, which have a position hint
2. arrays of instances
3. arrays of variables
4. arrays of output and inout signals
5. instances
6. remaining signals
7. anonymous expressions (reachable only via an input signal of an inner instance)

The reader may note the extensive use of the *array property*. In fact, the presence of arrays of signals and instances and the availability of this information in the back-end tools is the *single most crucial information*. It makes a simple, constructive approach usable, achieving a reasonably good placement on the first attempt.

In the following, we present the placement of arrays, instances and expression trees after giving a few preliminary explanations on free space management.

5.5.2 Free Space Management

Free space on a chip is managed using a *bitmap*. When a single gate from the netlist is placed in a specific cell position, a free bit in the bitmap is sought at that position or in its neighborhood (4 cells in each direction). If no free bit (cell) is found, placement fails and the user must provide a placement hint. When an instance with a certain width and height is placed, a rectangular area of that size is sought in the bitmap using a simple search strategy in the horizontal and vertical directions. Again, if no space is found, placement fails. The positions occupied by an instance or a gate are marked in the bitmap. Overlaps of instances and other instances or instances and single gates are allowed and can be very useful to achieve a dense layout (cf. Section 6).

A more clever way of free space management would involve the use of quad-trees [FB72]. However, since the algorithm described here gives reasonable results, we did not pursue this further.

5.5.3 Placement of Arrays

The algorithm used for placing arrays is as follows: a sequence of instances is placed according to the width and height of the corresponding type. Remember that all instances of the same type share the same placement information. If instances are higher than wide, they are placed adjacent to each other in the horizontal direction; if they are square, or wider than high, they are placed adjacent to each other in the vertical direction. Instances with lower indices are placed first, therefore an array is placed from left to right or from bottom to top. The latter is especially important, as the processor interface of the XC6200 allows to access columns of cells efficiently. For multi-dimensional arrays, the algorithm alternates between vertical and horizontal placement when going from one dimension to the next. Programs 5.8 and 5.9 show this algorithm in detail.

If the array elements are not instances, but signals representing expression trees, the distance between the elements is determined by the width or height of the trees. Arrays of signals

Program 5.8 Placement of Arrays I

```
PROCEDURE PlaceArray(p, obj, u, v, dim, lastDim, VAR w, h);

IF dim < lastDim THEN                       not last dimension
    REPEAT
        thisW := 0; thisH := 0;
        place array of next dimension (recursive call)
        PlaceArray(p, obj, u, v, dim+1, lastDim, thisW, thisH);
        p.dir := other direction(p.dir);
        adjust u, v, w, h according to thisW, thisH, p.dir (direction)
        obj := next in same dimension
    UNTIL obj = NIL
ELSE                                        last dimension
    index := 0;
    get pitch from first item
    IF obj IS TriBase.Instance THEN
        if obj is higher than wide, suggest horizontal placement
        IF obj.h > obj.w THEN
            p.dir := Horizontal;
            pitch := obj.w+1;
            u := p.minU                     start with lowest u coord
        ELSE
            p.dir := Vertical;
            pitch := obj.h+1;
            v := p.minV                     start with lowest v coord
        END
    ELSE                                    simple object
        p.dir := Vertical;
        pitch := p.pitch;                   default pitch used by placer
        v := p.minV                         start with lowest v coord
    END;
```

continued in Program 5.9

Program 5.9 Placement of Arrays II

continued from Program 5.8

```
    REPEAT
        thisW := 0; thisH := 0;
        PlaceObject(p, obj, u, v, thisW, thisH);
        IF (index <= 1) & ~(obj IS TriBase.Instance) THEN
            adjust prediction for simple objects,
            obj[1] sometimes has different size than obj[0]
            IF thisH > thisW THEN
                p.dir := Horizontal; pitch := Max(thisW, p.pitch)
            ELSE
                p.dir := Vertical; pitch := Max(thisH, p.pitch)
            END
        END;
        adjust u, v, w, h according to thisW, thisH, p.dir (direction)
        INC(index);
        obj := next in same dimension
    UNTIL obj = NIL
END
```

represent, for instance, an N-bit wide loadable register or an N-bit comparator circuit. Such simple array elements most often fit into one cell and no type is declared for them. For the special case where the first element (with index 0) of such an array has a different size than the next one, the algorithm adjusts the placement. This guarantees that the whole array is placed in the desired manner. Such a case could occur for the lowest bit of a counter, which consists of a toggle register, whereas other bits of the counter consist of an XOR-gate with a register and an AND-gate.

5.5.4 Placement of Instances

Instances are composed of other (arrays of) instances and of (arrays of) expression trees rooted in signals. Instances have a predetermined size, as their type already has been placed. Instances themselves are placed on top of each other at the next free position that is found in the free space bitmap. If the boundary of the chip is reached, the horizontal coordinate is incremented by the widest width encountered since the last reset of the vertical coordinate and the vertical coordinate is reset. Hence, instances that are not part of an array are placed going from the bottom to the top and from the left to the right of the chip.

5.5.5 Placement of Expression Trees

Expression trees are rooted in an object representing a signal name. The algorithm proceeds from this root into the leaves of the tree (pre-order traversal). Since every node of a Trianus data structure occupies at most one physical cell in the XC6200 FPGA, the first node is placed at the current position. Then, the first and second subtree of the tree are traversed recursively [MD95]. Several nodes are put into one cell whenever possible. For example, an expression of the form

```
a := REG(~x * y)
```

uses a single cell. In doing this, the placement algorithm determines the geometrical position of the nodes and thus performs a task traditionally associated with a technology mapper.

When a node cannot be merged into the current cell it is placed to the right of that cell, if it is in the first subtree, or above that cell, if it is in the second subtree. Thus, the first subtree is placed at the same vertical position as the root cell, and the second subtree is placed at the same horizontal position, with the vertical position increased by the first subtree's height. In case of a multiplexer, the second subtree is placed to the right of the root cell and above the first subtree, and the third subtree is placed above the root cell and above the second subtree. The figures at the end of this section illustrate the different placement strategies.

Program1 5.10 and 5.11 show in more detailed form the actions taken when placing the various node types. In that program, p is a placer data structure, which stores positional information for the placed nodes. This information is used during a type broadcast at a later stage.

Program 5.10 Placement of Nodes I

```
PROCEDURE PlaceArg(p, arg, u, v, offU, offV, to, VAR w, h);

(* If arg is a gate or another expression tree, it is placed at u+offU, v+offV.
   If it is an input label, it is placed at the point of usage u, v.
   w and h return the size of the arg expression tree. *)

PROCEDURE PlaceNode(p, sig, u, v, to, VAR w, h);

IF sig.id = Void THEN                          (* not yet placed *)
    myWx := 0; myWy := 0; myHx := 0; myHy := 0;
    CASE sig.fct OF
    | TriBase.BIT, TriBase.TS:                 (* new root, if already placed it is a leaf *)
        PlaceObject(p, sig(TriBase.Object), u, v, myWx, myHx);
        myWx := Max(myWx, Cell); myHx := Max(myHx, Cell)
    | TriBase.Zero, TriBase.One:               (* leaf cell *)
        ASSERT((sig.x = NIL) & (sig.y = NIL), 110);
        PositionNode(p, sig, u, v, to);        (* place sig and set sig.id *)
        myWx := Cell; myHx := Cell
    | TriBase.Buf, TriBase.Not:
        (* merge negations and buffers into cell u, v if possible *)
    | TriBase.And, TriBase.Or:
        (* mapper guarantees next assertion *)
        ASSERT(sig.y.fct # TriBase.Not, 111);
        PositionNode(p, sig, u, v, to);        (* place sig and set sig.id *)
        IF sig.x.fct = TriBase.Not THEN        (* ~x * y, ~x + y *)
            (* place negation on first subtree into same cell u, v *)
            PlaceArg(p, sig.x, u, v, 0, 0, XCBase.A, myWx, myHx)
        ELSE
            (* place first subtree to the right at u+Cell, v *)
            PlaceArg(p, sig.x, u, v, Cell, 0, XCBase.A, myWx, myHx)
        END;
        (* place second subtree above at u, v+myHx *)
        PlaceArg(p, sig.y, u, v, 0, myHx, XCBase.B, myWy, myHy);
        (* calculate this node's own width and height *)

(* continued in Program 5.11 *)
```

If a cell contains a register, there may be a feedback from the output of the register to the

Program 5.11 Placement of Nodes II

continued from Program 5.10

```
    | TriBase.Reg:
        try to put argument gate of REG into same cell as REG
        take special care in treating feedbacks x := REG(x * y)
        PositionNode(p, sig, u, v, to);          place sig and set sig.id
        reg1 := sig.y; r1u := u; r1v := v;
        mapper guarantees enable = one and reg1.y is gate, not label
        ASSERT(reg1.x = TriGen.one, 112);
        ASSERT(~(reg1.y.fct IN {TriBase.BIT, TriBase.TS}, 113);
        to := TriBase.Void;
        last gate of data is in same cell as register
        PlaceArg(p, reg1.y, u, v, 0, 0, XCBase.Func, myWx, myHx);
        arg := TriBase2.AArg(sig);
        IF arg = sig THEN                        feedback on upper path
            to := XCBase.A
        ELSE
            arg := TriBase2.BArg(sig);
            IF arg = sig THEN                    feedback on lower path
                to := XCBase.B
            END
        END;
        feedback indicated in reg1.to
        PositionNode(p, reg1, r1u, r1v, to);
        clock signal
        PlaceArg(p, sig.x, u, v, 0, myHx, XCBase.Clk, myWy, myHy);
        calculate this node's own width and height
    | similar processing for other node types
    END;
    INC(w, myWx); INC(h, myHx)                   adjust w, h by own size
END;
Check(p, u, v, w, h)                             adjust u, v if top of chip reached
```

gate in front of the register, e.g. x := REG(x * y). This case is encoded in the Trianus data structure with a special value in the to field of the Reg1 node. The placer handles this case in Program 5.11 (TriBase.Reg) using the TriBase2.AArg and BArg procedures. These traverse the expression tree starting at the current node and return the first or second argument of the node, respectively. In the case of the register, this is the first or second argument of the data input to the register. Applied to the example above, AArg yields the register itself (since it is rooted in object x, which is the first argument to the AND-gate in front of the register) and BArg yields object y. Thus, since the register refers to itself, the to field of the Reg1 node is set to XCBase.A. A feedback on the second path would result in to = XCBase.B.

5.5.6 A Note on Routability

Using the tree based approach described in the last section *guarantees a routable design* at the expression and type level. Of course, the approach wastes some cells by leaving them unused. In our experience, the user can manually improve such a placed expression quite easily, while still keeping it routable. The responsiveness of our tools guarantees that this can be done efficiently. The reader is referred to Chapter 6 and Appendix F for case studies on using the tools interactively.

5.5.7 Examples

In the following, we present some short Lola code examples and the corresponding layout produced by the placement algorithm. The routing is shown as well. The first six examples are taken from Chapter 6, Section 1.3 of [Pfi92], which served as examples for the placement algorithm for the CAL FPGA architecture [Alg90, Kea89]. The algorithm presented in that thesis uses several heuristics to ensure a routable placement. We chose an even simpler placement algorithm. As can be seen, the produced result is always routable at the level of the type.

Figure 5.10 shows a simple expression tree. The tree structure of the code can clearly be seen in the layout. The first argument of the XOR-gate is to the right of it, while the second argument is above.

```
TYPE Example1; Tree
  IN a, b, c, d, e: BIT;
  OUT z: BIT;
BEGIN
  z := ~((a * b) - (c + (~d * e)))
END Example1;
```

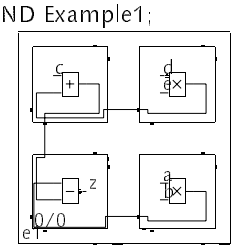

Figure 5.10: Tree Example

In the next example in Figure 5.11, an array of expression trees is placed. The expression tree itself is the one from Figure 5.10. The routing remains the same, as the basic structure is the same. Since the tree is as wide as tall, vertical array placement is used and z.1 comes to lie above z.0.

```
TYPE Example2(N); Forest
 IN a, b, c, d, e: [N] BIT;
  OUT z: [N] BIT;
BEGIN
   FOR i := 0 .. N-1 DO
     z.i := ~((a.i * b.i) - (c.i + (~d.i * e.i)))
   END
END Example2;
```

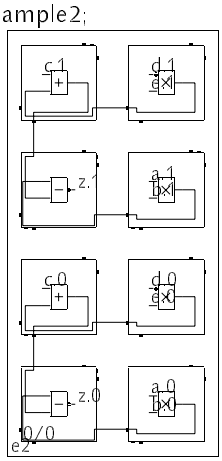

Figure 5.11: Array of Trees Example

The example in Figure 5.12 nicely illustrates our placement strategy, which uses the height of one subtree to determine the vertical position of the other subtree. The circuit grows in height from its leaves and the largest gap between two subtrees occurs near the root (e.g. p.0). The first subtree of p.0 is comprised of all cells to the right of p.0, while the second subtree is comprised of the AND-gate above p.0. Note that only neighbor routing resources are used to preserve position independence of the resulting instance. It would be better to use the length-4 FastLANE connection to route the second argument of p.0 and p.1. The layout is far from optimal, but it is predictable and, most importantly, *repeatable*, i.e. a second invocation of the placer on the same input will yield the same result. An additional *compaction step* such as the one used in [Pfi92] could be used to improve the placement, i.e. to reduce the height of the expression tree by two rows.

The next example shown in Figure 5.13 realizes the multiplexer function shown in Figure 5.12 with Lola's multiplexer construct and encoded selector signals. The resulting circuit is much more compact than the one shown in [Pfi92]. This is, of course, a direct result of the availability of the multiplexer in a single cell in the XC6200 architecture, a feature CAL lacked.

The following example shown in Figure 5.14 implements a left and a right shift register. The first register is placed according to the array placement heuristic, while the second, although similar in structure, is placed according to the expression tree heuristic. In the first example, subsequent array elements refer to previous array elements, which have been placed already, hence the expression tree traversal is always only one level deep. In the second example, however, the placement of the first array element causes a tree traversal of depth N-1, as all subsequent array elements have not yet been placed and are referenced by previous ones. It is questionable, if it would be advantageous to prevent the placement of array elements in expression tree traversals and rely on the array placement to place these nodes. It might produce better placements only in certain cases. We have not investigated this issue further.

Next, we present a parallel to serial converter in Figure 5.15. When the layout is compared

```
TYPE Example3(N); Selector
   IN a, b, c, d: BIT; q, r, s, t: [N] BIT;
   OUT p: [N] BIT;
BEGIN
   FOR i := 0 .. N-1 DO
      p.i := (a * q.i) + (b * r.i) + (c * s.i) + (d * t.i)
   END
END Example3;
```

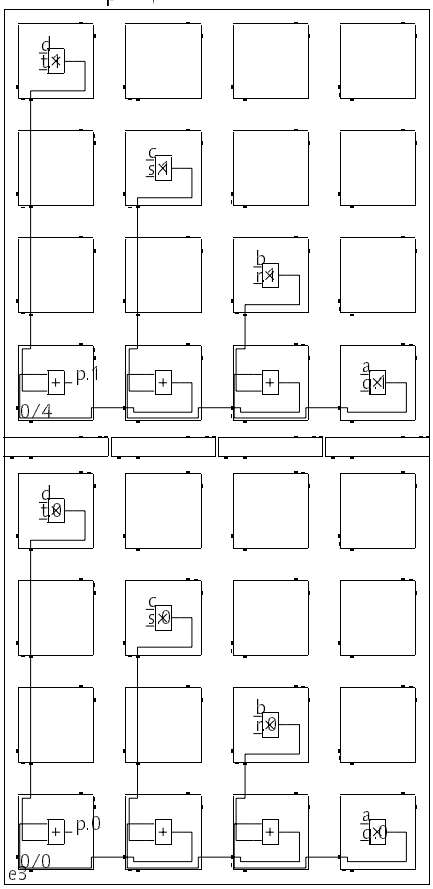

Figure 5.12: Selector Example

```
TYPE Example4(N); Mux
   IN a: [2] BIT; q, r, s, t: [N] BIT;
   OUT p: [N] BIT;
BEGIN
   FOR i := 0 .. N-1 DO
      p.i := MUX(a.1: MUX(a.0: q.i, r.i), MUX(a.0: s.i, t.i))
   END
END Example4;
```

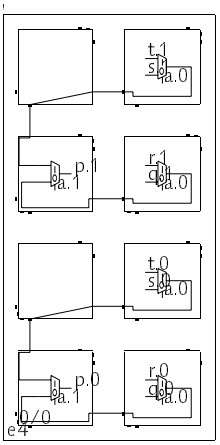

Figure 5.13: Multiplexer Example

to the same example in [Pfi92], the advantage of the XC6200's ability to implement a register and a multiplexer in the same cell is apparent. The last array element is a register with an enable input, which is implemented by the mapper with a multiplexer with feedback, and the rest are simple multiplexers in front of registers. Again, the expression tree placement heuristic overrules the array placement heuristic.

Finally, we show the resulting layout of the automatic placement algorithm of two fundamental circuits, the binary up-counter, shown in Figure 5.16, and the ripple-carry adder, shown in Figure 5.17.

Depending on the routing of the input signals, the default layout of the adder might be the desired one, although a routable layout with a bounding box of three by one cells can be accomplished manually. Most often, the desired bounding box of the counter should be two by one cells instead of one by two. Hence, the counter must be placed using hints. Of course, the layout and routing for such fundamental circuits should be defined once in the form of a library component, which can be imported into user programs.

This concludes the presentation of the layouts produced by the placement algorithm. A set of larger examples can be found in Chapter 6 and Appendix F. There, a coprocessor application and a microprocessor are developed and placed with Hades and the limitations of the algorithm when applied to large designs can be clearly seen.

5.5.8 Manual Placement and Back Annotation of Position Information

When the design placement is not satisfactory, the layout editor is used to improve the placement manually. For instance, the placement of a type such as the one in Figure 5.12 might be optimized manually. Since most users would like this additional placement information to be associated with the description of the logic itself, it must be *back annotated* into the Lola HDL description of the circuit. This is accomplished by a utility procedure, which produces a textual representation of the placement information contained in a displayed layout. When ap-

```
TYPE Example5(N); Shift Right Reg
   IN in: BIT;
   OUT out: BIT;
   VAR s: [N] BIT;
BEGIN
   s.0 := REG(in);
   FOR i := 1 .. N-1 DO s.i := REG(s[i-1]) END;
   out := s[N-1]
END Example5;

TYPE Example6(N); Shift Left Reg
   IN in: BIT;
   OUT out: BIT;
   VAR s: [N] BIT;
BEGIN
   s[N-1] := REG(in);
   FOR i := 0 .. N-2 DO s.i := REG(s[i+1]) END;
   out := s.0
END Example6;
```

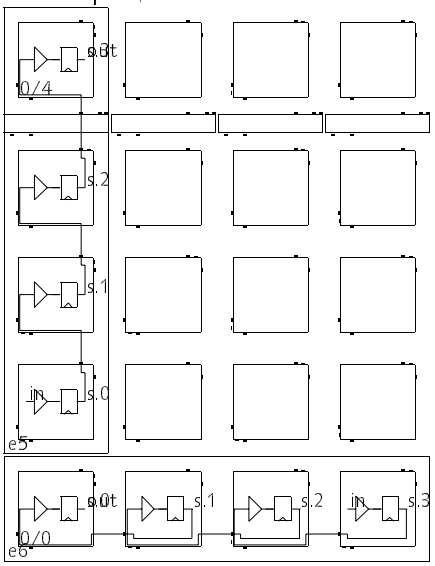

Figure 5.14: Shift Register Example

```
TYPE Example8(N); ParallelToSerial
   IN in: [N] BIT; read: BIT;
   OUT out: BIT;
   VAR s: [N] BIT;
BEGIN
   s[N-1] := REG(read, in[N-1]);
   FOR i := 0 .. N-2 DO
     s.i := REG(MUX(read: s[i+1], in.i))
   END;
   out := s.0
END Example8;
```

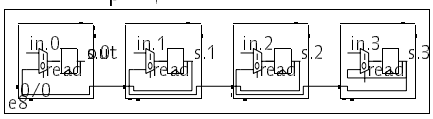

Figure 5.15: Parallel to Serial Converter

```
TYPE CountElem; Counter Element
   IN ci: BIT;
   OUT q, co: BIT;
BEGIN
   q := REG(q - ci); co := q * ci
END CountElem;

TYPE Counter(N); N-bit Up-Counter
   IN ci: BIT;
   OUT q: [N] BIT; co: BIT;
   VAR c: [N] CountElem;
BEGIN
   c.0(ci); q.0 := c.0.q;
   FOR i := 1 .. N-1 DO
     c.i(c[i-1].co); q.i := c.i.q
   END;
   co := c[N-1].co
END Counter;
```

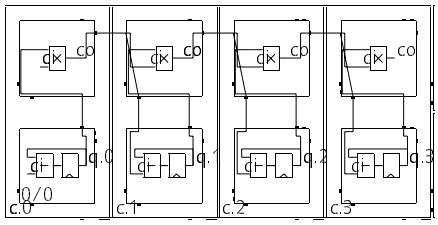

Figure 5.16: Counter Example

```
TYPE AddElem;    Full Adder
  IN x, y, ci: BIT;
  OUT s, co: BIT;
  VAR h: BIT;
BEGIN
  h := x–y; s := h–ci; co := MUX(h: x, ci)
END AddElem;

TYPE Adder;  N–bit Adder
  IN x, y: [N] BIT; ci: BIT;
  OUT s: [N] BIT; co: BIT;
  VAR add: [N] AddElem;
BEGIN
  add.0(x.0, y.0, ci);
  FOR i := 1 .. N–1 DO
    add.i(x.i, y.i, add[i–1].co)
  END;
  FOR i := 0 .. N–1 DO s.i := add.i.s END
END Adder;
```

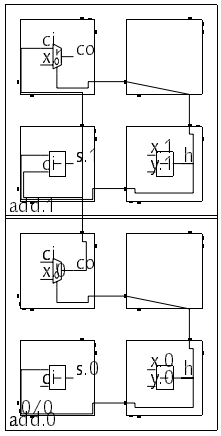

Figure 5.17: Adder Example

plied to the layout shown in Figure 5.16 and 5.17, for instance, the text shown in Program 5.12 is produced. This text can be copied into the Lola description, such that subsequent compilation and placement steps of the program yield the desired placement. Note that the layout information for a type is only displayed once, as all instances have the same placement (e.g. c.0, c.1 and add.0, add.1).

Program 5.12 Textual Layout Information

```
c :: 1, 1;                          Counter
    c.0 :: 0, 0;                    CountElem
        q :: 0, 0;
        co :: 0, 1;
    c.1 :: 1, 0;                    CountElem
    q.0 :: 0, 0;
    q.1 :: 1, 0;
    co :: 1, 1;
a :: 1, 3;                          Adder
    add.0 :: 0, 0;                  AddElem
        s :: 0, 0;
        co :: 0, 1;
        h :: 1, 0;
    add.1 :: 0, 2;                  AddElem
    s.0 :: 0, 0;
    s.1 :: 0, 2;
```

The use of the layout editor and placement hints is exemplified in the pattern matcher application in Section 6.2.

5.5.9 Floor Planner

In addition to the automatic placement algorithm we developed a simple floor planner. It makes use of the placer, but omits the last stage of placement, namely, the placement of the top-level instances and expression trees. Hence, all types are placed by the placement algorithm, but top-level instances are placed by the user. Typically, the floor planner is used if a design is too big to be placed automatically. The user can preplace certain components using a drag-and-drop mechanism, then optionally adjust the placement of certain types, and finally back annotate the Lola program with position statements. Then, a new compilation and placement step will most likely yield a fully placed design. The floor planner was used during the layout of the Wotan microprocessor presented in Appendix F.

5.5.10 Discussion

As will be shown in Chapter 6, the simple approach used by our placement algorithm leads to very fast placement times at the cost of quality. However, for regular designs such as datapaths, the array placement heuristic yields quite satisfactory results. The structure of the Lola code is reflected in the quality of the placement in the sense that the algorithm performs quite well on small types and small expressions. Large types and large expression trees result in wasteful placement leaving many cells unused.

The placement of instances does not take into account the connectivity between these instances. To improve this situation, a min-cut approach as described in [FM82] and [Kri84] could be used, which gives good results at almost linear runtime costs.

It would be advantageous to be able to place only single types and edit them manually. However, the Trianus framework currently does not support the duplication of types or the creation of objects based on a type, except by using the Lola back-end to interpret the Lola code tree. Hence, it is not possible to open the layout editor on a single type and improve its placement. Our floor planner is an intermediate solution to this problem, but not a satisfactory one. To implement libraries, the duplication of types is needed anyway, so the framework will have to support it in the future.

5.6 Router

As mentioned in the introduction to this chapter, the router used in Hades is a *maze-running router* based on the algorithm presented in [Lee61] and used in [Pfi92] (Section 5.3). Despite being a brute-force approach known for a long time, it is the only practical algorithm that meets our requirements. These are

- generality: the algorithm must be capable of routing all kinds of nets (two or more terminals, various shapes)
- customizability: the user must be able to influence the algorithm in various ways (used routing resources, direction of routing)
- adaptability: the algorithm must be adaptable to future routing architectures

A maze-running router finds a path between two cells (terminals) by considering all possible paths between the two and choosing the cheapest one. The cost of a path is a combination of distance and used routing resources. Rare routing resources cost more, e.g. a neighbor wire is cheaper than a length-4 FastLANE wire, which is cheaper than a length-16 FastLANE wire, etc.. Also, paths with bends, i.e. changes of direction, cost more than straight paths. As the router enumerates the paths with incrementing costs, it will find the cheapest path first and does not consider more expensive paths, which also connect the two terminals. Still, as all possible paths must be considered, the runtime is quadratic in the distance between the terminals. Various tricks can be used to limit this quadratic behavior, the effects of which are discussed in Chapter 6.

5.6.1 The Algorithm

The router takes as input a placed design and routes all connections between cells (also called nets). It finds a path from the source of a signal to all destinations of that signal. In the case of the XC6200, it finds a path from cells generating a signal to the cells using that signal at its inputs. It does this by routing connections point to point, i.e. nets with two terminals. If a net has more than two terminals, such as one source and more than one destination, one connection after the other is routed. The router sorts the type hierarchy topologically (innermost first) and routes all nets in each type, propagating the resulting routing information to all instances of that type using a type broadcast. As the module itself is also a type, it will be routed by this process in the same manner. The router data structure is shown in Program 5.13 and the overall structure of the algorithm is given in Programs 5.14 and 5.15. In those programs, r represents a router data structure.

The router makes extensive use of the broadcasting mechanism. The following list explains some of the more subtle points of the algorithm shown in Programs 5.14 and 5.15. The numbers in the list correspond to the numbers given in parentheses in the programs.

1. The user is allowed to insert wires into a design manually, for example, to preroute certain nets. The extractor of the Trianus framework (cf. Section 3.3.6) is used to collect

Program 5.13 Router Data Structure

```
Router = POINTER TO RECORD
    module, type: TriBase.Type;              module and type, which get routed
    coord: TriBase2.Coord;                   list of wire coordinates for
                                             type-based insertion
    nets: TriNetTables.Table;                table of nets to be routed
    minU, minV, maxU, maxV: INTEGER;         bounding rectangle
    hierarchies: SET;                        routing resources, which are
                                             allowed for routing
    map: LeeMap;                             matrix of routing resources, used
                                             for wave expansion
    wires: SET;                              wire directions, which should be
                                             considered for the marking of map
    cost: INTEGER;                           current cost class for queue
    queue: ARRAY CostClasses OF Node;        circular priority queue of next
                                             positions in the Lee-map
END;
```

these wires, attach them to the correct source node and insert them into the correct instance. This is necessary for the correct functioning of the router. See [Geh97] for a more detailed description of the extraction process.

2. Only types that have not yet been routed must be processed. This test is necessary, as it is possible to use the router incrementally, whereby certain types might have been routed during an earlier routing step.

3. As multiple instances of the same type are placed at different positions across the chip, not all instances have the same routing resources available at the same relative position. But since the instances are of the same type, they all must be routed the same way. Consider the four instances of the same type shown in Figure 5.18. In i1 we could use the length-4 FastLANE signal to route the output of cell s to the input of cell d (as done in i1r), however, in i2 this routing resource would not be available at the same position (as shown in i2r). Therefore, the router first has to check what routing resources are available when routing a type. This is determined by invoking procedure LegalHierarchies for each instance of the type to be routed. This procedure determines the position of the first instance and compares the positions of all subsequent instances to that position, excluding routing resources not available to all of them. Obviously, invoking this procedure is not necessary when the module itself is routed.

4. A further problem with multiple instances of the same type is the fact that routing resources crossing these instances, must be marked as used, as they are not available for routing the current type. Consider again the instances shown in Figure 5.18. The neighbor wires running upwards in the third column make the cell's north output of the second and third column unusable during the routing of the type. The north output is neither available in the third cell of i1 nor in the second cell of i2, hence it is not available for routing in the second *and* third cell of the type itself. Program 5.16 shows in detail how the marking is accomplished, namely by using *two nested broadcasts*, a type broadcast for finding all instances of the current type and a bounded broadcast for marking the wires intersecting the current instance.

5. Just as was the case in the mapper algorithm, the router has to process input variables separately, as they are additional destinations of nets, which are not accessible from

Program 5.14 Overview of Routing Algorithm I

```
PROCEDURE RouteSingleType(module, type);

PrepareRoute(module);                    extract manually inserted wires (1)
TriBase2.TopoSort(type, list);
WHILE list # NIL DO
    thisType := list.type;
    IF thisType.id # Routed THEN         not yet routed (2)
        msg.hierarchies := allowedHier;
        IF module # thisType
            collect legal hierarchies for routing (3)
            msg.doNode := LegalHierarchies;
            msg.type := thisType;
            init u, v, minU, minV, maxU, maxV of msg
            TriBase.Broadcast(module, msg, TriBase.SelType);
            move thisType to coords of first instance
            thisType.u := msg.u; thisType.v := msg.v
        END;
        NewRouter(r, module, thisType, msg.hierarchies);
        IF module = thisType THEN
            InitFromInst(r, module);     mark used wires
            RouteGlobals(r)              route clock and clear signals
        ELSE
            mark all used wires, which run over current type's instances (4)
            msg.doNode := MarkWires;
            msg.type := thisType; msg.router := r;
            TriBase.Broadcast(module, msg, TriBase.SelType)
        END;
        collect all nets to be routed and store them into r
        RoutePlacedNodes(r, thisType);
```

continued in Program 5.15

Program 5.15 Overview of Routing Algorithm II

continued from Program 5.14

```
        process all objects declared in thisType
        obj := thisType.dsc;
        WHILE obj # NIL DO
            IF obj.fct = TriBase.Inst THEN
                it is an instance
                next assertion is guaranteed by topological sort
                ASSERT(obj.id = Routed, 110);
                connect inputs in obj with sources in thisType (5)
                RouteInputs(r, obj(TriBase.Instance))
            END;
            obj := obj.next
        END;
        process list of nets to route and send a type broadcast (6)
        RouteAndUpdate(r);
        undo move of thisType
        IF mod # thisType THEN thisType.u := 0; thisType.v := 0 END
    END;
    list := list.next
END
```

within the current type. Therefore, all instances contained in the type must be processed, and all input signals of those instances must be connected to the sources which are contained in the current type.

6. All nets to be routed are collected in a data structure rooted in the router data structure. The nets are first sorted according to a distance criteria discussed in Section 5.6.3 and are then routed using a Lee-map algorithm which is introduced in Section 5.1.3 and described in more detail in Section 5.6.4. The information on wires that need to be inserted into the Trianus data structure is stored in the router. After all nets have been routed, this information is broadcast to all instances of the current type using a type broadcast.

5.6.2 Finding the Nets

To find out which nets of a type need to be routed, all placed nodes of the type are traversed sequentially using the list rooted in type.y (cf. Program 3.7). If a node represents a gate, its x and y subtrees point to the source signals read by that node. If the position of the source node is different from the current node's position, the net is appended to a list of to-be-routed nets, which is rooted in the router data structure (field nets in Program 5.13). For each net, this list contains the source and destination nodes, as well as the input, to which the source has to be connected (e.g. the upper or the lower input of an AND-gate).

5.6.3 Scheduling the Nets

Once it is known, which nets have to be routed, the sequence in which the nets are routed has to be determined. We choose a straightforward heuristic to determine this sequence. *Shorter nets are routed before longer nets*, and nets with the *source and destination in the same row or the*

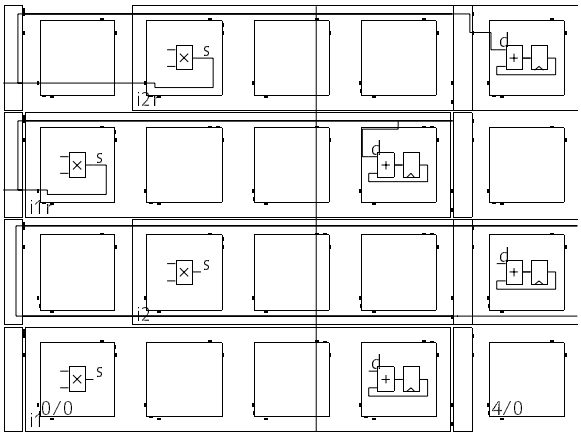

Figure 5.18: Routing Resource Conflicts

Program 5.16 Marking of Wires Running over Instances

Below, some necessary type guards have been left out for brevity.

```
PROCEDURE MarkWire(wire, VAR mark);

r := mark.router;
TriBase2.AbsWire(wire, wu, wv);          absolute position of wire
DEC(wu, mark.minU); DEC(wv, mark.minV);
wu, wv relative position of wire to instance causing this broadcast
IF In(wire.to, r.wires) THEN             only consider legal wires
    mark wire position relative to type's u/v
    SetMap(r, r.type.u+wu, r.type.v+wv, wire.to, Used)
END

PROCEDURE MarkWires(inst, VAR msg);

ASSERT((inst.type = msg.type) OR (inst = msg.type), 100);
IF inst.type = msg.type THEN             it is an instance, not the type itself
    minU, minV contains absolute position of inst
    TriBase2.AbsNode(inst, mark.minU, mark.minV);
    bounding box of inst is stored into mark.r
    mark.r.u := mark.minU; mark.r.v := mark.minV;
    mark.r.w := inst.w; mark.r.h := inst.h;
    mark.doWire := MarkWire; mark.router := msg.router;
    send mark message to all wires, which intersect with mark.r
    TriBase.Broadcast(msg.router.mod, mark, TriBase.SelRect)
END
```

same column are routed before other nets. This *scheduling policy* achieves quite satisfactory results, in that it routes the nets in a similar way an experienced designer would. A good scheduling policy is crucial for the performance of the router. Experience with this simple policy was quite positive, especially since we did not schedule the nets at all in the beginning. We refrained from examining alternative policies, but as they are so crucial to the performance of the router, it might be worthwhile to try out different ones. A similar scheduling policy is also used by [DM95].

5.6.4 Routing a Net

In the following, we explain the steps and data structures involved to route a single net, that is, a connection from a source node to a destination node (also called terminals). As was said before, the router uses a Lee-map to represent the routing resources. This is a two-dimensional matrix of four routing resource entries, where each entry represents a routing resource at a certain position in a certain direction; e.g. the length-4 FastLANE multiplexer in north direction in column 15, row 4. The indices of the matrix are the horizontal and vertical coordinates of the routing resource. A "wave" is spread in this map to find all possible paths between two terminals.

Prior to routing, the map is initialized with the value Free, indicating that all routing resources are available. Then, already used routing resources, i.e. wires already present in the Trianus data structure, are marked in the Lee-map with Used. Finally, all wires that are being sourced by the source node and all neighbor outputs of the source node itself are marked as possible destinations of the wave with the value Dest. The spreading of the wave in the Lee-map can terminate as soon as a position marked as Dest is encountered.

The spreading starts at the destination node, proceeding outwards in all directions until the bounding rectangle (cf. next paragraph) is encountered or an entry marked as Dest is found. The spreading has to start at the destination node, since we might have multiple target positions marked as Dest. For each routing resource, which might be connected to the destination, all possible sources feeding that routing resource are marked as well and the wave spreads from those points further on. To ensure that once the source is found, a way back to the destination can be constructed, the routing resource from which the wave spread is entered into the map at the current position.

To limit the number of map points visited, a *bounding rectangle* is used. If a type is routed, the size of this rectangle is the size of the type's bounding box (cf. Section 5.6.5). If the net is in the top-level (module) the rectangle is made 1/4 larger on each side than the bounding box spanned by the source and destination nodes (cf. Figure 5.19). If routing fails within this rectangle, it is enlarged to the size of the chip and a new attempt is made. This two-phase approach speeds up the routing of the top-level nets by a factor of two on the average.

A *priority queue* of points, from which the wave will spread further, is maintained. Each queue element is weighed according to the current direction of wave spreading and the cost of the routing resource (as was explained earlier). This priority queue ensures a breadth-first spreading strategy of the wave and also helps to limit the use of costly routing resources and bends (turns of the routing direction). For example, if the current routing resource is a neighbor north multiplexer, then further possible points of spreading are the next north multiplexer further south, the east multiplexer west of it and the west multiplexer east of it. In this case, the spreading will continue with the south multiplexer, as the east and west multiplexers represent a change of direction, which are penalized with a higher cost.

The spreading of the wave proceeds until an entry Dest is found. Now, backtracing can start. By examining the entries in the Lee-map, the path back to the destination node can be constructed out of wire segments. The segments themselves are not yet inserted into the Trianus data structure in the form of Wires, but the necessary information is attached to the router data structure (coord field in Program 5.13). This is done so that only one update per

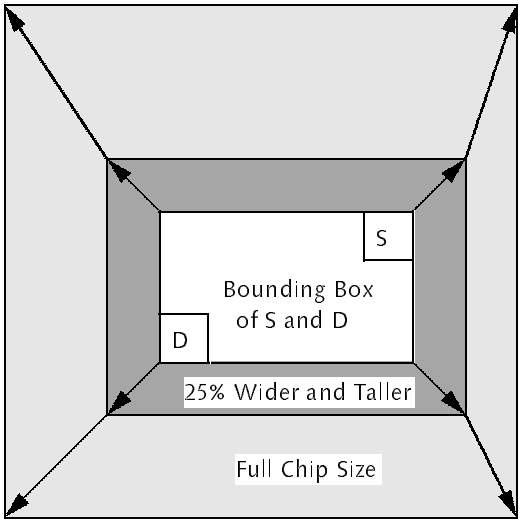

Figure 5.19: Growing of Bounding Box

type has to be made to the Trianus data structure using a type broadcast.

Program 5.17 summarizes the necessary steps to route a single net and the next section explains the process with a small example.

Example

The routing of some expression trees was already shown in Section 5.5.7. For a detailed example, consider Figure 5.20, where d has to be connected with s. In the following, numbers written in italics refer to the numbers in the comments of Program 5.17.

First, the algorithm marks the elements of the Lee-map as Free and then marks the wires to the left of s and below d as Used (not available for routing). Then, the wires leaving s to the north (*1*) and the remaining neighbor outputs of cell s (*2*) are marked as possible destinations (with Dest). The algorithm starts to spread the wave at d, namely at the input multiplexer for the upper input (*3*). Possible sources feeding that multiplexer are the neighbor outputs of the cells to the north, south and east of d, and the neighbor output of the switch to the west of d. The neighbor output of the south cell is already marked as Used and can therefore not be used. Additionally, the length-4 FastLANE outputs of the switches to the north, south, east and west are possible sources, which can be connected to the upper input multiplexer of d. Therefore, seven elements are inserted into the priority queue of further wave spreading points: three elements with equal weight, namely the south output multiplexer to the north of d, the west output multiplexer to the east of d and the east output of the switch to the west of d, and four entries with higher weight — since these resources are rarer than neighbor routing resources, namely the south output for length-4 FastLANE of the switch to the north and the corresponding length-4 FastLANE outputs to the south, east and west of d.

The algorithm then proceeds by removing the first element from the priority queue (*5*), say the entry for the east output of the switch west of d, and by spreading the wave (*7*, *9*) at that position (since it is not a destination, *6*). Further entries will be put into the priority queue for all possible sources of that neighbor multiplexer at the switch, which are the north and south neighbor multiplexer in column 3, the east multiplexer in column 2, length-4 FastLANE routing resources in east and west direction and the length-16 FastLANE routing resource in east direction. All these entries have their according weights, but increased by one compared to the ones before, because these entries are farther away from d. The remaining entries

Program 5.17 Routing of a Net

```
PROCEDURE FindPath(r, src, dst, input, inst, VAR done);
route from src to input of dst, insert wires into inst

handle clock, clear and constant signals separately
mark all wires of src as destinations (1)
w := src.wire;
WHILE w # NIL DO
    TriBase2.AbsWire(w, wu, wv);
    SetMap(r, wu, wv, w.to, Dest);
    w := w.next
END;
mark unused outputs of src as destinations (2)
FOR dir := North TO West DO
    IF r.map[su, sv, dir].from >= Free THEN SetMap(r, su, sv, dir, Dest) END
END;
calculate bounding rectangle into r.minU, r.minV, r.maxU, r.maxV

find path by wave expansion: start from inputs of dst (3)
FOR dir := West4 TO North BY -1 DO QueuePos(r, du, dv, input, dir, 1) END;
LOOP
    INC(r.cost);                                    (4)
    IF r.cost = MaxCost THEN EXIT END;
    GetPos(r, u, v, to, tu, tv, newTo);             consider cells in cost class r.cost (5)
    WHILE to # TriBase.Void DO
        GetMap(r, u, v, to, mark);
        IF mark.from = Dest THEN EXIT               found it, result in u, v, newTo, to (6)
        ELSIF mark is free entry THEN               spread (7)
            mark cell as visited, with direction where we came from (8)
            SetMap(r, u, v, to, newTo, tu, tv);
            Spread(r, u, v, to)                     (9)
        END;
        GetPos(r, u, v, to, tu, tv, newTo)          (10)
    END
END;
done := to # TriBase.Void;
IF done THEN                                        generate path while backtracing (8)
    WHILE not at dst DO
        from := to; to := newTo;
        append information for new wire to r.coord
        PositionWire(r.coord, inst, src, u, v, from, to);
        get next entry
        GetMap(r, u, v, to, mark);
        newTo := mark.from
    END
END
```

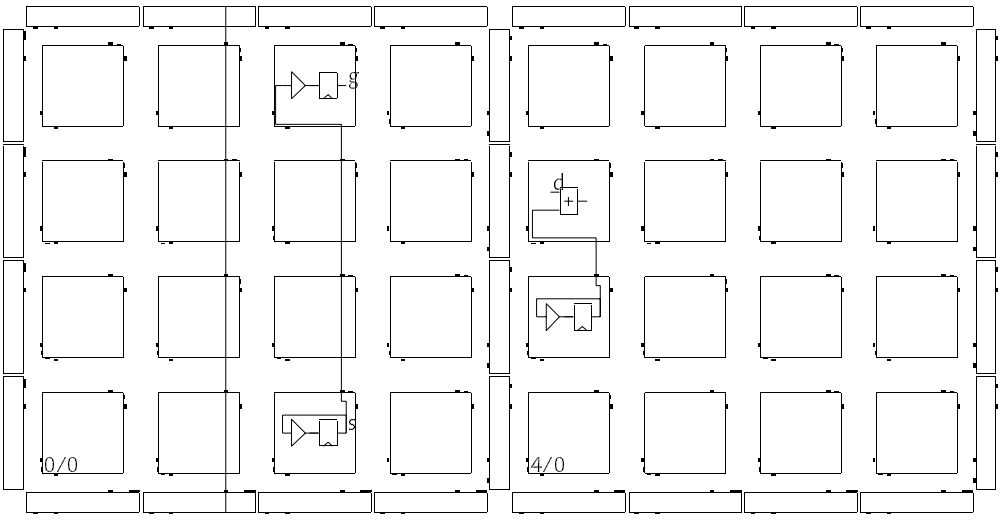

Figure 5.20: Spreading of the Wave

with the current weight are processed (*10*, neighbor multiplexers to the north and east of d) before the weight is increased (*4*). Eventually, the east neighbor multiplexer of cell 2/2 will be processed and the destination coming from s will be found (*6*). Now, backtracing can start and since the algorithm stored the direction from which it was coming from in the Lee-map (*8*), the path back to the destination can be constructed. Figure 5.21 shows the resulting route.

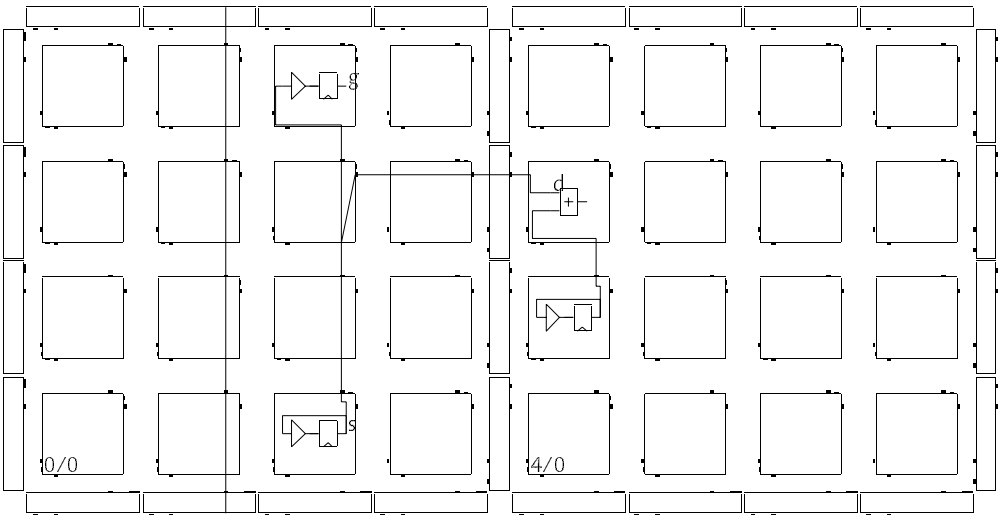

Figure 5.21: Resulting Route

5.6.5 Interactive Routing and Scripting

The user can influence the router in various ways. By setting or clearing flags, the user can determine which routing resources are to be used. For example, when routing small types, which cover more than four cells but should not use the length-4 FastLANE resources, their use can be disallowed. Also, the user can let the router use the bounding box of the placed type,

or expand the type to the maximum possible size prior to routing. This expansion step is also accomplished by using a topological sort of the type hierarchy, but this time, the outermost type is expanded first. (Or rather, all instances of the outermost type.)

A single type can be routed, including all types contained in that type, to give the user the possibility to try out different placements for a type, without having to route the whole design. Also, single nets can be routed. This can be useful to be able to route certain nets manually in order to impose an order on the use of routing paths. For high-performance designs, this feature is essential. Chapter 6 and [Mul97] present some examples.

All routing commands can be recorded in a *script*. This script is a simple text, which the user can edit. A *playback* command can be used to execute prerecorded routing steps. This feature is useful when the router has to be used iteratively to obtain a routed design. If some small change is made to the Lola program and the circuit is synthesized and placed again, the user would have to re-enter all routing commands again manually. By using a script which can be stored together with the Lola program, a design can be routed in the same way as in a previous step. Scripts serve the same function in the router as position assignments in the placer. The typical length of scripts is less than twenty commands.

As the ultimate measure for guiding the router, the user can insert wires manually into the layout to enforce a specific routing. When switching from manual to automatic routing, an extraction and verification step is necessary to ensure the consistency of the Trianus data structure (cf. first note for Program 5.14).

5.6.6 Ripup

If a net cannot be routed, it is necessary to rip up (unroute) certain nets to make room for a different route. This task is completely left to the user. No attempts are made to ripup nets automatically. The user can ripup the whole design, only the top level (all nets in the module), only instances of a selected type, or individual nets. Ripping up a type is an operation accomplished in two phases: in the first phase, a type-broadcast is used to mark the wires that need to be deleted in the type itself and all its instances, and in the second phase, all marked wires in the design are removed.

Ripup should only be used during interactive routing to find a routing schedule for the design. The schedule should then be recorded in a script and be associated with the design.

5.6.7 Discussion

The router is the most crucial piece of software in Hades. It relieves the user from the most cumbersome task required for layout synthesis, namely connecting gates. We extended the router of [Pfi92] to handle the newly available routing resources of the XC6200. Hence, we had a working router after a short time. However, it turned out to be quite a large piece of software, as all the special cases of the XC6200 routing architecture have to be considered. For instance, at a FastLANE switch, the output of the length-4 FastLANE multiplexer is dependent on the output of the neighbor multiplexer (shared routing multiplexer). This introduces all kinds of special cases in the router during the spreading of the wave and also during backtracing. A more regular multiplexer arrangement, which might have used more silicon, would have been advantageous, at least from a software engineering point of view. But as is often the case in hardware design, certain “features” of the chip have to be fixed or by-passed in software. See also Section 5.7 for a further problem of the architecture, whose solution was deferred to software.

A second complication during the development of the router was the support for type-based routing. Although it speeds up the routing of multiple instances by a factor proportional to the number of instances of a type and is a worthwhile capability, the code to implement it is quite complicated and needs profound knowledge of the Trianus data structure.

A resource of the XC6200 which we simply ignored is the “magic” routing resource [Xil96]. It lets one of two inputs of the cell be routed to the next switch directly, by-passing intermediate cells. Its use in type-based routing is questionable, as all instances of a type would be constrained to lie at the same position in a 4x4 block to make use of the magic resources. Apparently, the usefulness of the magic resources is limited, as more often it is the function output of the cell that has to be routed to the next switch quickly [Buc96] and not some input signal of a cell. In engineering, whatever is called “magic” should be met with suspicion.

In our experience, almost all designs are routable after some iterations in the placement phase. Certain nets may need to be prerouted by the user but almost always, this solves a routing or performance problem satisfactorily. The quick response of the router helps in making an iterative design style viable. On a contemporary computer, the user seldom experiences delays longer than one minute (cf. Section 6).

5.7 Bitstream Generator and Loader

Once a design is successfully placed and routed, the configuration bits for the SRAM of the XC6200 can be generated. The configuration data of an FPGA is normally called *bitstream*, and we will use this term from now on. The XC6200 is the first FPGA architecture from Xilinx, whose bitstream format is made public. Hence it is possible for third party vendors and universities to develop their own bitstream generator and associated drivers for downloading the bitstream to the FPGA. We developed a board-independent bitstream generator and a board-dependent driver module for the Hades coprocessor board. The separation into board dependent and independent parts is important as it allows us to port the Hades software onto a different XC6200 coprocessor board by only writing a new driver module.

5.7.1 Bitstream Generator Algorithm

The bitstream generator takes as input a fully placed and routed design. It issues a broadcast to all placed nodes and wires of the data structure. The broadcast invokes a procedure for each node to generate the configuration bits for a cell, and on each wire to generate the configuration bits for the routing multiplexers. The configuration bits are stored into an array, which is a mirror image of the XC6200’s configuration memory. This array of bytes can be stored into a file for later use or downloaded directly onto the Hades hardware. Figure 5.22 depicts the function unit of the XC6200. This is the same figure as in Section 2.3, repeated here for convenience.

Inversion Compensation for Routing Multiplexers

The major problem during the development of the bitstream generator is the presence of inversions on routing multiplexers [Xil96]. We already mentioned this problem in Section 2.3. A process called *inversion compensation* has to be implemented and executed for each cell input, to determine the polarity of the input signal. Once this is done, the multiplexers of the cell can be programmed such that the cell implements the desired function. For example, if the cell should implement $F := a \wedge b$, we showed in Section 2.3 that the upper part of Figure 5.23 implements the desired function, provided that the polarities of the input signals are correct. If, for instance, the a input passes from its source to the destination cell through an odd number of inverting routing multiplexers, the AND-function must be implemented as shown in the lower part of Figure 5.23.

The whole process of inversion compensation is tricky and cumbersome to get right. There is no systematic structure to the presence or absence of inversions on routing multiplexers, so a large table for all possible cases has to be consulted. Again, a low-level detail of the hardware makes the software complicated.

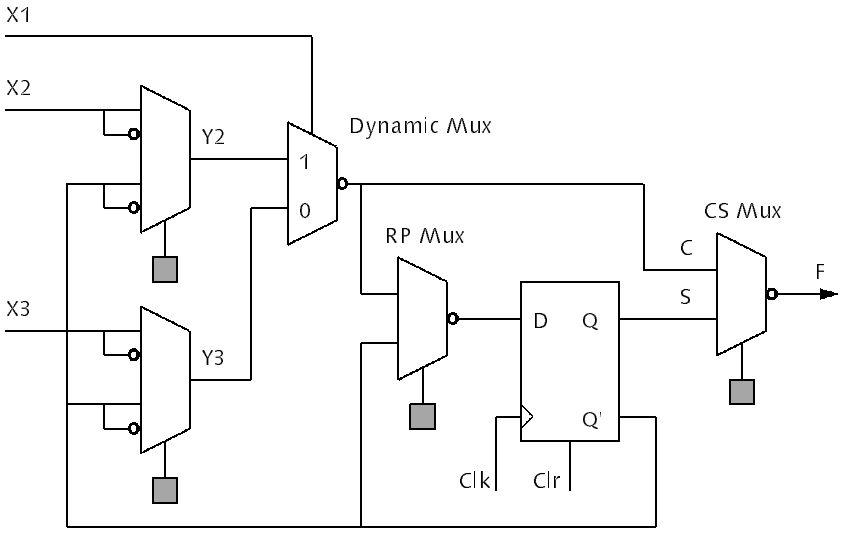

Figure 5.22: XC6200 Function Unit

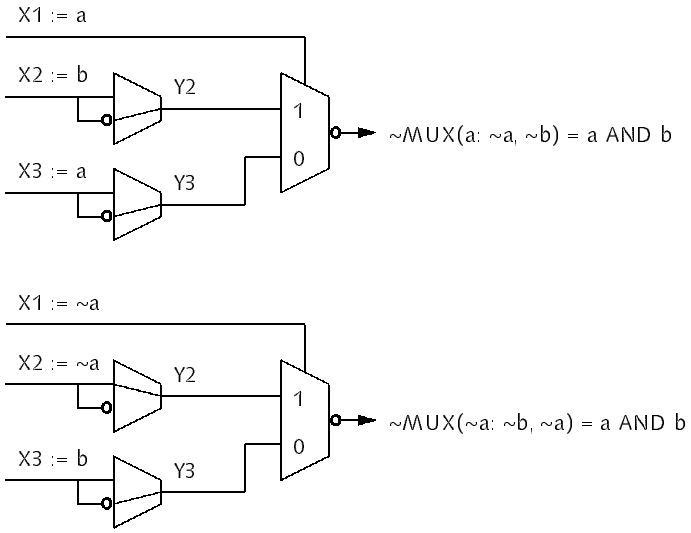

Figure 5.23: Inversions on Inputs

Inversion Before the Register

A further complication arises from the inversion at the output of the RP Mux shown in Figure 5.22. If the cell should implement F := REG(a * b) the output of the central multiplexer is inverted before being read by the register. Should the register see the values of an AND-gate on its input, the central multiplexer must implement a NAND-gate to compensate for the inversion at the RP Mux.

The inversion on the RP Mux is also the reason that the cell cannot implement all possible functions of type F := a op REG(F). This type of cell would implement a Moore-type state machine. However, several two-input gates and the multiplexer function cannot be implemented this way. They store the inverted value of the function into the register, hence they implement F := a op REG(˜F). Xilinx realized this problem only recently and removed these classes of functions from the data sheet.

Inversions and Input/Output Blocks

The biggest problem with inversion compensation happens in input/output blocks (IOBs). While inversions on input signals to a cell can be compensated easily with the Y2 and Y3 multiplexers, there is no optional inversion possible in an IOB. Hence, it is not possible to compensate an inversion caused by the routing multiplexers and, with it, the polarity of the signal on the pad (to the outside world) might not be correct [Xil96]. We solved this problem by requiring the presence of a buffer cell right next to the IOB (cf. Section 5.4). By configuring the buffer cell accordingly, a possible inversion can be compensated. Note, however, that this solution adds additional delay to all output signals. Another solution would be to invert the signal at its source, but this is not a satisfactory solution to the user, as the inverted value of what is expected would be read back during a state access through the processor interface. Also, if the source is connected to another IOB, it might even be impossible to compensate the inversion.

It is interesting (or rather, sad) to report that programmable inversions in the IO buffers were left out for performance reasons [Kea96], but that the vendor's design guidelines now recommend the insertion of these buffer cells as well, causing additional delay.

5.7.2 Loader

Since the configuration bits for a design are stored in an array of bytes, it can be written to a file or downloaded directly to the FPGA. When writing to a file, we use a simple algorithm to compress the bitstream to $1/4$ of its original size on average [BJL92]. The configuration bits are downloaded to the XC6200 FPGA using 32-bit transfers where possible. A clear separation between the low-level interface to the hardware and the hardware independent parts of the loader make it easily retargetable.

5.7.3 Discussion

While not difficult in principle, the development of the bitstream generator was tricky due to the presence of inversions on the routing multiplexers and inside the logic cell. It is these "features" of hardware that contribute to software bloat. As of now, the bitstream generator and loader lacks support for partial reconfiguration of the hardware and for making use of the wildcard registers of the XC6200 (cf. Section 2.3.4). These features can be useful for the implementation of an operating system for the XC6200 like the one described in [Bre96]. The unit of reconfiguration would be a placed and routed instance, where input and output would occur only through padless IO, i.e. using buried inputs and outputs (cf. Section 5.4).

5.8 Runtime System

A reconfigurable coprocessor is of little use if it cannot be accessed in a convenient way from the software side. In Hades, the coprocessor application is described in the form of a Lola program. Interaction between the host and the coprocessor is only possible through input and output signals defined in the Lola program. These signals can be implemented using IOBs, i.e. physical connections between the FPGA and the bus (cf. Section 4.5.1) or as logical inputs and outputs in the form of buried IOs (cf. Section 5.4). The latter is more flexible and the preferred way of interfacing with a coprocessor application in Hades. The advantages of buried IOs are the following:

- No wiring to IOBs is needed.
- The application is relocatable within the FPGA.
- The application is portable to different hardware platforms using the XC6200 FPGA.

5.8.1 Automatic Interface Generator

Hades features an automatic interface generator. It generates an Oberon module from a Trianus data structure. Such an interface module constitutes a *driver module* for the hardware application. The module contains variable definitions of interface objects representing Lola variables in the design. Each interface object has associated with it a map register value and a column position. Arrays of bits are translated into correspondingly sized basic types of the Oberon language. For instance, a bit vector of length-16 is represented by an interface object of type Int. Type-bound procedures are used to read and write the values. Writing a value only makes sense if the object represents a register. A write has no effect on combinational logic gates as long as the register is not used for constant generation in these gates [Xil96]. Program 5.18 shows an excerpt of the interface generator. Types are derived from a generic type Interface and represent the basic types of Oberon (such as CHAR, INTEGER, etc.).

Once an Oberon module is generated for a coprocessor application, the software programmer can use interface objects to safely interact with the application. The interface preserves *type-safety on the software side*, as no low-level features of the Oberon language must be used to interact with the hardware part of the application. Obviously, type-safety on the hardware part is non-existent, as untyped bit values are manipulated. The software programmer can augment the automatically generated interface with additional code, for instance, to initialize an array of bytes with one procedure call.

In Chapter 6, an interface for a pattern matcher application is automatically generated and used to steer the data flow to and from the application.

5.8.2 Future Work

Every computer has an operating system managing its resources, such as disks, graphics and input/output devices. A coprocessor as defined in Section 1.1 is not managed by the operating system, but by the programming language compiler (e.g. a floating point coprocessor) or by the programmer in the form of a library (e.g. a digital signal processor, graphics coprocessor). A reconfigurable coprocessor (RC), however, is a resource changing its “behavior” when it is reprogrammed. Therefore, a runtime system is needed to manage this resource in a way transparent to the user. The first paper analyzing these issues in detail is [Bre96].

In its current form, the runtime system of Hades is sufficient for the manual loading of an RC application and interaction through interface objects. It is the software programmer or the client of the RC application, who initiates the loading of the necessary bitstream onto the RC.

The runtime system should provide more support for these tasks. It has to be known, what application is currently loaded on the coprocessor in order to know if a software request results

Program 5.18 Interface Objects

```
TYPE
    Interface = POINTER TO InterfaceDesc;
    InterfaceDesc = RECORD
        next: Interface;                    interface objects can be linked
        name: ARRAY 32 OF CHAR;             name of variable in Lola
        map: XCBoard.MapRegister;           relevant bits
        col: INTEGER;                       column where this value resides in
        read value from column col using map register map
        PROCEDURE (VAR i: InterfaceDesc) Read;
        write value to column col using map register map
        PROCEDURE (VAR i: InterfaceDesc) Write;
    END;
...
    Int = POINTER TO CharDesc;
    IntDesc = RECORD (InterfaceDesc)
        val: INTEGER;                       value being read and written
        PROCEDURE (VAR i: IntDesc) Read;
        PROCEDURE (VAR i: IntDesc) Write;
    END;
...
PROCEDURE InitDescriptor(VAR i: InterfaceDesc;
    name: ARRAY OF CHAR; map: XCBoard.MapRegister; col: INTEGER);
PROCEDURE SetMap(VAR i: InterfaceDesc);
is map register value of sub a subset of of?
PROCEDURE SubMap(VAR sub, of: InterfaceDesc): BOOLEAN;
```

in the loading of a new hardware module. This and other information about an RC application should be stored in an application descriptor, to answer questions such as

- Where on the RC is the application located and how much area does it occupy?
- Does the application use IOBs to interface with the host?
- Does the application use the on-board SRAM?
- Are interrupts used?

All of this information is present in a Trianus data structure. It has to be distilled into an application descriptor, which can be stored together with the configuration bitstream to a file or into a program module.

5.9 Support Modules and Genericity in Oberon

5.9.1 Data Structure to Store Temporary Data

The Trianus framework implements a data structure that contains enough information to represent an FPGA design. If certain algorithms require additional information associated with the nodes and wires occurring in the data structure, these algorithms must manage this additional information themselves. Both, the placer and the router are tools that require such additional data structures. We developed a module, which defines a data structure and operations on it supporting both the placer and the router. It is a simple linear list of node and coordinate pairs, which is used to store temporary information about wires and nodes that are to be inserted into the TriBase data structure during a type-broadcast (cf. Section 5.5 and Section 5.6). For efficiency reasons, a hash table accessed by the coordinate pairs is superimposed on the list to speed up the search for already inserted wires during routing. This hash table speeds up the routing of nets with high fanout by a factor of 3. Such a simple hash table could also be used to reduce the time needed to locate nodes and wires in the TriBase data structure, a problem already mentioned in Section 3.4.

5.9.2 Table Modules

A table module was implemented to store coordinates of already placed nodes. The table supports dynamic growth, i.e. when new data is inserted, the table grows in size when needed. For the implementation of the net sorting algorithm in the router, a similar container data structure with different contents was needed (cf. Section 5.6.3). Here, the lack of support for generic types in the Oberon language became apparent, as the two modules were identical except for the type they stored. Developments such as [RS97] will make the Oberon language more suitable for the development of general data structure container modules. Currently, the language and the Oberon System lack support for this.

5.9.3 Xilinx Software Interface

Together with Marco Sanvido the author wrote a tool to produce CFG configuration files, which are used by the Xilinx XACT step Series 6000 software. Using this tool, it is possible to generate a design within the Trianus system, and use the Xilinx software to place and route the design. This is useful for comparing the Hades and XACT back-end tools. The converter makes use of two other data structure modules, which are very similar. One is to store associations between strings and strings and one is for associations between integers and strings. Again, both modules could be merged, if generic types would be available in the Oberon language.

5.10 Quantitative Issues

In this section, we evaluate the complexity of the Trianus and Hades systems and compare them to the commercially available tools for the XC6200 FPGA available from Xilinx. We also analyze the memory consumption of the tools when a large coprocessor application is compiled, placed and routed.

5.10.1 Code Complexity

In the following tables, we list the number of source code lines (Lines), the number of statements as reported by the Analyzer tool (Statements) and the number of bytes of object code for the Intel i386 processor (Object). We believe that the number of statements is a better measure for source code complexity than the number of lines of code, as it is independent of the coding style and the presence of comment lines in the source code.

Table 5.2 presents the data for the Hades back-end modules. The router and the loader are by far the largest modules of the back-end. Each constitutes more than half the size of the respective subtotals. This clearly is a sign for the additional complexity introduced with hierarchical routing and the architecture's irregularities and peculiarities.

Module	Lines	Statements	Object
XCMapper	635	442	10437
XCPlacer	1265	886	20358
XCFloorPlanner	112	69	1947
XCRouterBase	1381	1152	25355
XCRouter	1231	865	16463
Hades	182	141	2645
Subtotal	4806	3555	77205
XCDriver	372	211	1943
XCBoard	576	266	5187
XCLoaderBase	1575	925	23387
XCLoader	844	733	16735
HadesInterface	199	49	1266
HadesInterfaceGen	309	266	4678
Subtotal	3875	2450	53196
Total	8681	6005	130401

Table 5.2: Hades Software Size

Table 5.3 lists the sizes of the Lola compiler (front-end), the Trianus front-end (including the Lola compiler back-end, the data structure and the checker and editor frameworks), the editor and checker for the XC6200 FPGA and the timing analyzer. It also lists the total size of the Trianus and Hades system and compares it to Version 0.3.5 of the XACT step Series 6000 software from Xilinx Development Corporation, Scotland. XACT is bigger by a factor of 2 and does not include a runtime system, a driver for a coprocessor board or an architecture-independent framework. However, it features a placer using various algorithms such as min-cut, simulated annealing and constructive placement. Also, the router supports the Magic routing resource and makes use of the shared configuration RAM in a length-4 FastLANE switch [Xil96]. The latter is a constraint in the routing architecture, which the XC editor back-end of Trianus does not support (cf. Section 5.6.7). In this table, we also give the size of the object code for the Ceres workstation.

Subsystem	Modules	Lines	Statements	Object	(Ceres)
Lola Compiler	2	780	946	15289	10184
Trianus Front End	18	7270	5789	127931	81388
XC Editor+Checker	9	5428	4406	96235	70020
Timing Analyzer	6	3630	3433	68217	50032
Support	8	2469	1620	39679	25340
Subtotal	43	19577	16194	347351	236964
Hades	12	8681	6005	130401	99548
Total	55	28258	22199	477752	336512
XACT	-	-	-	1046528	-

Table 5.3: Total Size of Trianus/Hades System

It is interesting to note that the Trianus front-end is quite large when compared to the rest of the system. This indicates that a lot of functionality in a CAD system can be made independent of the target architecture.

To put the Trianus and Hades systems into perspective, Table 5.4 lists the size of the Oberon-2 compiler for the Intel i386 architecture, which is about a third of our tools.

	Modules	Lines	Statements	Object
Compiler	10	11954	8705	165880

Table 5.4: Oberon Compiler Size

On a quantitative issue, Intel object code size divided by number of statements consistently gives a factor of 22. Intel object code is roughly 40% larger than object code for the Ceres (National Semiconductor 32000). The ratio of number of statements to number of lines of code is roughly 75%. But this ratio varies between 65% and 120%, indicating different coding styles. Therefore, the number of statements *is* a better measure for source code complexity.

Comparison to CALLAS

Cuno Pfister and Beat Heeb implemented a system similar to Trianus/Hades for the Algotronix CAL architecture [Hee93, Pfi92]. Table 5.5 compares the back-ends of the CALLAS and Hades systems. As can be clearly seen, the additional complexity of the XC6200 FPGA reflects itself in the complexity of the loader and the router software of Hades. In addition, the more complex, but also more flexible, Trianus data structure adds complexity to the algorithms. The increase in the number of statements and object code is nearly a factor of 4. These are quite high costs for the support of the supposedly moderately more complex XC6200 FPGA.

5.10.2 Memory Consumption

We measured the memory consumption of the Trianus front-end and the Hades back-end during the compilation of the big pattern matcher (16x12) described in Chapter 6. Table 5.6 summarizes the data. All numbers are in Kilobytes and were measured under Windows Oberon, Version 2.0 from the University of Linz. The total does not take into account that some of the memory consumed in previous stages might be recycled by the garbage collector.

	Hades (Stats)	CALLAS (Stats)	Hades (Obj)	CALLAS (Obj)
Mapper	442	-	10437	-
Placer	886	275	20358	5985
Router	2017	726	41818	15795
Loader	1658	407	40122	8838
Total	5003	1408	112735	30618

Table 5.5: Comparison to CALLAS

Phase	KB
Lola Compilation and Expansion	947
Mapping	98
Placing	182
Routing	960
Total (without GC)	2187

Table 5.6: Memory Consumption for Compiling PatternMatch 16 x 12

The memory requirements of the tools are moderate. During expansion of the Trianus data structure, many nodes are allocated resulting in quite substantial memory requirements of the final data structure. The pattern matcher application contains 3048 cells, which roughly corresponds to about 3500 TriBase Nodes. Additionally, the design contains about 3000 labels (TriBase Objects).

Most memory requirements of the Hades back-end can be attributed to the router. It allocates a Lee-map for the wave expansion on three hierarchical levels for a chip of dimension 64x64. 8299 nets have to be routed and temporary storage has to be allocated.

Still, even for an application filling most of an XC6216 FPGA, the 2 MB of memory are very moderate if compared to commercial tools. This statement is supported by the fact that we developed and used the Hades software on a Ceres-2 with only 4 MB of main memory. It is remarkable that a “small” and “slow” machine is sufficient to place and route a design for which the commercially available tools require a PC with a Pentium processor with 16 or better 32 MB of main memory (cf. Table 6.5).

5.11 Experiences with Our Programming Methodology and Oberon

5.11.1 Defensive Programming Pays Off without the Costs

As was stated in Section 5.2, we used defensive programming techniques to develop the Hades software. Upon first sight, checked preconditions and assertions in the program code may incur high runtime overheads if they are executed frequently. Also, index checks are believed by some programmers to slow down program execution considerably. To measure these effects, we compiled the whole Trianus and Hades system without index and assertion checks and compiled, placed and routed the big pattern matcher application from Table 6.5. The difference in runtime was only 5%. Normally, the effect of caching on the performance of software is much larger than index and assertion checks.

In conclusion, it is our belief that programming with assertions pays off. We have no quantitative measure for how much time was saved during testing, but the source of an error

was much more quickly localized by knowing which precondition, assertion or index check failed, than by having to single-step through the code to know where a false calculation was made. Therefore every programmer should enable index checks and insert assertions at crucial points in the code, provided that the programming language has semantics that allow for index checks (C does not!).

5.11.2 Garbage Collection

For Oberon programmers an old hat, but for programmers switching from C and C++ to Java an enlightenment, is the availability of a garbage collector, i.e. of automatic memory reclamation. An extensible system can only be developed successfully, if the programmer can allocate memory at will and release it again simply by removing references to it.

5.11.3 Oberon-2 Language

Hades is written in Oberon-2 [MW91]. Type-bound procedures are only used in the interface generator, to allow the programmer to extend the read and write methods with additional code. Dynamic arrays and the read-only export are used quite extensively, especially in the support modules and the router. Oberon-2 is an elegant language whose only downside is the lack of genericity for defining container data structures.

5.11.4 Oberon System

The Oberon System Version 4 is our host platform. Its availability on most computers and operating systems is beneficial to the spread and acceptance of the tools by other researchers and developers. The Oberon System makes for a very productive programming environment due to its fast compilation times and integrated environment. The system can be extended very easily and new toolets (small tools) can be developed in very short time. An example of such a toolet is the floor planner, which allows the manual placement of instances. It was developed in an afternoon. Another example is a viewer associated with an Oberon background task. The viewer displays information about the location of the mouse in a design, such as the name of the label, the function of the cell or the cell coordinates. It was developed in half an hour.

A drawback of the Oberon System Version 4 is the lack of a reference implementation. When porting our tools from one platform to another, ever so often small adjustments had to be made.

5.12 Discussion

Hardware synthesis is a difficult problem and may never be as fast as software synthesis (i.e. object-code generation). The layout problem alone is more difficult, as it is two-dimensional whereas in software it is one-dimensional (placement of instructions in the instruction stream). Hence, it might always be necessary that designers must resort to low-level descriptions of hardware, including placement information, to achieve high performance and density. Similarly, in software, programmers sometimes have to resort to assembly language to reach higher levels of performance. To support this design style, the current Hades software provides for a very fast design cycle and lets the user exert tight control over the design process.

As we develop good, reusable libraries, fewer and fewer gates have to be placed manually using the layout editor. Data-path elements should be preplaced and prerouted. In the end, only the placement of state-machines and other control logic might need manual assistance. For this, a simulated annealing approach might be viable. For the placement of instances, a min-cut algorithm such as the one in [Kri84] should be implemented and evaluated.

Once library components are used, the placer will be constrained in choosing the placement for already routed instances, as it must take the routing resources used by the instance into account. For example, an adder type using length-4 FastLANE in the vertical direction will be constrained to a specific vertical position with respect to the switches driving the length-4 FastLANE signals. This information must be gathered prior to placement. Currently, it is gathered prior to routing.

Another area where further improvement is possible is the inclusion of timing constraints. The placer might use timing information to determine which gates should be close to each other, although our simple approach already performs quite well, as it places the gates in the order they are connected. The router might use timing constraints to sort the nets according to their criticality.

Runtime systems and suitable languages for the development of reconfigurable coprocessor applications are two areas where much work is needed in the future and where considerable improvements must be made to bring this technology into the hands of software programmers.

6 Application and Evaluation

A large body of literature exists demonstrating the usefulness of configurable hardware to accelerate applications. The yearly IEEE Workshop on FPGAs for Custom Computing Machines is the main conference on this subject.

In this chapter, we use the Hades software to develop a coprocessor application running on the Hades reconfigurable coprocessor board. We present performance data of the Hades software tools and put them into perspective to the commercially available tools for the XC6200 FPGA. In Appendix F, we implement a small microprocessor on the XC6200. Hades was also used by other groups to develop libraries of arithmetic circuits and DSP algorithms. A short overview of their work is given.

All timings in this chapter were conducted on a Dell OptiPlex XL 5120 PC, equipped with an Intel Pentium CPU running at 120 MHz, with 256 KB of second level cache and 32 MB of main memory. We used Windows 95 from Microsoft and Oberon for Windows V4.0-2.0 from the University of Linz.

6.1 Applications Running in Hardware

Coprocessor applications consist of a hardware part and a software part. The software implements the control part of the application, steering the data flow to and from the reconfigurable coprocessor. It also implements the user interface of the application, i.e. it makes the coprocessor accessible to the software programmer. The hardware typically implements that part of the whole application, which accounts for most of its runtime. Good candidates for operations to implement in hardware are inner loops or whole procedures (subroutines) being executed many times. These operations should consist mainly of integer and bit operations and should contain only a small amount of control logic. Hardware in general is profitably used for highly parallel, repetitive applications of primitive operations. If an application processes a stream of regular data, such as a pixel bitmap, and applies simple operations on it, such as convolution, it can most likely be sped up using hardware.

To make the hardware part of an application usable and accessible to the software, a driver must be written for it. Ideally, this driver should be generated automatically such that a software programmer can simply invoke interface procedures to communicate with the hardware. In some cases, however, it is necessary that the driver module be adapted and extended to specific needs. Hades eases this task in that it can automatically generate a driver module, which abstracts from the low-level hardware details of the application. The software programmer can then use this driver and augment it with more powerful interface procedures.

6.2 Pattern Matching Application

Pattern matching is an important application area of computers. Many applications of reconfigurable hardware exist in the field of image recognition and classification [CAC96, Guc95, VSC96]. These applications often run in phases, where the hardware is reconfigured between the phases thus reusing the available silicon.

One type of pattern matching is text searching [Ber93, PTS93, VBR96]. For example, a text editor has a function for searching for a pattern in a text, most operating systems provide a tool for searching a pattern in files (e.g. grep in Unix, Find in Oberon). And with the widespread use and availability of the Internet, searching through large amounts of data such as electronic news becomes a daily task for computer users. A text search application using a reconfigurable coprocessor board to find relevant articles based on user profiles in the daily news feed is presented in [GMN96].

6.2.1 Problem Statement

In the following, we build a simple pattern matcher optimized for finding one or more patterns in a text stream composed of 8-bit characters. To reduce the space requirements and add tolerance to the search, these 8-bit characters are mapped to 5-bit characters using the mapping shown in Table 6.1. In hardware, this results in a reduction in area, whereas in software, nothing can be gained.

6.2.2 Software Solutions

In text searching, it has to be determined whether one or more patterns occur in a stream of data and if so, the patterns are to be located. For single patterns, software solutions use an algorithm such as [BM77, CLR90] to find a pattern. If multiple patterns are to be sought, a finite automaton can be used [ASU86, CLR90]. To implement the aforementioned mapping, the text stream has to be preprocessed using a mapping table.

6.2.3 Hardware Solution

In hardware, a pattern is compared to a stream of data using a comparator circuit which tests for Boolean equivalence of the bits representing the data and the pattern, respectively. Since *parallelism* is easily implemented in hardware, we can detect the occurrence of one or more patterns by implementing *multiple comparator circuits*. A 3x3 pattern matcher is shown in Figure 6.1. It has 3 patterns each of length 3. The data flow is very regular and little control logic is needed. The text is simply streamed by the comparator circuits, which detect a match.

Grey boxes in Figure 6.1 represent loadable registers and the boxes with an equal sign represent comparator circuits. The box with the plus sign represents an OR-gate, which detects a match calculated by any one of the comparators. The Preprocess box implements the character mapping shown in Table 6.1. In the following, we develop the Lola code for this application and use the Hades software iteratively to place and route the pattern matcher.

6.2.4 Overall Structure

Each 8-bit data character is first loaded into a register, then mapped to a 5-bit character using the mapper circuit shown in Program 6.2, and finally loaded into a 5-bit data register. This data register is compared to the corresponding pattern register, which is loaded beforehand.

In our implementation, four data characters at a time are loaded into a 32-bit wide input register. A matching step consisting of 4 cycles is started. It loads a mapped character into the 5-bit data register and shifts the other characters by one position. After 4 such shifts, another 4 characters are loaded and shifted. The result is read back and a match in the previous 8 characters can be detected. Program 6.1 lists a pseudocode description of this process.

A more detailed description of the software part and the interface is given in Section 6.2.10.

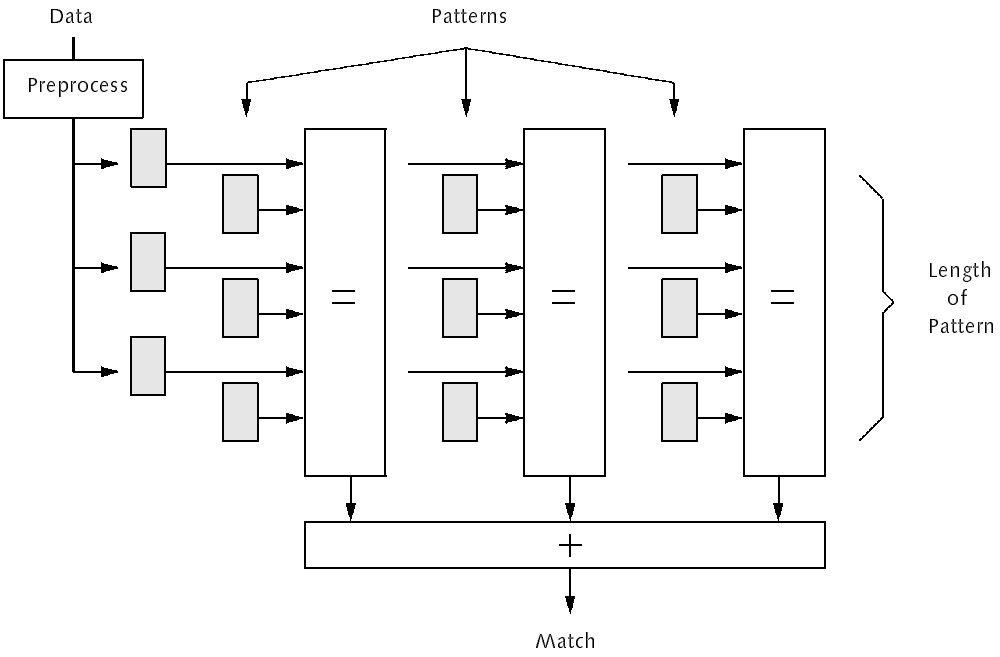

Figure 6.1: Pattern Matcher

Program 6.1 Control Flow as Seen From Software

```
load patterns
WHILE data available DO
    load 4 characters into input register
    perform 4 shift steps, comparisons happen
    load 4 characters into input register
    perform 4 shift steps, comparisons happen
    read back result
    report matches in 8 previous characters
END
```

6.2.5 Preprocessing

To be case insensitive our pattern matcher works with 5-bit characters. Before we put them into the circuit we preprocess the patterns according to the mapping shown in Table 6.1. For performance reasons, the data stream is preprocessed in the circuit itself since the source of the data might be a network adapter in which case the data should not have to pass through the CPU.

Char	8-bits	5-bits
A	65	1
..	..	..
Z	90	26
a	97	1
..	..	..
z	122	26
0	48	31
..	..	..
9	57	31
Rest		0

Table 6.1: Mapping of 8-Bit to 5-Bit Characters

Program 6.2 shows the logic equations implementing the mapping. We tabulated the bit mappings manually and used a logic minimization program [Hof96] based on the Quine-McCluskey method to find a minimal expression tree for each output bit. To reduce the resulting circuit, subexpressions such as t1x0xxxx were defined manually.

Figure 6.2 shows the default placement of an instance of the Mapper type as obtained from Hades. The expression trees are spread out and the resulting instance is quite big: 38 cells in an area of 8x14 with 34% utilization. The layout is routable but can be reduced in size manually. The input to output delay of the mapping function is 17.5 ns.

Next, we preplace all output and variable signals with the help of position assignments to obtain the improved placement shown in Figure 6.3: 38 cells in an area of 5x15 with 51% utilization. The delay is increased to 21 ns. A further reduction in size can only be achieved, if we break up longer expressions into named subexpressions, which we can then preplace with position assignments. Ideally, to have optimal control over the layout, each operator in Lola should have a name, which can then be placed manually. However, this is too cumbersome and not needed in most cases. Since only one instance of the Mapper type exists in our application, the layout shown in Figure 6.2 is sufficiently dense for our purpose.

6.2.6 Comparators

Figure 6.4 shows the schema of a 5-bit comparator circuit. The data bits are compared to the pattern bits using XNOR-gates. These are linked together with an AND-gate tree, which is the fastest way to implement a high fan-in AND-gate. The last AND-gate (e.3.0) will have a value of one, if and only if each data bit (d.i) has the same value as its corresponding pattern bit (p.i).

Program 6.3 shows the Lola type for that comparator (x and y are the bit vectors to be compared).

Figure 6.5 shows the placement of an instance of the Comparator type. It is composed of three arrays and the resulting placement, thanks to the array heuristic, is quite satisfactory: 9

Program 6.2 Mapping of 8-Bit to 5-Bit Characters

```
TYPE Mapper;
    IN in: [8] BIT;
    OUT out: [5] BIT;
    VAR
        t1x0xxxx, t1xx0xxx, t1xZZxxx,
        t1xxx0xx, txxxx0ZZ, tx11xxxx,
        numeric: BIT;
BEGIN
    t1x0xxxx := in.6*˜in.4;
    t1xx0xxx := in.6*˜in.3;
    t1xZZxxx := t1x0xxxx+t1xx0xxx;
    t1xxx0xx := in.6*˜in.2;
    txxxx0ZZ := ˜in.2*(˜in.1 + ˜in.0);
    tx11xxxx := in.5*in.4;
    numeric := ˜in.6*tx11xxxx*(˜in.3 + ˜in.2*˜in.1);

    out.0 := ˜in.7 * ((t1xZZxxx + t1xxx0xx*˜in.1)*in.0 + numeric);
    out.1 := ˜in.7 * ((t1xZZxxx + t1xxx0xx*˜in.0)*in.1 + numeric);
    out.2 := ˜in.7 * (t1xZZxxx*in.2 + numeric);
    out.3 := ˜in.7 * (t1x0xxxx*in.3 + in.6*in.3*txxxx0ZZ + numeric);
    out.4 := ˜in.7 * (tx11xxxx*(˜in.3 + ˜in.2*˜in.1) +
        in.6*in.4*(˜in.3 + txxxx0ZZ))
END Mapper;
```

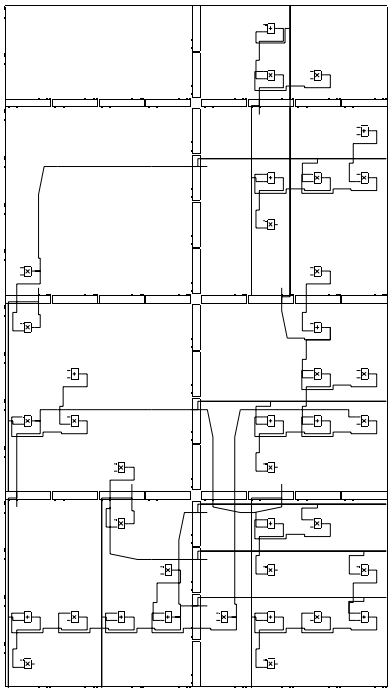

Figure 6.2: Mapper Circuit without Placement Hints

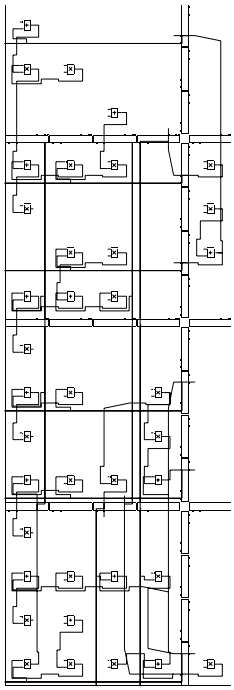

Figure 6.3: Mapper Circuit with Placement Hints

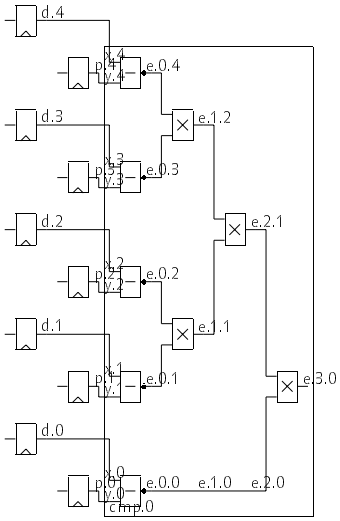

Figure 6.4: Comparator Schema

Program 6.3 Comparing Two 5-Bit Characters

```
TYPE Comparator(Place);
    CONST N := 5; Log := LOG(2*N-1)+1;
    IN x, y: [N] BIT;
    OUT eql: BIT;                          eql = (x = y)
    VAR e: [Log][N] BIT;
BEGIN
    Xnor-gates
    FOR i := 0 .. N-1 DO e.0.i := ~(x.i - y.i) END;
    e.1.0 := e.0.0;                        first level of And-gates
    FOR i := 1 .. 2 DO e.1.i := e.0[2*i-1] * e.0[2*i] END;
    e.2.0 := e.1.0;                        second level
    e.2.1 := e.1.1 * e.1.2;
    e.3.0 := e.2.0 * e.2.1;                third level
    eql := e.3.0
END Comparator;
```

cells in an area of 4x5 cells with 45% utilization. The initial placement is routable and has a delay of 12 ns.

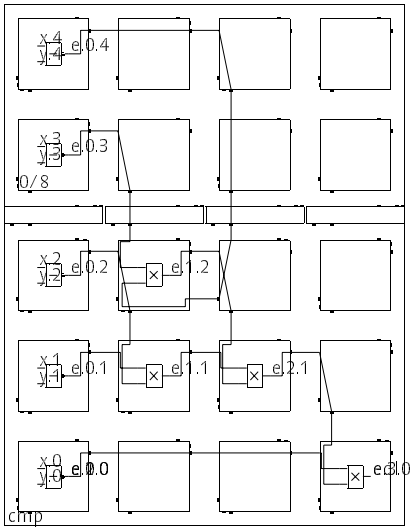

Figure 6.5: Comparator Circuit without Placement Hints

Since the comparator circuit is so central to the size and performance of our pattern matcher, it is advantageous to optimize its layout manually. With the layout editor, a good placement can be found quickly. It is shown in Figure 6.6: 9 cells in an area of 2x5 cells with 90% utilization. Luckily, the size of the tree structure is small enough that all AND-gates fit into one column of the same height as the one containing the XNOR-gates. Note that one cell remains free, which can be used to chain comparators together to form larger comparators (as will be done subsequently).

The initial routing, shown on the left in Figure 6.6, is not optimal due to the routing scheduling algorithm, which routes the connection e.0.0 to e.3.0 before e.0.1 to e.1.1. A small routing script can be used to remedy this situation. The result is shown on the right in Figure 6.6. Note that only neighbor routing resources are used to route the type, since many

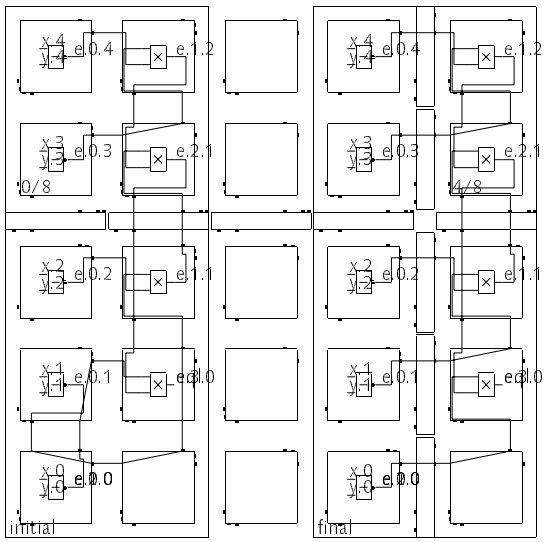

Figure 6.6: Comparator Circuit With Initial and Final Routing

comparators at different locations exist and the routing resources at these locations differ. The delay of the left circuit in Figure 6.6 is 10.4 ns and the delay of the right circuit is 9.5 ns.

6.2.7 Registers

Pattern registers are implemented with buried registers. These make use of the processor interface, a unique feature of the XC6200 FPGA. The patterns are directly loaded into the registers without passing through I/O buffers.

The 8-bit and 5-bit data registers have a load enable signal, which is used to implement the shift steps mentioned in Section 6.2.4. Data is loaded through the processor interface into the 32-bit (4 x 8-bit) input register. Again, no I/O buffers are needed to do that. If the data would be copied directly from a network adapter, I/O buffers might be used to route the data to the input register.

Program 6.4 shows the two Lola type definitions for buried and loadable registers. Note that we could have the technology mapper implement buried registers for us, by simply declaring them as global inputs (cf. Section 5.4). But we would like to define a pattern register in a type, such that an instance of such a type represents a register of a certain bit-width.

The array placement heuristic places the registers optimally, i.e. the bits vertically on top of each other. They are hence easily accessible through the processor interface and can be read or written in one access cycle. Figure 6.7 shows the resulting placement of an 8-bit data input register, a loadable 5-bit mapped data register and a 5-bit pattern register.

6.2.8 Connecting Everything

Now that the building blocks are defined, we connect them together and define the control logic, which steers the data flow in the application. Programs 6.5 and 6.6 show the Lola program of the final pattern match application, which implements 2 pattern matchers each of length-4 (2x4). This application together with a software driver is used for the performance analysis in Section 6.2.11.

The following list explains some points in the code of Programs 6.5 and 6.6. The numbers in the list correspond to the numbers in parentheses given in the programs.

Program 6.4 Loadable and Buried Registers

```
register of N bits, loadable with control signal
TYPE LoadReg(N);
    IN ld: BIT; d: [N] BIT;
    OUT q: [N] BIT;
BEGIN
    FOR i := 0 .. N-1 DO q.i := REG(ld, d.i) END
END LoadReg;

buried register of N bits, loadable with direct register write
TYPE BuriedReg(N);
    OUT q: [N] BIT;
BEGIN
    FOR i := 0 .. N-1 DO q.i := REG(q.i) END
END BuriedReg;
```

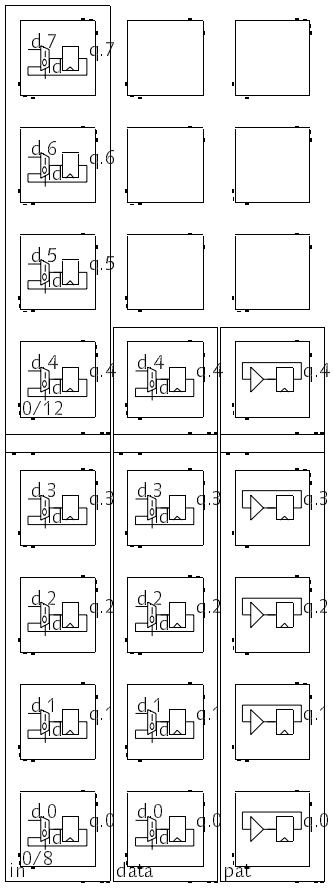

Figure 6.7: Data and Pattern Registers

Program 6.5 Complete Pattern Matcher I

```
MODULE PatternMatch;
    type definitions as shown in Programs 6.2, 6.3 and 6.4

    CONST
        DataSize := 5;                  width of pattern data
        PatternSize := 4;               pattern size
        NofPatterns := 2;               nof parallel comparators
        ResultSize := 8;                length of result vector
    VAR
        input3: BuriedReg(8);           uppermost input register
        input: [3] LoadReg(8);          lower input registers, loadable
        in: [32] BIT;                   input vector (1)
        map: Mapper;                    map incoming data
        the data stream we match against
        data: [PatternSize] LoadReg(DataSize);
        the patterns we look for
        pat: [NofPatterns][PatternSize] BuriedReg(DataSize);
        the comparators
        cmp: [NofPatterns][PatternSize] Comparator;
        eql.i.0 == pattern i matches    (2)
        eql: [NofPatterns][PatternSize] BIT;
        patMatch: [NofPatterns] BIT;
        match: BIT;                     match == any pattern matches
        store result of previous match  (3)
        queue: [ResultSize-1] LoadReg(1);
        result: [ResultSize] BIT;       result vector (4)
        shiftReg: [32] BIT;             control logic state machine
        shift: BIT;                     control bit for shifting
BEGIN
    control logic:                      (5)
    Start shifting by writing '1 into shiftReg.
    Shift for 4 clock cycles by writing '1, '1, '1, '1 into shiftReg, then stop.
    Insert zeroes in shiftReg to allow for longer delays in circuit.
    E.g. shiftReg := '1, '0, '1, '0, '1, '0, '1 => every second clock cycle
    shifting is enabled

    shiftReg.31 := REG('0);
    FOR i := 0 .. 30 DO shiftReg.i := REG(shiftReg[i+1]) END;
    shift := shiftReg.0;

continued in Program 6.6
```

Program 6.6 Complete Pattern Matcher II

continued from Program 6.5

```
    input register, direct memory write from host
    input3(); input.2(shift, input3.q);
    shift characters down
    FOR i := 0 .. 1 DO input.i(shift, input[i+1].q) END;

    map(input.0.q);                          strip input characters to 5 bits

    stream of input data
    data[PatternSize-1](shift, map.out);
    shift characters down
    FOR i := 0 .. PatternSize-2 DO data.i(shift, data[i+1].q) END;

    instantiate pattern registers

    FOR j := 0 .. NofPatterns-1 DO           pattern matchers
        FOR i := 0 .. PatternSize-2 DO
            compare data with pattern, equal chain
            cmp.j.i(data.i.q, pat.j.i.q); eql.j.i := eql.j[i+1] * cmp.j.i.eql
        END;
        cmp.j[PatternSize-1](data[PatternSize-1].q, pat.j[PatternSize-1].q);
        start eql chain
        eql.j[PatternSize-1] := cmp.j[PatternSize-1].eql
    END;

    patMatch.0 := eql.0.0;
    FOR i := 1 .. NofPatterns-1 DO           Or-gate
        patMatch.i := eql.i.0 + patMatch[i-1]
    END;
    match := patMatch[NofPatterns-1];        Does any pattern match?

    result queue
    queue.0(shift, [match]);
    FOR i := 1 .. ResultSize-2 DO queue.i(shift, queue[i-1].q) END
END PatternMatch.
```

1. Variable in is only needed to make the variables input3.q and input.0.q .. input.2.q accessible under one name (cf. Section 6.2.10).
2. eql represents the AND-gates that are used to link individual character comparators together to form a comparator of size PatternSize. Such an AND-chain is started at the highest character and flows down to the lowest character.
3. A result vector in the form of a FIFO queue is used to store the results of comparisons. This is an optimization to avoid polling the match variable by the software driver.
4. Variable result is only needed to make the variables patMatch[NofPatterns-1] and queue.-0.q .. queue[ResultSize-1].q accessible under one name (cf. Section 6.2.10).
5. A shift register suffices to implement the control logic of the pattern matcher. shift, the lowest bit of the register, is connected to the load enable signals of the various registers. A one in the lowest bit of the shift register causes the data stream to advance by one position. Between ones, zeroes can be inserted if additional time is needed between shift steps to propagate signals.

The Hades placer produces the layout shown in Figure 6.8. Note the long chain of registers in the lower right. This is the shift register, which is placed horizontally due to the fact that the lower register reads the output of the upper register. This effect on placement was already shown and explained in Figure 5.14 of Section 5.5.7. We see that the instances are placed close together and hence the layout is not completely routable. As is shown in Table 6.3, this layout results in 38 unrouted nets.

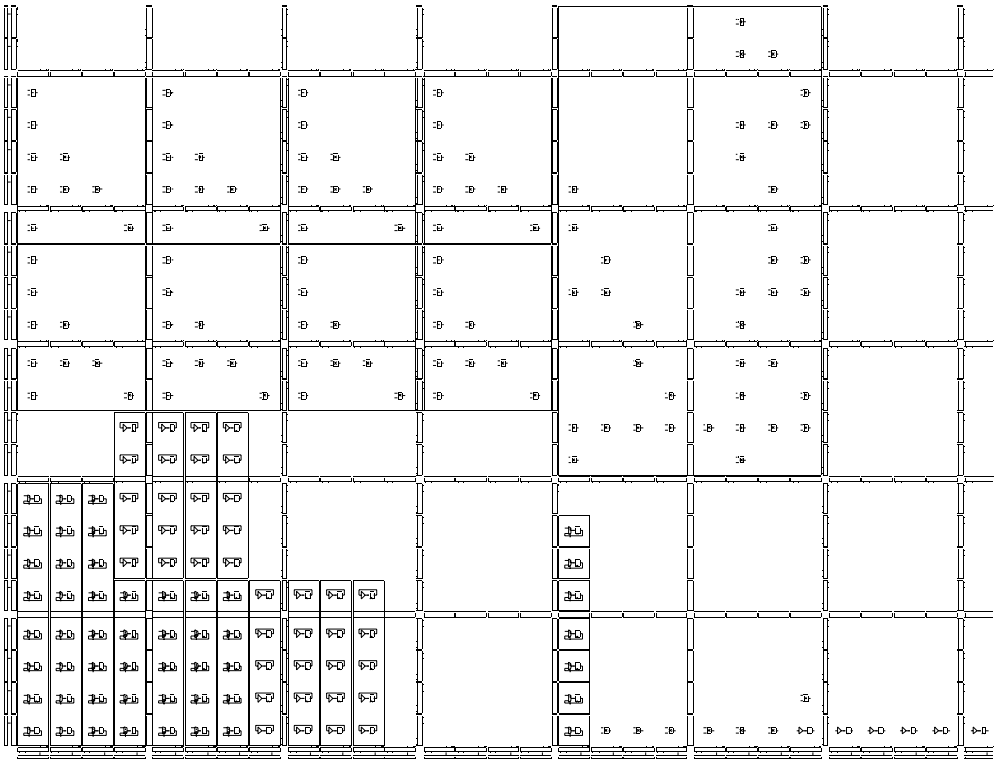

Figure 6.8: Pattern Matcher without Placement Hints

Ideally, the pattern registers should be placed right next to the comparators. Also, since the data registers are read by the comparators, they should be placed such that the same bit of

the register and the comparator lie in the same row. The AND-gates of the eql chain should be placed within the comparators as indicated in Figure 6.6. The layout of Figure 6.8 is therefore optimized manually and the Lola code is augmented with position statements using the back annotation capability described in Section 5.5.8. Based on this code, the Hades tools produce the layout shown in Figure 6.9: To the left, there is a block containing the shift register (leftmost cells), the mapper (big block on top) and the input register (to the right of the shift register). Further to the right are the data registers. Then, the patterns and comparators follow, which are packed together optimally. The AND-gate of the eql chain is located in the lower right corner of each comparator. The result vector appears on the right.

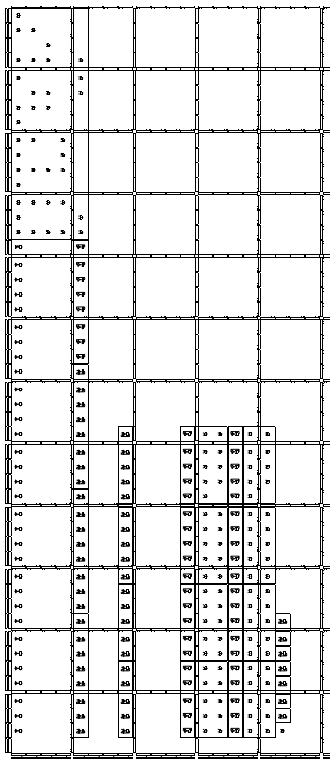

Figure 6.9: Pattern Matcher with Placement Hints

Note that all registers needed by the software interface have the same pitch (distance between bits) and start in the same row. This ensures that the map register only has to be set once for accessing these registers (cf. Section 6.2.10). The worst case delay for the critical path of this circuit is 45 ns; the path runs from the input register through the mapper circuit to the data register. Our Hades coprocessor is clocked with the Ceres clock at 25 MHz, therefore we need two cycles (80 ns) to meet this timing constraint. The shift register of the control logic should therefore be loaded with the pattern 1, 0, 1, 0, 1, 0, 1.

6.2.9 Large Pattern Matcher

The Lola code is written in a way such that by changing the constants NofPatterns and PatternSize, we can produce a large pattern matcher circuit with 16 patterns each of length 12. The fully placed and routed layout is shown in Figure 6.10. Its characteristic data are listed in Table 6.5 in the top left corner.

The bottom row in Figure 6.10 contains the OR-gates that are used to form the match variable, which indicates if any of the patterns matched. Since the last OR-gate represents

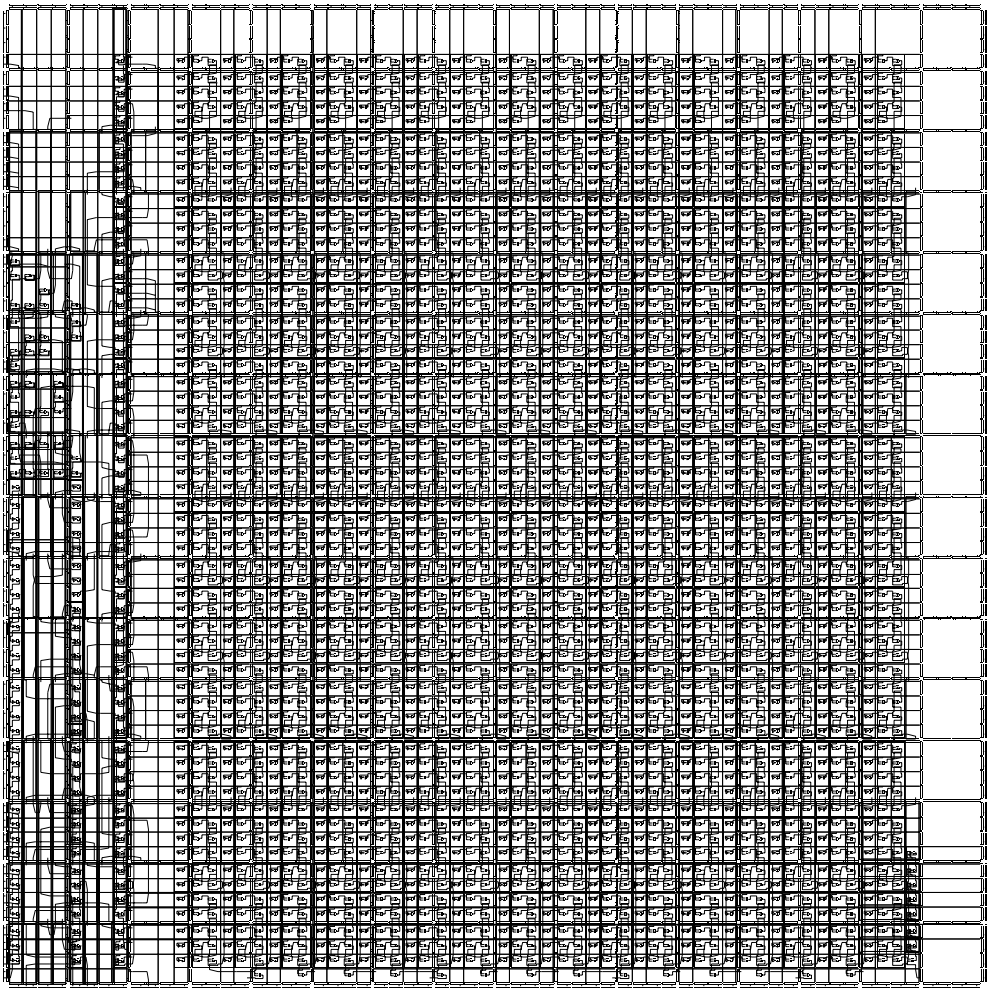

Figure 6.10: Large Pattern Matcher with Placement Hints

the first bit of the result vector, it is placed below the result FIFO queue in the rightmost column. The critical path through this circuit is 137 ns. It runs from a data register through the comparator, the eql chain and the OR-gates to the result queue. Clearly, such a delay is unacceptable, although it still results in an aggregated performance of 1.4 billion character-comparisons per second (10^9/137 * 16 * 12). Pipelining could be used to lower the critical path of this circuit. Even so, the 7 MHz throughput rate of the circuit would suffice when reading data from a disk.

6.2.10 Software Interface

To make a hardware application accessible from software, a driver module is needed. Using the automatic software interface generator available in Hades, a programmer can construct a driver module quickly (cf. Section 5.8). Programs 6.7 and 6.8 list the module produced by the Hades interface generator for the 2x4 pattern matcher shown in Figure 6.9.

The two procedures marked with (1) and (2) were written by the programmer to ease the task of downloading the patterns. Note that instead of implementing the mapping table for the patterns in software, we simply use the available mapping hardware in the coprocessor application to map the pattern characters to their 5-bit values.

If needed, the software programmer can edit the generated code, for example, to change the interface types of shiftReg and result to a set. The impact of this change can be seen in Program 6.9, which lists the search procedure using the interface to communicate with the hardware. The precondition ensures that the registers have the same pitch and the same vertical position. The application then downloads the patterns using the utility procedure from the interface module. It sets the map register and initializes the shift register interface object. The stored value causes the circuit to take two clock cycles per character. Now, text can be read from disk and written into the in register. A write into the shiftReg register causes 4 shift steps. This process is repeated for the next 4 characters. Finally, the result vector can be read and the matching positions can be reported.

6.2.11 Performance

What is the speed limiting factor of a pattern matcher application? For the simple case of searching through a text it is most likely the transfer rate from disk. For modern PCs, this transfer rate is between 3 and 8 MB/s. The 2x4 pattern matcher has a critical path of 45 ns, hence it could support a throughput of 20 MB/s. The large pattern matcher with a critical path of 137 ns could still support a throughput of 7 MB/s. The hardware part of the pattern match application would therefore be fast enough to support the disk transfer rate of today's PCs.

On the PC the Oberon System achieves a disk transfer rate of 4370 KB/s when using block reads and 3204 KB/s when using words reads (4 characters at a time).

To evaluate the performance a user experiences, we modified the Find program of the Oberon System, which is used to find occurrences of a single pattern in a set of files. To avoid file directory operations we merged the source code of the Trianus and Hades systems, which constitute a total of 1.1 MB, into one file. We measured the time to search the word "MODU" in this file. It occurs 659 times. The software solution using the Boyer-Moore algorithm achieves a throughput of 3055 KB/s. It transfers disk data using block reads and comes to within 70% of the disk read speed. A naive string search algorithm has a throughput of 1358 KB/s. It too transfers disk data using block reads. This indicates that the Boyer-Moore algorithm can skip over a large number of characters, eliminating unnecessary tests. In fact, for the pattern "MODU" 98.5% of the comparisons fail on the first compared character.

The software/hardware solution shown in Program 6.9 achieves a throughput of 712 KB/s. Thus, it is 4.3 times *slower* than Boyer-Moore. To see if this is mainly due to reading data word-wise, we used block transfers. With a performance of 756 KB/s the increase was only

Program 6.7 PatternMatch Software Interface I

```
MODULE PatternMatchInt;

    IMPORT HI := HadesInterface;

    VAR
        in*: HI.LIntDesc;
        result*: HI.CharDesc;
        shiftReg*: HI.LIntDesc;
        pat*: ARRAY 2, 4 OF HI.CharDesc;
        data*: ARRAY 4 OF HI.CharDesc;

    written by programmer                    (1)
    PROCEDURE PutPat*(i: INTEGER; pattern: ARRAY OF CHAR);
        VAR j: INTEGER;
    BEGIN                                    set pattern i
        FOR j := 0 TO 3 DO
            write character into input register
            in.val := ORD(pattern[j]); HI.SetMap(in); in.Write;
            shift once -> input register is mapped into data register 3
            HI.SetMap(shiftReg); shiftReg.val := 1; shiftReg.Write;
            read mapped value from data register 3
            HI.SetMap(data[3]); data[3].Read;
            load mapped value into pattern register
            HI.SetMap(pat[i, j]);
            pat[i, j].val := CHR(ORD(data[3].val) MOD 32);
            pat[i, j].Write
        END
    END PutPat;

    written by programmer                    (2)
    PROCEDURE IgnorePat*(i: INTEGER);
        VAR j: INTEGER;
    BEGIN
        load pattern i with unused pattern (not occurring characters)
        FOR j := 0 TO 3 DO
            pat[i, j].val := IgnoreChar; HI.SetMap(pat[i, j]); pat[i, j].Write
        END
    END IgnorePat;
```

continued in Program 6.8

Program 6.8 PatternMatch Software Interface II

```
continued from Program 6.7

    automatically generated
    PROCEDURE Init*;
    BEGIN
        HI.Load("PatternMatch.XC6Bits"); load bitstream
        IF HI.res # HI.Done THEN          error processing
        ELSE
            HI.map[0] := {1..31}/{0..31}; HI.map[1] := {0}/{0..31};
            HI.InitDescriptor(in, "in", HI.map, 4);
            in.val := 0;

            HI.map[0] := {1..8}/{0..31}; HI.map[1] := {}/{0..31};
            HI.InitDescriptor(result, "result", HI.map, 17);
            result.val := 0X;

            HI.map[0] := {1..31}/{0..31}; HI.map[1] := {0}/{0..31};
            HI.InitDescriptor(shiftReg, "shiftReg", HI.map, 0);
            shiftReg.val := 0;

            HI.map[0] := {1..5}/{0..31}; HI.map[1] := {}/{0..31};
            HI.InitDescriptor(pat[0, 0], "pat00", HI.map, 11);
            pat[0, 0].val := 0X;

            similar for remaining patterns

            HI.map[0] := {1..5}/{0..31}; HI.map[1] := {}/{0..31};
            HI.InitDescriptor(data[0], "data0", HI.map, 7);
            data[0].val := 0X;

            similar for remaining data

            unneeded interface objects manually removed
        END
    END Init;

BEGIN
    Init
END PatternMatchInt.
```

Program 6.9 PatternMatch Application

```
PM = PatternMatchInt

PROCEDURE Search(r: Files.Rider; pat: ARRAY OF ARRAY OF CHAR);
    VAR pos: LONGINT; i, j: INTEGER;
BEGIN
    ensure that setting of map register via PM.in
    works for other interfaces as well
    ASSERT(HI.SubMap(PM.shiftReg, PM.in)
        & HI.SubMap(PM.result, PM.in), 100);
    i := 0;                                    load the patterns
    WHILE i < LEN(pat, 0) DO PM.PutPat(i, pat[i]); INC(i) END;
    WHILE i < LEN(HI.pat, 0) DO PM.IgnorePat(i); INC(i) END;

    HI.SetMap(PM.in);
    PM.shiftReg.val := {0, 2, 4, 6};           only every second clock cycle
    WHILE ~r.eof DO
        read 4 characters, store them into input register, shift 4 times
        Files.ReadBytes(r, PM.in.val, 4); PM.in.Write; PM.shiftReg.Write;
        read 4 characters, store them into input register, shift 4 times
        Files.ReadBytes(r, PM.in.val, 4); PM.in.Write; PM.shiftReg.Write;
        read back result
        PM.result.Read;
        IF PM.result.val*{0..7} # {} THEN found something in prev 8 characters
            report position based on set bits
            in PM.result.val and file position
        END
    END
END Search;
```

marginal. Additionally, we eliminated the overhead caused by calling the methods of the interface objects by inlining the coprocessor communication code into the search loop. The resulting throughput was still only 950 KB/s. Although this value is 33% better than the solution using interface objects, it is still 3.2 times slower than Boyer-Moore. If we search for a pattern occurring frequently (4 white space characters), which does not allow long skips, the throughput of Boyer-Moore drops to 1828 KB/s while the hardware solution still has the same throughput.

When we search for *multiple patterns* in the text using the hardware solution, we achieve the *same throughput* as we can make use of the parallelism available in hardware.

Table 6.2 summarizes the throughput values for pattern "MODU" and also lists the values obtained on the Ceres-2. There, disk read speed is very slow and both, Boyer-Moore and the hardware solution with inlined communication, are bound by the disk read speed.

	PC	Ceres
Disk: Block Read	4370	123
Disk: Word Read	3204	42
Naive	1358	39
Boyer-Moore	3055	108
PatternMatch	712	40
Block Read	756	45
Block Read + Inline	950	108

Table 6.2: Searching "MODU" (Throughput in KB/s)

Communication Bottleneck

Why is the hardware solution so slow? The main problem is the relatively high cost of communication. With the current software interface, an indirect procedure call is used (method) to transfer a value to and from the coprocessor. This is the cost that has to be paid to support extensibility and versatility of the interface during development. During each access, a method call and two procedure calls are invoked. Additionally, one of the procedure calls checks its arguments for validity (precondition).

With the Hades coprocessor board for the Ceres-2, the final procedure call accesses the board at the same cost as normal memory (cf. Section 4.5.1). With a prototype PCI-board for the PC [LSC96], the final procedure call first stores the destination address into a latch on the PCI-board and then accesses data on the board. Hence, accessing the PCI-board is at least twice as expensive as accessing normal memory. Additionally, since IN and OUT commands of the Intel i386 CPU are used, which have latencies of 20 cycles, accessing the board costs at least 40 cycles. Compared to the 6 cycles on the Ceres-2, this is a relative difference of more than a factor of 6.

If we inline the communication code in the main search loop, we can achieve the same speed on the Ceres-2 as with Boyer-Moore. Hence, on the Ceres-2, the method call overhead is the reason for the slow performance of the initial solution. On the PC, although inlining increases throughput by 33%, the hardware solution is still very slow compared to the Boyer-Moore algorithm. We anxiously await the new PCI-board [VCC97] together with appropriate driver software, which lets us access the board using direct memory accesses.

6.2.12 Improvements

Up to now, we have only made use of the processor interface to store the pattern registers into the circuit without having to connect the registers to I/O buffers. We have not made use of the fast reconfiguration speed of the XC6200 FPGA. In fact, any memory location in the XC6200 can be altered just as quickly as the user registers. Therefore, we studied the possibility of improving the performance and reducing the size requirements of the comparator circuits by applying this feature.

Making Use of Reconfigurability

In Figure 6.4, the basic structure of a 5-bit comparator was shown. The data and the pattern registers are compared using an XNOR-gate, which yields one, if both inputs are the same. Note that the value in the pattern register does not change. Therefore, we can propagate this constant value through the XNOR-gate and replace it with a buffer or an inverter. Figure 6.11 shows the two resulting circuits (shown on the right), when a constant zero or a constant one is present at one of the inputs of an XNOR-gate (shown on the left).

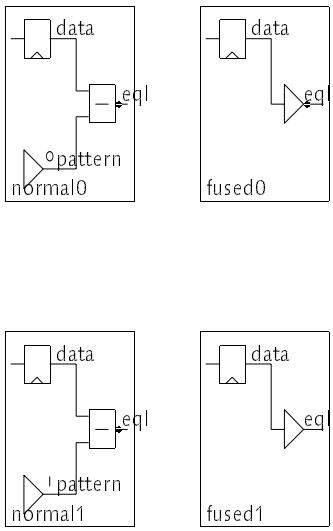

Figure 6.11: Constant Propagation

We can implement these negations with no extra cost in the XC6200 cell and can therefore implement a two-bit comparator circuit using a single AND-gate with appropriate negations on its inputs. Once the pattern is known, the buffer or inverter function in front of the AND-gate can be programmed directly into the cell. Figure 6.12 shows the comparator from Figure 6.6 together with a pattern register on the left and the same comparator after constant propagation on the right. The number of cells and the space required to implement a 5-bit comparator is reduced by a factor of three.

The original comparator had a delay of 9.5 ns, while the compact comparator's delay is 5.7 ns. This is an improvement of just 3.8 ns, but the size advantage of the compact comparator is drastic. While the bounding box of the 16x12 pattern matcher shown in Figure 6.10 is 60x61 cells, the bounding box of the compact version is just 29x61 cells.

Pipelining

To reduce the delay for large pattern matchers, one can introduce pipeline registers in the comparator's eql chain. This approach, however, would require additional data registers to

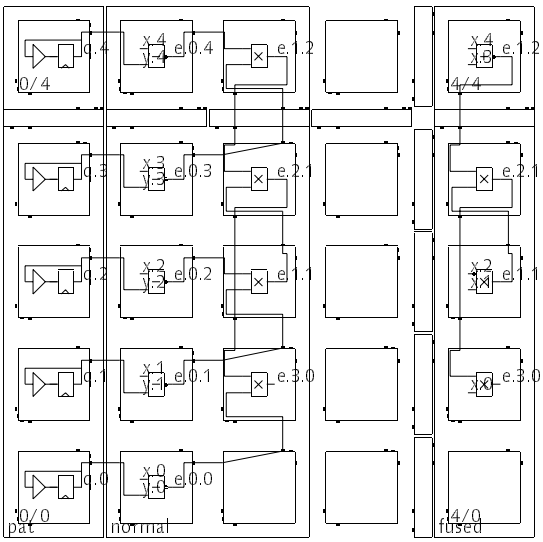

Figure 6.12: Conventional and Fused Comparators

compensate for the additional delay in the eql chain. By using an AND-gate *tree* instead of a chain, we can omit these delay registers. This way, the critical path for the 16x12 pattern matcher can be reduced from 137 ns to 45 ns, which again is caused by the mapper circuit.

6.2.13 Discussion

The pattern match application presented in this section was a proof-of-concept application. Performance of the circuit itself is quite good even though the performance is not seen by the user. One reason is the inefficiency of the automatically generated interface. In the future, we intend to support the generation of *inlinable interface procedures.* In Oberon, this is possible through the use of *code procedures.* One drawback is that the interface generator becomes target machine specific. The gain in performance justifies this, however. Another reason for the inefficiency is the lack of a memory-type interface on the PCI-card. The availability of the necessary driver software, however, is only a matter of time.

6.3 Comparison to XACT step Series 6000

In this section, we compare the (semi-)automatic generation of the pattern matchers using the Hades software from Chapter 5 against the XACT step 6000 software, Version 0.3.5 from Xilinx Development Corporation, Scotland (simply called XACT in the following). We gratefully acknowledge the permission to perform this evaluation. We do not compare Trianus to commercial front-end tools, i.e. HDL compilers or schematic entry systems, as none were available to us. We want to point out, however, that VHDL compilers have reported runtimes in the order of minutes to hours to compile small modules of the size of the PatternMatch application [Woo96a, WLH97]. These tools are three to four *orders of magnitude* slower than the Lola compiler with the Trianus back-end.

The files used by XACT were produced by Hades using the CFG-file converter tool. It generates design files in the internal file-format used by XACT to store designs on disk.

6.3.1 Small Pattern Matcher without Hints

The first experiment is the automatic placement and routing of the pattern matcher developed for the Hades reconfigurable coprocessor. It has two parallel pattern matchers, each consisting of four 5-bit characters. It was chosen as a typical example of a coprocessor application with a regular datapath (data and pattern registers, comparators), some random logic (mapper) and little control logic (shift-step register). Most often, such a small design is used to determine the placement hints for the data path part, which are then put into the Lola code. It is therefore mandatory that the design cycle is fast, as many iterations are needed to find a satisfactory placement, which is also routable.

Table 6.3 summarizes the results obtained. We used various options in XACT for automatic placement and routing. A typical user may just use the default options supplied by the tools, which are: high effort, no type based routing, use of Magic routing resources.

248 Cells and 779 Nets	Hades[7]	XACT low[8]	XACT medium[8]	XACT high[8]	XACT trans[9]
Compile	0.1 s	-	-	-	-
Reading Design Files[1]	-	1 s	1 s	1 s	1 s
Map	< 0.1 s	-	-	-	-
Place	0.1 s	5.8 s	17.6 s	29.4 s	47.5 s
Bounding Box	56 x 22	18 x 18	13 x 63	13 x 54	13 x 55
Route					
Without Magic[2]	-	18.5 s	173.9 s	56.6 s	54.4 s
Unroutes[3]	-	133	31	43	33
Prerouted Types[4]	**7.5 s**	17.8 s	112.8 s	55.9 s	93.1 s
Unroutes[3]	**38**	136	57	43	24
With Magic[2]	-	24.9 s	162.4 s	101.1 s	**53.5 s**
Unroutes[3]	-	93	12	13	**5**
Prerouted Types[4]	-	**27.2 s**	**97.1 s**	**98. 9 s**	57.3 s
Unroutes[3]	-	**86**	**10**	**13**	5
Total (M+P+R)[5]	7.6 s	33 s	114.7 s	128.3 s	101 s
Speedup of Hades[6]		4.3	15.1	17.1	13.3

Table 6.3: Pattern Matcher without Hints: 2 Patterns of 4 Characters Each

The following list contains explanations for the corresponding labels in Table 6.3.

1. This row lists the time needed by XACT to read the design files, which were produced by Hades. The files are in the same format as the ones used by XACT itself. Hence, the delay encountered would be the same if a different tool was used to produce the files, such as a commercial schematics editor or an HDL compiler.

2. The Hades router does not support the “Magic” routing resources (cf. Section 2.3.2 and Section 5.6). The times for Hades are hence listed in the “Without Magic” rows. As XACT has an option to allow or disallow the use of the Magic resources, we list the times for both cases.

3. This row lists the number of unrouted nets, i.e. connections for which the router failed to find a path.

4. Hades preroutes all types by default. All instances of the same type have the same routing. In XACT, the user can preroute the types, i.e. enforce the same routing on all instances of that type. In this row, we list the routing results when types are prerouted.

5. This row lists the sum of map, place and route times. In each column, the best route time is taken, indicated in boldface. Best means the lowest time with the lowest number of unrouted nets.

6. This row lists the speedup obtained when Hades is used instead of XACT.

7. The Hades column lists the time for Lola compilation and the generation of the Trianus data structure from it. Only one route time is listed, as Hades always uses type-based routing and does not support the Magic routing resource.

8. XACT has an "effort" option used during placement. "Low" stands for little effort and fast run time, "high" stands for high effort and long run time. As can be seen from the size of the bounding box, the placer packs the cells as closely as possible. It makes no attempt at placing the registers next to the point of usage. Consequently, the number of unrouted nets is very high for all routing options we tried. The routing rectangle used is that of the bounding box in this case. If a larger rectangle is given, the routing time increases dramatically.

9. "High" uses the highest placement effort and "trans" indicates, that instances of the same type (comparators in our case) may be transformed, i.e. they do not all have to have the same placement as their type.

Evaluation

Hades stands out with exceptionally fast compile, map and place times (less than a second). The router is very quick as well. The resulting placement was already shown in Figure 6.8. Not surprisingly, it is not routable (38 unrouted nets).

XACT is slower by a factor of 5 to 12 with comparable results. Higher placement efforts generally result in better routable designs. But even using the highest placement effort and allowing for the transformation of instances still results in an unroutable design. By using two ripup and reroute steps on that design, however, the design can be routed successfully. The mapper circuit of Figure 6.2 is placed within 5x12 cells as opposed to 8x14 cells by Hades. In this case, the stochastic placement algorithm results in a compact placement.

It is noteworthy that for XACT no clear recommendation can be given on whether to use type-based routing or not. Also, for multiple runs using the same option, the stochastic nature of the placement algorithm becomes apparent as the resulting routing times varies by as much as 400%. The numbers listed in Table 6.3 indicate typical run times, i.e. run times a user most likely experiences. It might be possible that a route completes 4 times faster (as was the case during testing), but there is no clear recipe to achieve that.

For some placements, prerouting the types results in faster routing ("medium"), and for others it slows down routing ("high trans"). The use of the Magic routing resources, however, is recommended, as it always reduces the number of unrouted nets. It can have a negative effect on the routing time, though.

None of the placements produced by XACT would have been usable for an RC application because registers in an array were arranged in a circle and could not have been accessed efficiently using the processor interface.

6.3.2 Small Pattern Matcher with Hints

In the second experiment, the input to the placement and routing algorithm was the Lola code annotated with placement hints. Essentially, placement is done by hand in the form of Lola position statements, thus only the router has to perform "real" work. Table 6.4 summarizes the results. As the placement was already performed for all cells except for the expression trees of the mapper, we used the default placement option for XACT, which was "high". A run

with “low” did not improve the runtime of the placement phase. To see how much workstation technology has improved in 8 years, we conducted the experiment also on Ceres-2. It features a National Semiconductor 32532 CPU clocked at 25 MHz and was developed in 1987. The PC was purchased in 1995, i.e. 8 years of technological improvement lie between the two machines.

248 Cells and 779 Nets	Hades	Hades on Ceres	XACT
Compile	0.1 s	1.5 s	-
Reading Design Files	-	-	1.2 s
Map	$<$ 0.1 s	0.3 s	-
Place	0.1 s	0.7 s	2.3 s
Bounding Box	18 x 47	18 x 47	18 x 45
Route			
Without Magic	-	-	41.6 s
Unroutes	-	-	12
Prerouted Types	**1.9 s**	26.6 s	42.1 s
Unroutes	**0**	0	12
With Magic	-	-	41.8 s
Unroutes	-	-	0
Prerouted Types	-	-	**39.8 s**
Unroutes	-	-	**0**
Total (M+P+R)	2 s	29.1 s	42.1 s
Speedup of Hades		14.6	21.1

Table 6.4: Pattern Matcher with Hints: 2 Patterns of 4 Characters Each

Evaluation

As can be seen, Hades performs very well compared to XACT and it routes the design without retries or manually prerouting certain nets, and without using Magic routing resources. XACT succeeds on the first try only when using the Magic routing resources. Note that using type-based routing does not improve the result much. Hades has a compile, place and route design cycle that is a factor of 20 (!) faster than place and route using the Xilinx tools. In fact, Hades completes the design in the same time as XACT places a preplaced netlist (i.e. no work by the placer has to be done except for placing the mapper circuit and checking the validity of the placement hints). Even on the “slow” Ceres-2, the design is completed faster than on the PC using XACT. This is a somewhat sad result as it indicates that 8 years of technological advancement is annihilated by software. This is a clear case for the applicability of *Reiser’s law*, which says that software is getting slower faster, than hardware is getting faster.

A factor of 20 makes a qualitative difference in the usage of the tools. If we include VHDL compilation times, this factor increases to about 100 to 1000, depending on the VHDL compiler used. Note that the turnaround time with XACT is below one minute, which should therefore be considered as fast, but when the user has to perform several iterations to find a routable placement, a turnaround time of 2 seconds is much better than one of 40 seconds. Such a fast design cycle increases productivity and reduces the time needed to find a satisfactory placement.

6.3.3 Large Pattern Matcher with Hints

In our final experiment, a big instance of the pattern matcher is generated. Table 6.5 lists the result for a pattern matcher with 16 patterns, each of which has a length of 12 characters. This results in a design with 3048 cells, filling most of the available space on a XC6216 FPGA (83% utilization within the bounding box).

3048 Cells and 8299 Nets	Hades	Hades on Ceres	XACT 1	XACT 2
Compile	2.4 s	21.0 s	-	-
Reading Design Files	-	-	108.9 s	108.9 s
Map	0.5 s	3.5 s	-	-
Place	0.7 s	5.2 s	22.7 s	21.1 s
Bounding Box	60 x 61	60 x 61	60 x 61	60 x 61
Route				
Without Magic	-	-	198.8 s	192.3 s
Unroutes	-	-	2	2
Prerouted Types	**20.8 s**	158.6 s	396 s	382.1 s
Unroutes	**0**	0	2	2
With Magic	-	-	**145.9 s**	318.3 s
Unroutes	-	-	**0**	0
Prerouted Types	-	-	292.1 s	**274.6 s**
Unroutes	-	-	0	**0**
Total (M+P+R)	22.0 s	188.3 s	168.6 s	295.7 s
Speedup of Hades		8.6	7.7	13.4

Table 6.5: Pattern Matcher with Hints: 16 Patterns of 12 Characters Each

Evaluation

Again, Hades is very fast, a factor of 8 faster than XACT. We made several runs using XACT and two typical ones are listed. They only differ in the placement of the mapper circuit, which is non-deterministic. The effect on routing time is quite drastic, though. The second run takes twice as long as the first. Interestingly, prerouting the comparator type results in much slower routing speed in the first case and faster routing speed in the second case. As was seen earlier, routing only succeeds if the Magic routing resource is allowed, so it truly deserves its name, at least in combination with XACT. The speed of Hades on the Ceres-2 is still on par with XACT on the PC.

6.3.4 Discussion

When faced with choosing CAD tools for an FPGA, users typically have only one choice, namely the tools provided by the vendor of the FPGA. To satisfy all possible demands users might have, vendors provide tools with a plethora of features and options. However, they often neglect the quality of the underlying core algorithms, which perform the real work in CAD tools.

The above observation also holds true for the new XACT step 6000 tools for the XC6200 FPGA. Three different placement options with more suboptions are provided, with which a user can influence the quality of the resulting placement. As our experiment in Table 6.3 showed, even using the highest effort and allowing for transformations of the types does not result in a routable design on the first try. And after several ripup and reroute cycles, the

resulting design could still not be used as a coprocessor application. Hence, the user has to give placement hints, which have to be back-annotated to the HDL source code or the schema.

Since XACT uses among others a simulated annealing algorithm for placement, the quality of the produced result can vary drastically. The router sometimes runs slower, sometimes much faster and the user might just be lucky and get a routable design, or he or she must iterate ten times and try out different options to achieve a good result. This is a very time-consuming method to develop a design. We believe that direct user control is better, since the produced result is the one expected.

The XACT router can be influenced in several ways as well. Normally, of course, users will enable all routing resources to ensure the successful routing of a design, but it is not at all clear if types should be prerouted or not. For certain designs, it results in faster routing with higher quality (less unrouted nets), but for other designs, it can have the opposite effect. It seems that the user is only left with a trial-and-error approach, which is very time-consuming if the design cycle time is in the range of minutes.

The main advantage of our approach is *speed*. Using Hades, a user can *explore the design space* much more quickly. It becomes possible to give placement hints in the design description and see the effect of it half a minute later. This enables a completely different style of constructing hardware, as it *allows the user's knowledge about the design to enter the design cycle* much more easily. The result is, we believe, that the *design is finished in less total time* than using smarter but slower tools.

6.4 Hades in the World

The Trianus and Hades systems are publicly available (cf. Appendix G). Two groups outside the Institute for Computer Systems used Hades in the past to design circuits for the XC6200 FPGA. Their work is presented subsequently. Virtual Computer Corporation will distribute the Trianus and Hades system with their XC6216 based PCI-board as an alternative to the Xilinx tools [VCC97].

6.4.1 Using Iris and Hades for DSP Algorithms

The DSP laboratory of Queen's University, Belfast coupled Hades to their Iris synthesis framework [Tra95, TWM95, TW96]. Iris works on the building block philosophy where the designer can define digital signal processing blocks and perform synthesis on these circuits. At first glance, this methodology may not appear attractive but DSP designers typically like to mix and match circuits, using different number representations and clever circuit design techniques to achieve efficient FPGA solutions. Iris achieves this by enabling the extraction of parameterized expressions from complex VLSI processing elements, and using these expressions to achieve functionally correct solutions for circuits built from these processors. Designers can quickly create and evaluate architectures that utilize existing hardware blocks.

Previously, Iris generated structural, parameterized VHDL code, which was then synthesized using the Synopsys VHDL compiler [Syn92]. Compilation times were on the order of hours. Iris has then been closely integrated with Hades resulting in a powerful system capable of quickly investigating an FPGA implementation. This integration was achieved by developing a Lola interface, which allows Iris to produce Lola code for the algorithm to be realized. The systems are well matched as there is considerable structure within Iris which Hades can preserve and quickly translate into a layout. This allows a different design strategy to be employed in that it removes the designer from the low level design flow. For example, if the designer finds the required target performance has not been met at the circuit layout level, he or she can go back to Iris and apply some of the many circuit transformations available. The option of adding pipelining delays is particularly well-suited for FPGA designs, as it can

sometimes be implemented at no extra cost. The key issue is that circuit optimization is being performed at the algorithmic and architectural level, which is less consuming than varying placement and routing at the FPGA level. The resulting system is presented in more detail in [WLH97].

6.4.2 Developing Arithmetic Circuits for the XC6200

During a term project at the Institute for Integrated Systems at ETH Zürich, P. Müller implemented various adder structures for the XC6200 FPGA. He described the adders in Lola and then used Hades to place and route the designs. Since adders are fundamental circuits which must be optimally placed, hints were used to achieve good placements. However, it was possible to write the Lola code in a parameterized fashion, such that adders of arbitrary sizes can be defined. The router was used interactively and often single nets were prerouted to guide the routing of subsequent nets. A 32-bit carry-increment adder [ZK97] with a delay of 39 ns and a pipelined ripple-carry adder with a delay of 10 ns were built. More details can be found in [Mul97].

6.5 Possible Future Applications

In this section, we present some possible future applications of the Hades RC board or variants of it. Many applications found in the literature could and should be implemented on the Hades RC board to evaluate its architecture and also the architecture of the XC6200 FPGA. They are not discussed in this section.

6.5.1 Switcherland Reconfigurable Coprocessor Node

One interesting application of an XC6200-based coprocessor board would be its incorporation into a network such as Switcherland [OE95]. The throughput of Switcherland is about 20 MB/s, which is well matched to the processing speed of an RC. A pipelined version of the pattern match application from Section 6.2 could be used to filter a data stream and detect certain patterns in packets passing by, for instance to gather statistical data. A packet can be processed by simply routing it via the coprocessor board instead of directly to its destination. Therefore, the presence of an RC in the network is completely transparent to an application.

6.5.2 Guard Evaluator for Active Oberon

In Active Oberon, tasks are synchronized using guards, which are evaluated by the scheduler [DR97, Gut97]. If a guard is asserted, a task can resume execution. This guard evaluation step can be quite costly when the guards depend on global variables of the system. One way to speed up this evaluation process is by using a second processor in a multi-processor system. Another way would be to use an RC board which has access to main memory. The guards could then be evaluated by the coprocessor and the scheduler running on the CPU could simply test a bit vector to determine which guard is asserted.

6.5.3 System Monitoring

An RC board, which has access to the system bus, can monitor the computer system and gather statistical information about bus traffic. This can be useful to analyze system performance and to find out, where an application spends most of its time. At DEC SRC, for evaluation purposes a PCI Pamette [Sha96] is used to generate and monitor traffic on the PCI bus.

6.5.4 Support for Arbitrary Precision Integers

To speed up calculations with arbitrary precision integers, an RC board can be used to accelerate addition and especially multiplication operations. The PAM group used FPGAs to speed up long integer multiplication and implemented an RSA algorithm which held the speed record for a long time [VBR96].

6.6 Discussion

Our own experience and experiences from other groups with Lola, Trianus and Hades give reason to believe that we have constructed a usable and reliable system to define and implement RC applications. More work is needed to find out what kinds of applications are suitable for RCs, i.e. a taxonomy is needed for quickly determining the applicability of RCs on a given problem. We need more experience with building actual applications, especially to see what support is needed in the runtime system to make this technology usable for software programmers.

Lola is quite good for describing datapaths, i.e. regular logic, but descriptions of random logic and state machines tend to be illegible (cf. Program 6.2 and Section F). Tabular methods or truth tables might be better suited for this. A simple translator could be written to generate Lola code from such tables. The XC layout editor of Trianus has proven its value for experimenting with placement during the construction of an RC application. The speed of the tools is excellent and allows for an efficient, effective and iterative design cycle.

7 Related Work

The idea of using FPGAs to build reconfigurable computers must occur to every engineer who hears about FPGAs for the first time. It is a compelling idea and seems to have many advantages and promises great speedups over conventional software running on general purpose CPUs. When confronted with reality, however, euphoria quickly turns into disillusionment. The reason is the difficulty of programming such a system.

In this chapter, we present and discuss selected projects that are related to our work, either on the hardware side or on the software side. Both issues have received and still receive a great deal of attention by the research community and by commercial companies. For each project listed, we discuss the hardware involved (if any) and the software to program this hardware.

The list is not exhaustive, but presents one or two exponents of a particular approach to reconfigurable coprocessors and related synthesis software. If a project is not listed here, it does not mean that it is not relevant. The wealth of such projects simply makes a complete listing impractical. Steve Guccione's WWW list of custom computing machines lists over 50 entries and is growing steadily [Guc94].

7.1 Custom Computers

The first category we discuss is that of *custom computers*. We define a custom computer to consist of *several* FPGAs with attached RAM. Custom computers, in contrast to reconfigurable coprocessors, typically use a host computer only for data management (i.e. input and output). Custom computers are used to implement *large applications* in hardware.

7.1.1 Programmable Active Memories

One of the first custom computers was implemented at the Paris Research Laboratory of Digital Equipment Corp. [BRV89]. The pioneering work of the PRL group in the late 80s and early 90s culminated in a paper first published in 1993, titled "Programmable Active Memories: Reconfigurable Systems Come of Age" [VBR96]. This group had extensive experience implementing successful, high performance applications on their custom computer.

Perle-1 is the successor project to Perle-0 and consists of an array of 4x4 XC3090 FPGAs from Xilinx, representing about 100K logic gates. Attached to each side of that array is 1 MB of SRAM, for a total of 4 MB of local storage. The host computer is a DEC 3000 workstation with a TURBOchannel bus interface capable of delivering 100 MB/s.

Software

Applications are described in Modula-2, Lisp or C++ using proprietary tools. Placement is done by hand, or rather program statements. Partitioning the design onto the 16 available FPGAs is also done by hand. The commercial tools are only used to route the netlists of individual FPGAs and to generate the configuration bitstreams. We estimate the turnaround time of the tools to be in the range of tens of minutes.

7.1.2 Splash

Developed at the Supercomputing Research Center in Maryland, the Splash 1 and Splash 2 custom computers were among the earliest systems of their kind [GHK90, ABD92]. Splash 2 consists of up to 16 boards, each containing 16 Xilinx XC4010 FPGAs for computation, for a total of 2.5 million logic gate equivalents. The FPGAs are connected to each other via a serial path and to a 16x16 crossbar switch. In addition, each FPGA has access to 512 KB of SRAM. The host computer is a Sun SPARC-2. Data can be moved to and from Splash 2 at a rate of 50 MB/s.

Software

VHDL is used to write applications. Partitioning a design onto the multiple FPGAs is done manually. With a maximum of 256 FPGAs used for computation and 16 for communication, this is not a practical approach. Logic synthesis tools from Synopsys Inc. [Syn92] are used to compile the VHDL code into netlists. The Xilinx tools (ppr) are used to compile the final configuration bitstream. We have no reported performance numbers on compilation speed, but from discussions with other researchers, VHDL compilation is in the range of tens to hundreds of minutes and ppr is known to have long runtimes [Fie95]. We expect, therefore, that layout synthesis for Splash 2 is a rather lengthy process, allowing for only a few design iterations per day.

7.1.3 Teramac

Teramac [ACC95, ACC96, CAC96] of the Hewlett-Packard Laboratories in Palo Alto, California consists of 16 boards with a total of 1728 custom FPGAs (PLASMA) and 512 MB of RAM (64 independent 32-bit wide banks). The system provides at least the equivalent of 1 million logic gates. The most important advantage of the PLASMA FPGA is that it has abundant routing resources which allow for fully automatic placement and routing tools. Designs typically have clock rates of under 1 MHz. Communication with a host computer occurs via a SCSI interface and is thus limited to relatively low speeds.

Software

Interestingly, the group developed their own FPGA because the place and route tools for commercially available FPGAs had unacceptable execution times for a custom computing machine. PLASMA is rich in routing resources and layout synthesis is very fast, 3 seconds per FPGA. Compilation of a volume-visualization design consisting of a quarter million gates requires 30 minutes. This design would require about 25 XC4010 chips, for *each* of which the place and route time using commercial tools would be in that time frame. Synthesis, partitioning, placement and routing are *fully automatic* and hence the long compilation time for the whole application is acceptable.

7.2 Reconfigurable Coprocessors

This class of FPGA-based computers covers smaller devices, which are used in close cooperation with a host computer. Our Hades RC falls into this category.

7.2.1 Chameleon

Built in 1992 at ETH Zürich, Chameleon is a workstation using Algotronix CAL FPGAs and a MIPS CPU [HP92, Hee93]. One CAL chip is used to implement the control logic of the

workstation, such as the keyboard and mouse interface, the video controller and the network interface. A 2x3 array of CAL chips is used to implement a custom computer. No local memory is attached to the FPGAs, but the main memory of the host computer can be used via the processor. Data transfers are relatively slow, as no DMA is supported.

Software

Applications are described in the Debora HDL [Hee93]. The CALLAS layout synthesis software is used to generate layouts. Manual improvement of the placement is necessary, but no position hints in the HDL are allowed. Instead, a match tool is used to propagate information from a previously generated layout into a new one. The tools are very fast and design cycles in the order of tens of seconds to minutes are attainable. The fact that the control logic of Chameleon is described and synthesized using Debora and CALLAS proves the usefulness of the tools. CALLAS was a constant source of inspiration during the development of Hades. Hades' size is compared to that of CALLAS in Section 5.10.

7.2.2 PCI-Pamette

A smaller Programmable Active Memory machine, called PCI-Pamette, was developed at Digital's Systems Research Center in Palo Alto, California and is the third generation of the Perle family [Sha96]. The Pamette is a PCI-board featuring 4 Xilinx XC4010E FPGAs and 2 banks of SRAM, each with 128 KB. A fifth XC4010E implements the PCI interface, which allows for data throughputs close to the theoretical maximum of 133 MB/s. The board is used as a flexible, programmable I/O device, for instance, as a real-time data acquisition interface.

Software

The Pamette is programmed with the same software methodology as the Perle machines from DEC PRL. Commercial tools are used for routing and design cycles typically lie in the range of tens of minutes.

7.2.3 VCC's Reconfigurable Processing Unit

Virtual Computer Corp. manufactures a PCI board hosting a Xilinx XC6216 FPGA used as a reconfigurable coprocessor and a Xilinx XC4013E FPGA implementing the PCI interface [VCC97]. The card has two banks of SRAM, each with 256 KB.

Software

Design software for that board includes our Trianus/Hades system, as well as the Xilinx XACT step series 6000 and a relatively fast VHDL compiler.

7.3 Reconfigurable Processors

A different approach to custom computing is taken by research groups who investigate the combination of FPGAs with CPU cores. Most of these projects were started only recently and few results are available. None of these projects have reported a working implementation. The potential seems to be very promising, however.

7.3.1 PRISC

One of the earliest work in this field is PRISC (Programmable Instruction Set Computer) from Harvard University, Cambridge, Massachusetts [Raz94, RS94]. It consists of a CPU augmented with a programmable function unit (PFU), which has the equivalent die area of 1 KB of cache memory. A compiler exists, which analyzes the source code for operations that could be implemented in the PFU. A CPU with one PFU executes the SPEC integer benchmarks 22% faster. No compilation times are reported.

7.3.2 BRASS

The Berkeley Reconfigurable Architectures, Systems and Software [BRA96] project from the University of California, Berkeley, consists of a reconfigurable CPU and a C compiler generating configurations for it (similar to PRISC). The project was started in 1996 and no concrete results have been presented yet. The hardware consists of the Garp processor, which contains a MIPS-II core and an FPGA optimized for datapath applications. A modified C compiler is used to generate code and configuration bitstreams for Garp. Logic and layout synthesis is oriented towards datapaths in that it tries to preserve the hierarchical and structural information available in the design description. One of the project's goals is to have fast tools.

7.3.3 MATRIX

MATRIX [MD96, DeH96] from the Massachusetts Institute of Technology, Cambridge, is an array of programmable functional units operating on 8-bit operands and a dynamically programmable connection network. Data flow inside MATRIX can be steered by the application itself and instructions for the functional units can flow alongside the data. The architecture can be used to implement systolic arrays, VLIW processors, microcoded processors or any combination thereof. No compiler exists yet.

7.3.4 RaPiD

RaPiD from the University of Washington, Seattle, implements a reconfigurable, pipelined data path. Similar to MATRIX, it has programmable functional units, but not an equally flexible connection network. It is optimized for systolic array operations. The functional units have floating point capability (an ALU and a multiplier). The connection network is augmented with pipeline registers, leading to efficient, small systolic array structures. No compiler exists yet.

7.4 High-Level Hardware Description

Currently, there are three widely used ways to describe an application on a reconfigurable coprocessor or a custom computer. One uses schematic entry, one uses a hardware description language and one uses a program written in a general-purpose high-level programming language to produce the netlist representing the application. All three approaches are used by at least one of the presented projects in Sections 7.1 and 7.2. A fourth approach becoming more and more popular is compiling a high-level language directly into hardware, as is done in the projects in Section 7.3 and in the following ones.

7.4.1 PRISM

PRISM and PRISM-II [AS93, WAL93] from Brown University, Providence, Rhode Island, are large compiler systems that analyze C source code to find code sequences suitable to be

synthesized into hardware. The partitioning unit is a C function. The compiler produces VHDL code. The PRISM-II hardware consists of three Xilinx XC4010 and an AMD 29050 RISC CPU. The advantage of PRISM is that normal C programs not written specifically for a hardware implementation can profit from hardware acceleration. The disadvantage is that this speedup comes at a high cost in terms of long compilation times. Reported figures for compilation of small programs are in the minute range, not including synthesis of VHDL and place and route using commercial tools.

7.4.2 Transmogrifier-C

Transmogrifier is a custom computer from the University of Toronto, Ontario. To program applications, a C compiler was developed which maps C statements and expressions onto hardware [Gal95]. Only a subset of the C language is supported. In this project, C is used as a hardware description language. There are special semantics for each construct. The main advantage of this approach should be that a programmer does not have to learn a new language to describe the hardware. We consider this as mistaken, since the same language has different semantics depending on the context in which it is used.

7.4.3 nlc/Spyder

A similar approach as in Transmogrifier-C is taken in the “nlc” project from EPF Lausanne, Switzerland [Ise96, IS95]. A C++ compiler was developed for generating netlists from C++ programs. One goal of this project was to support simulation as well as synthesis. That is, the same C++ code can be compiled using a normal compiler to obtain an executable simulation of the hardware design. Commercial tools are used to perform layout synthesis.

7.5 The Need for Better Tools

Most of the described systems sooner or later require the use of commercial place and route tools to produce the final layout of a design. This is mainly due to the fact that the bitstream format for the used FPGAs was not available. Hence, these systems suffer from limited interactivity, long run times and practically no integration (cf. Section 5.1.4). Some groups would implement their own tools, if the bitstream format was made public.

Many groups report on the need to give manual hints to successfully place and route dense designs. The Teramac group successfully implemented a custom computer with fully automatic synthesis tools. The cost for this was the development of a new, routing-rich FPGA. For groups without this kind of resources, fast, interactive, integrated tools like the Trianus and Hades system provide a better way to achieve dense layouts quickly.

8 Summary, Conclusions and Outlook

8.1 What has been Accomplished?

A project covering both disciplines of computer science and computer engineering is very interesting and challenging. In this thesis we described the development of both *hardware* and *software* to build a complete *system* called *Hades*. A reconfigurable coprocessor based on the new Xilinx XC6200 FPGA architecture was developed, along with associated layout synthesis tools (place and route) and a rudimentary runtime system for coprocessor applications. No new algorithms were developed, no novel approaches in hardware design were invented. We made use of the available knowledge of algorithms and of the novel features the hardware provided, and combined these into a *usable, efficient, reliable and small system* for experimenting and implementing algorithms on a reconfigurable coprocessor.

8.2 Hades Hardware

The Hades hardware consists of a *reconfigurable coprocessor board* containing the new Xilinx *XC6200 FPGA*. The board is designed for the Ceres-2 workstation. It implements a *memory card interface* such that communication with the board uses the *same protocol* and has the *same latency as normal DRAM memory*. The configuration memory and cell states of the XC6200 FPGA are accessible via this interface as well. It is hence possible to use normal memory operations to read and write register values. On the board, local SRAM is used for fast storage of intermediate data. The board is integrated via its memory interface into the host operating system, which makes it easily and transparently accessible from within application code.

8.3 Hades Software

The Hades software implements a layout synthesis back-end for the Trianus framework. It consists of a *technology mapper*, a *constructive, deterministic placement algorithm*, a maze-running *routing algorithm*, a configuration bitstream generator, a driver for the Hades hardware and an *interface generator* to make a hardware design accessible to software.

For the same input, the placer produces the same layout. It places arrays constructively and almost always optimally. Expression trees are placed in a way to make them routable. The placer uses space generously and relies on hints given by the designer to achieve a satisfactory result.

The router uses a maze-running routing algorithm, which can be influenced in several ways. The shape of wave expansion, the routing resources used and the sequence in which nets are routed can be set. The router is scriptable, automating these tasks in an iterative design cycle.

The place and route *software preserve and make use of the hierarchical information* provided by the Trianus front-end. The software operates type-based, that is, it places and routes

the (proto)type of a circuit and then propagates this information to all components of that type in the design.

The synthesis tools support a *fast, interactive and iterative design cycle*. Hints in the Lola HDL description of a design can be used by the user to influence the produced result. The design cycle, starting with compiling Lola HDL code and ending with a finished layout, is very fast and usually takes under one minute on a Pentium-class PC. Compared to commercial tools, the Trianus/Hades software is a factor of 2 smaller, implements nearly the same functionality, requires at least a factor of 2 less memory and produces the result faster by a factor of 10 (this does not include HDL compilation by commercial tools).

8.4 Lola and Trianus

Lola is a simple, easy to learn hardware description language for describing digital circuits on a structural level. Position statements are very useful to guide a placement algorithm to obtain a good layout. Datapaths can be described concisely and the inclusion of hierarchy is essential for tackling large designs.

Trianus is a fast, robust, device independent framework for FPGA circuit design offering a general, yet simple data structure that is efficiently and comfortably altered using a flexible iteration mechanism. It supports and maintains the hierarchical information available in Lola and provides algorithms for manipulating the data structure respecting the hierarchy. The layout editor for the XC6200 can be used to optimize a circuit layout by hand and features fast view updates.

8.5 Conclusions

The Hades hardware implements a memory-card interface to a reconfigurable coprocessor. Such an interface is essential to implement a low-latency communication path from the host CPU to the application running on the reconfigurable coprocessor.

The Hades software tools make use of the type information and operate type-based, i.e. they operate on the (proto)type of a circuit and propagate the produced result to all instances of that type. The advantages of this approach are as follows:

- Synthesis results are predictable, as all instances (components) of a certain type have the same placement and the same routing.
- The results are produced quickly, since the algorithms have to perform the work only once and can then propagate the information.
- The algorithms scale to larger devices. When the complexity of a design increases, it will have more levels of hierarchy but not necessarily more components per hierarchy. When FPGA devices get bigger, the runtime of type-based tools therefore increases only linearly and not quadratically.

Automatically generated interfaces make accesses to the hardware easier and safer. They relieve the software programmer from knowing intricate details of the hardware implementation. Further, the designer of the hardware application is encouraged to produce a high-level interface to the application, since the necessary components are produced automatically.

Manual intervention during layout synthesis is still state of the art. Hades supports a fast and interactive design cycle and lets the designer's knowledge enter this cycle.

Interactive and incremental tools result in designs with the same or better quality than automatically generated designs in the same or less time.

A fast, easily usable interface to the reconfigurable coprocessor is essential for the performance of algorithms executed by it. In our opinion, reconfigurable computing has a great potential. However, more work on interface issues is needed and libraries of whole algorithms are to be built.

8.6 Outlook

We never knew as much about the subject of this thesis as now, at the end of the project. Therefore, we now know what we should have done differently and what worked out well. The following sections give some ideas on what could be done differently.

8.6.1 Hardware

As was seen in Chapter 6, the most pressing issue in the design of a reconfigurable coprocessor is the communication speed between the CPU and the FPGA. The Hades board has a memory card interface and has a relative speed advantage over the PCI-board of a factor of 6. However, the Hades board was developed for obsolete host hardware. An Alpha CPU from Digital Equipment Corporation can be clocked at 500 MHz and can issue 2 instructions per cycle. During an access to the FPGA over the 33 MHz PCI-bus, which takes an optimistic 60 ns, the Alpha could issue, in theory, 60 64-bit instructions. Hence, either the latency of communication must be improved drastically, or reconfigurable *co*-processors make only sense with slower CPUs, which are hard to come by these days.

In the future, we see several possibilities for improving the communication speed and reducing the latency:

- An RC could be mounted on a memory card and attached directly to the fast local memory bus of the CPU (as is done on Ceres), as opposed to the system bus (as is done on PCI).
- The RC could be attached to the CPU via a coprocessor interface.
- The FPGA could be moved directly onto the CPU die and incorporated into the data path of the CPU [BRA96, DeH96, ECF96, Raz94].

8.6.2 Software

It seems that due to its very nature, *soft*ware is never finished. We see several opportunities for improving the software in Hades.

Placer

Although the deterministic placement algorithm currently used in Hades is very fast, the produced results must be improved manually. Slower, but smarter algorithms should be evaluated to improve the placement within types and the placement of instances. For random logic, a stochastic algorithm could be used, for arrays, the approach used in the Hades placer gives good results, and for placing whole instances, a min-cut approach could be used.

Router

The routing algorithm is currently the bottleneck in the Hades software design cycle. It could be improved by implementing separate algorithms for special routing cases such as straight or L-shaped connections. The speed of wave-expansion of the maze-running algorithm could be improved by letting the wave spread from both ends (from the single target and from all

source points) [DM95]. Currently, the router treats multi-point nets as several two-point nets. Research results [BKV96] show that a special treatment of multi-point nets can be advantageous.

Timing and Automatic Retiming

In general, layout synthesis tools should incorporate timing information to make better decisions about the placement of cells and the routing of nets.

An algorithm for the automatic retiming (insertion of pipeline registers) could be useful during the evaluation of a circuit's performance. Approaches such as [LSC96] and [Tra95] show promising results.

8.6.3 Hardware/Software Co-Design Issues

A topic not covered and not even mentioned in this thesis except in this section is hardware/software co-design. This is a very active area of research and promises a new way of designing electronic systems. Design partitioning is done either automatically [Bli96] or by the designer, but with extensive tool support. This thesis provides means to describe the hardware part bottom-up and provides a sufficiently high-level abstraction of the hardware to the software programmer. However, there is no methodology or tool supporting the partitioning of an application, i.e. to tell what should be realized in hardware and what in software.

A Syntax of Lola

The following is a definition of the syntax of the Lola hardware description language. It is given in EBNF notation (extended Backus-Naur form).

```
Identifier =        Letter {Letter | Digit} [" ' "].
Integer =           Digit {Digit}.
LogicValue =        "'0" | "'1".

BasicType =         "BIT" | "TS" | "OC".
SimpleType =        BasicType | Identifier ["(" ExpressionList ")"].
ExpressionList =    Expression {"," Expression}.

Type =              { "[" Expression "]" } SimpleType.
ConstDeclaration =  Identifier ":=" Expression ";".
VarDeclaration =    IdList ":" Type ";".
IdList =            Identifier {"," Identifier}.

Selector =          {"." Identifier | "." Integer |
                    "[" Expression [ ".." Expression] "]"}.
Factor =            Identifier Selector | LogicValue | Integer |
                    "~" Factor | "(" Expression ")" |
                    "MUX" "(" Expression ":" Expression
                    "," Expression ")" |
                    "SR" "(" Expression "," Expression ")" |
                    "LATCH" "(" Expression "," Expression ")" |
                    "REG" "(" [Expression ":"][Expression ","]
                    Expression ")" .
Term =              Factor {("*" | "/" | "DIV" | "MOD" | "^") Factor}.
Expression =        Term {("+" | "-") Term}.

Assignment =        Identifier Selector ":=" [Condition "|"] Expression.
Condition =         Expression.
Relation =          Expression ("=" | "#" | "<" | "<=" |
                    ">" | ">=") Expression.
```

```
IfStatement =       "IF" Relation "THEN" StatSequence
                    {"ELSIF" Relation "THEN" StatSequence}
                    ["ELSE" StatSequence]
                    "END" .
ForStatement =      "FOR" Identifier ":=" Expression ".." Expression
                    "DO" StatSequence "END" .
UnitAssignment =    Identifier Selector "(" ParameterList ")".
ParameterList =     Expression | Constructor.
Constructor =       "[" Expression {"," Expression} "]" .
PosAssignment =     Identifier Selector "::" Position.
Position =          Expression {"," Expression} |
                    "[" Position {";" Position} "]" .
Statement =         [Assignment | UnitAssignment | PosAssignment |
                    IfStatement | ForStatement].
StatSequence =      Statement {";" Statement}.

InType =            {"[" Expression "]"} "BIT".
InOutType =         {"[" Expression "]"} ("TS" | "OC").
OutType =           {"[" Expression "]"} ("BIT" | "TS" | "OC").
TypeDeclaration =   "TYPE" Identifier ["*"] ["(" IdList ")"] ";"
                    ["CONST" {ConstDeclaration}]
                    ["IN" {IdList ":" InType ";"}]
                    ["INOUT" {IdList ":" InOutType ";"}]
                    ["OUT" {IdList ":" OutType ";"}]
                    ["VAR" {VarDeclaration}]
                    ["BEGIN" [StatSequence]
                    "END" Identifier.
ImportList =        "IMPORT" Identifier {"," Identifier} ";" .
Module =            "MODULE" Identifier ";" [ImportList]
                    {TypeDeclaration ";"}
                    ["CONST" {ConstDeclaration}]
                    ["IN" {IdList ":" InType ";"}]
                    ["INOUT" {IdList ":" InOutType ";"}]
                    ["OUT" {IdList ":" OutType ";"}]
                    ["VAR" {VarDeclaration}]
                    ["CLOCK" Expression ";"]
                    ["BEGIN" StatSequence]
                    "END" Identifier "." .
```

B Schema of Hades Coprocessor Board

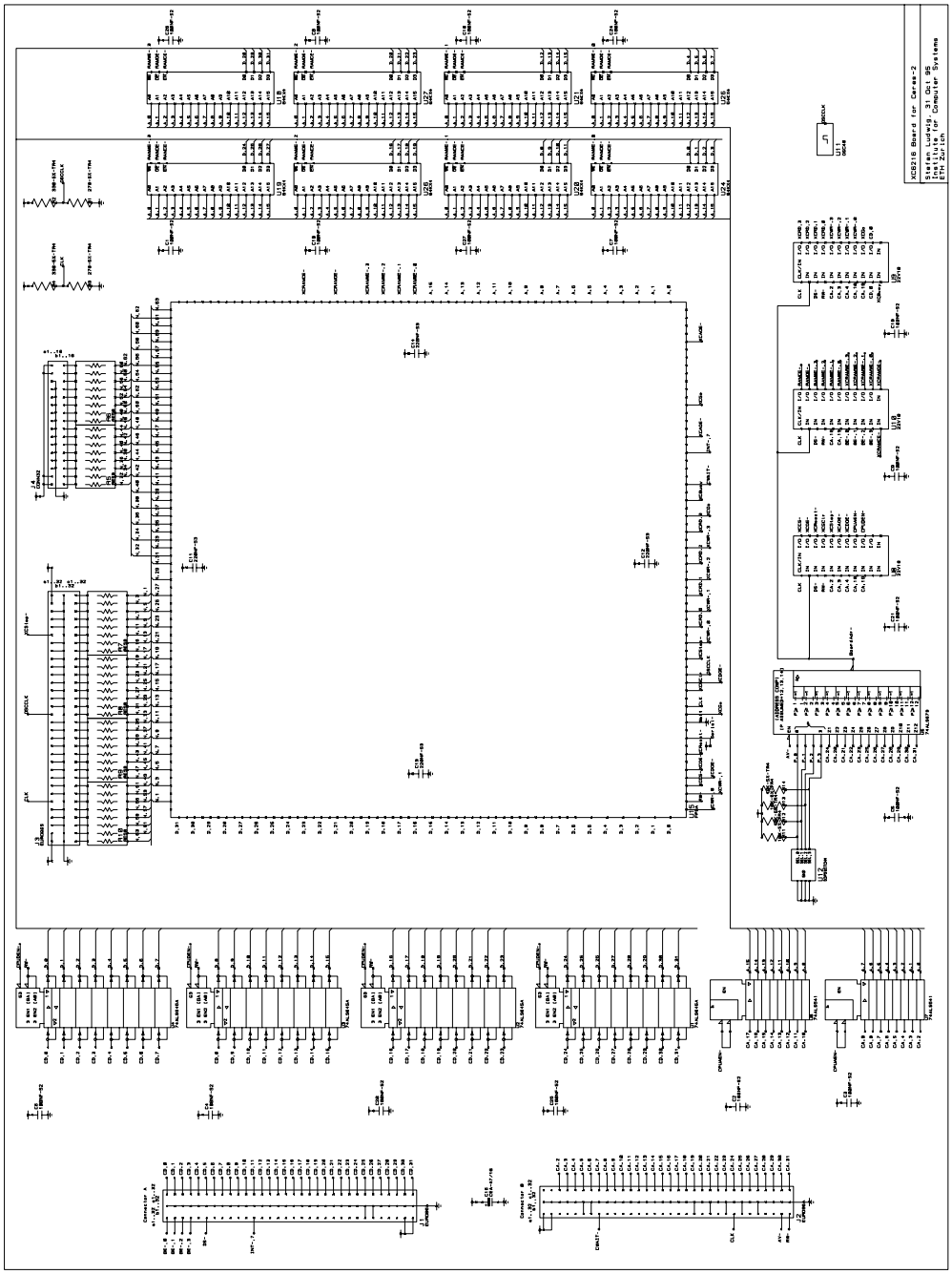

Figure B.1: Schema of Hades RC Board

C Photograph of Hades Coprocessor Board

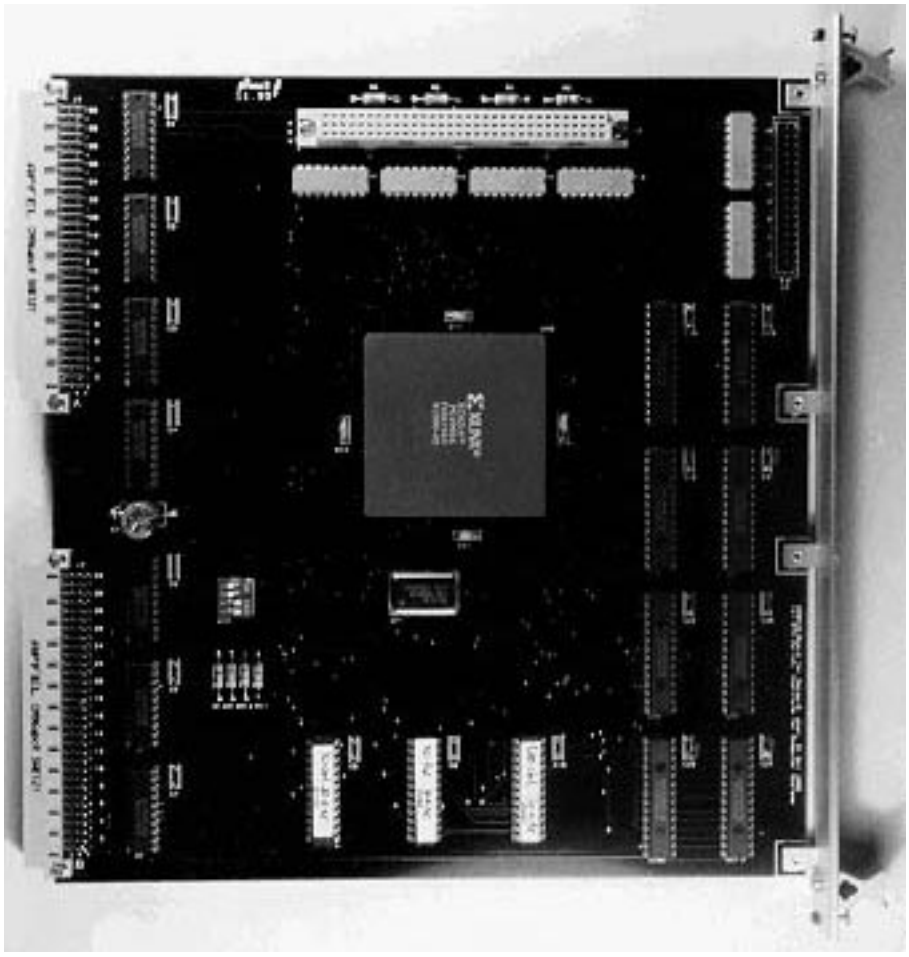

Figure C.1: Photograph of Hades RC Board

D Components for a Hades Board

1	XC6216 FPGA, PGA 299
1	PGA Socket 299 pin
8	SRAM 64K x 4 bit with output enable, 15 ns, DIP-28 300 mil Motorola MCM 6209C P15
8	DIP-28 300mil socket with 100 nF capacitor
3	22V10, 7.5 ns speed grade, DIP-24 300 mil Lattice GAL22V10B-7LP
3	DIP-24 300 mil socket with 100 nF capacitor
2	A(L)S541-J Octal buffer and line driver with 3-state output
4	A(L)S645-J Octal bus transceiver with 3-state output
1	A(L)S679-J 12-bit address comparator
7	DIP-20 300 mil socket with 100 nF capacitor
1	40 MHz oscillator
2	330 Ohm resistor
2	270 Ohm resistor
4	10 kOhm resistor
6	8 resistors in a DIP-16 300 mil
6	DIP-16 300 mil socket

4	0.22 μF capacitor
18	100 nF capacitor (if sockets with capacitors are not available)
1	47 μF capacitor
2	96-pin Euroconnector (male) angled
1	96-pin Euroconnector (female) straight
1	32-pin connector (male, 100 mil pitch)
1	4-bit DIP switch

E Hades RC Board Decoder

The following three programs list in full the Lola code for the PALs implementing the control interface on the Hades RC board. The first program (DecoderXCCtrl) lists the code for the XC6216 control PAL, the second (DecoderRAMCtrl) lists the code for the SRAM access controller and the third (DecoderXCRW) lists the code for the communication port controller.

```
TYPE DecoderXCCtrl;                 (* implemented in a PAL22V10, U8 *)
    IN
        Clk: BIT;
        BoardAdr': BIT;             (* board is selected *)
        A19, A18, A4, A3, A2: BIT;  (* address lines needed for decoding *)

        CPURW', CPUDS': BIT;        (* CPU: read/write, data strobe *)
        RESET': BIT;                (* master reset *)
    OUT
        XCCS', XCOE': BIT;          (* XC: is selected, may drive pins *)
        XCAOE', XCDOE': BIT;        (* XC: may drive A/D buses *)
        XCReset', XCGClr: BIT;      (* XC: reset, global clear *)
        CPUAEN', CPUDEN': BIT;      (* CPU drives the A/D-bus *)
        XCStep: BIT;                (* single stepping *)
    VAR
        select, write, XCSel, RAMSel, PortSel: BIT;
        oe: BIT;                    (* register *)
BEGIN
    select := ~CPUDS' * ~BoardAdr';
    write := select * ~CPURW';

    XCSel := select * ~A19 * ~A18;      (* 00'xxx *)
    RAMSel := select * ~A19 * A18;      (* 01'xxx *)
    PortSel := select * A19 * A18 * ~A4;  (* 11'0xx *)

    XCCS' := ~XCSel;
    (* 10'000 disable, 10'001 enable OE' *)
    oe := REG(write * A19 * ~A18 * ~A4 * ~A3, A2));
    XCOE' := ~oe;

    (* 10'010 *)
    XCReset' := ~(write * A19 * ~A18 * ~A4 * A3 * ~A2) * RESET';
    (* 10'011 *)
    XCGClr := write * A19 * ~A18 * ~A4 * A3 * A2;
    (* 10'100, generate one clock pulse, high -> low -> high *)
    XCStep := ~(write * A19 * ~A18 * A4 * ~A3 * ~A2);

    (* CPU uses data bus *)
    CPUDEN' := ~(XCSel + RAMSel + PortSel);
    (* CPU drives address bus *)
    CPUAEN' := ~(XCSel + RAMSel);

    (* XC may drive data bus *)
    XCDOE' := ~(oe * ~XCSel * ~RAMSel * ~(PortSel * ~CPURW'));
    (* XC may drive address bus *)
    XCAOE' := ~(oe * ~XCSel * ~RAMSel)
END DecoderXCCtrl;
```

```
TYPE DecoderRAMCtrl;                (* implemented in a PAL22V10, U10 *)
    IN
        Clk: BIT;
        BoardAdr': BIT;             (* board is selected *)
        (* address lines needed for decoding *)
        A19: BIT;
        A18: BIT;
        CPURW': BIT;                (* CPU: read/write *)
        CPUDS': BIT;                (* CPU: data strobe *)
        CPUBE': [4] BIT;            (* CPU: byte enables 0..3 *)
        XCRAMCE': BIT;              (* XC: selects SRAM *)
        XCRAMOE': BIT;              (* XC: reads SRAM *)
        XCRAMWE': [4] BIT;          (* XC: write enables 0..3 for SRAM *)
    OUT
        RAMOE': BIT;                (* RAM: read enable *)
        RAMCE': BIT;                (* RAM: chip enable *)
        RAMWE': [4] BIT;            (* RAM: write enable 0..3 *)
    VAR
        XCSel, RAMSel, PortSel: BIT;
        select: BIT;
BEGIN
    select := ~CPUDS' * ~BoardAdr';

    XCSel := select * ~A19 * ~A18;      (* 00'xxx *)
    RAMSel := select * ~A19 * A18;      (* 01'xxx *)
    PortSel := select * A19 * A18;      (* 11'xxx *)

    (* for RAM access: XC is NOT selected and
        CPU selects RAM and reads/writes
        or CPU doesn't select RAM and XC reads/writes *)
    RAMCE' := ~(~XCSel * (RAMSel + (~PortSel * ~XCRAMCE')));
    RAMOE' := ~(~XCSel * ((RAMSel * CPURW') +
        (~RAMSel * ~PortSel * ~XCRAMOE')));
    FOR i := 0..3 DO
        RAMWE'.i := ~(~XCSel * ((RAMSel * ~CPUBE'.i * ~CPURW')
            + (~RAMSel * ~PortSel * ~XCRAMWE'.i)))
    END
END DecoderRAMCtrl;
```

```
TYPE DecoderXCRW;                    (* implemented in a PAL22V10, U9 *)
    IN
        Clk: BIT;
        BoardAdr': BIT;              (* board is selected *)
        (* address lines needed for decoding *)
        A19, A18, A4, A3, A2: BIT;
        CPURW': BIT;                 (* CPU: read/write *)
        CPUDS': BIT;                 (* CPU: data strobe *)
        XCBusy: BIT;                 (* XC: busy flag *)
        CPUD0In: BIT;                (* CPU: D.0 input *)
    INOUT CPUD0: TS;                 (* CPU: D.0 tri-state output *)
    OUT
        XCRD: [4] BIT;               (* XC: decoded read signals 0..3 *)
        XCWR': [4] BIT;              (* XC: decoded read/write signals 0..3 *)
        XCGo: BIT;                   (* XC: go flag *)
    VAR
        select, XCSel, RAMSel, PortSel, ComSel: BIT;
        Port: [4] BIT;
        busy: BIT;                   (* register *)
BEGIN
    select := ~CPUDS' * ~BoardAdr';
    XCSel := select * ~A19 * ~A18;   (* 00'xxx *)
    RAMSel := select * ~A19 * A18;   (* 01'xxx *)
    (* 10'101 *)
    ComSel := select * A19 * ~A18 * A4 * ~A3 * A2;
    (* 11'0xx *)
    PortSel := select * A19 * A18 * ~A4;

    Port.0 := PortSel * ~A3 * ~A2;   (* 11'000 *)
    Port.1 := PortSel * ~A3 * A2;    (* 11'001 *)
    Port.2 := PortSel * A3 * ~A2;    (* 11'010 *)
    Port.3 := PortSel * A3 * A2;     (* 11'011 *)
    FOR i := 0..3 DO
        XCRD.i := Port.i * CPURW';
        XCWR'.i := ~(Port.i * ~CPURW')
    END;

    XCGo := REG(MUX(ComSel * ~CPURW': XCGo, CPUD0In));
    busy := REG(MUX(ComSel * CPURW': busy, XCBusy));
    CPUD0 := ComSel * CPURW' | busy
END DecoderXCRW;
```

F Wotan Microprocessor

F.1 Architecture and Principle of Operation

F.1.1 Overview

Wotan is a small microprocessor designed by N. Wirth. It contains a 24-bit wide data path, realized as 24 ALU slices, and a 16-bit wide address path. The data path contains 8 registers and has support for a multiply/divide step. Figure F.1 shows the floorplan of Wotan, which is implemented by the layout shown in Figure F.8.

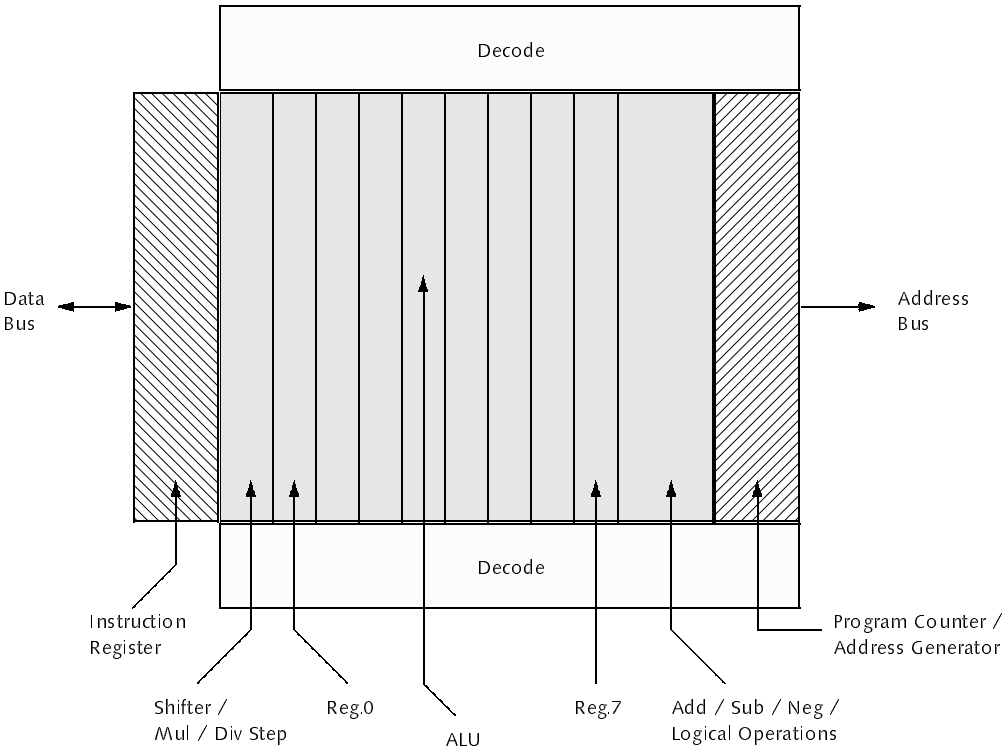

Figure F.1: Floorplan of Wotan Microprocessor

F.1.2 Arithmetic Logic Unit

The register file consists of 8 registers, each 24-bits wide (Reg.0 .. Reg.7 in Figure F.1). On a XC6216 there is not enough room to accommodate 32-bit wide registers in addition to the decoding circuitry. The registers can be loaded with a value coming from the data bus, the instruction register or the program counter. Operations on the registers include addition, subtraction, shifting, and, or, exclusive-or, negation and support for multiplication and division

steps. Since the XC6200 architecture does not have tri-state buses inside the chip, reading and writing the register file is accomplished through a series of multiplexers. One slice of three registers is shown in Figure F.2. Data lines run horizontally and control lines run vertically. The multiplexers' select lines are driven by control lines from the decoding circuitry above and below the ALU. They determine, which register is allowed to write its value to the x or y "bus".

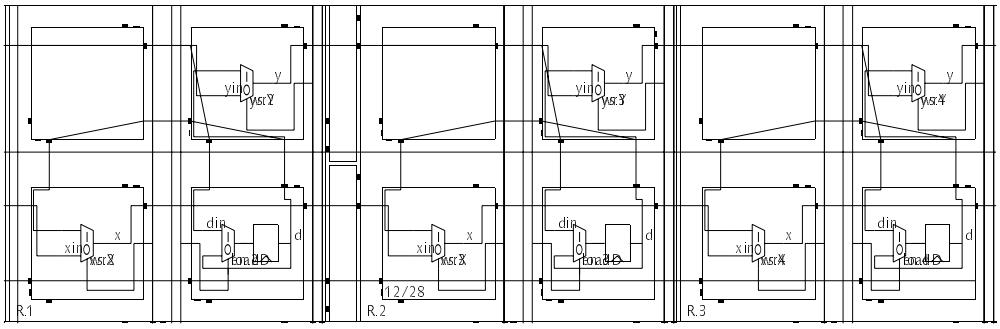

Figure F.2: Register Slice

The layout of one complete ALU slice is shown in Figure F.7. A data flow diagram of the ALU is shown in Figure F.3. An ALU instruction has two source (x, y) and one destination register (z).

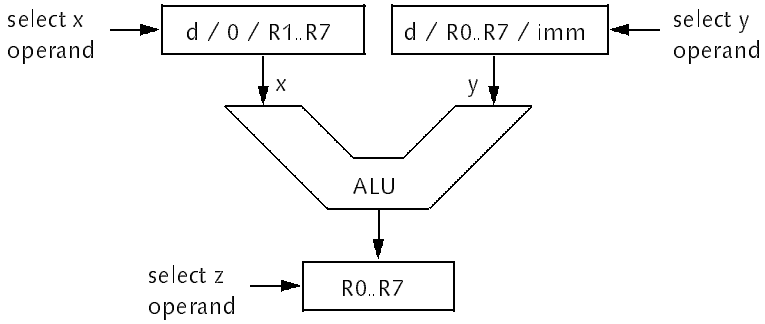

Figure F.3: Data Flow

F.1.3 Control Unit

The control unit, consisting of the program counter and address generator circuitry, is 16-bits wide and is used to address external memory. The address used in a particular step is either the value of the program counter (for fetching the next instruction), the value from the instruction register (for jumps) or the value from the ALU (for return jumps). Figure F.4 shows two bits of the control unit. The program counter is shown on the left and the multiplexers for the address selection are shown on the right.

F.1.4 Decoders and Instructions

Control signals for the various multiplexers in the ALU and the program counter are generated by the decoding circuits below and above the ALU and the control unit. The decoding circuitry takes as inputs the instruction register holding the current instruction. Wotan implements the

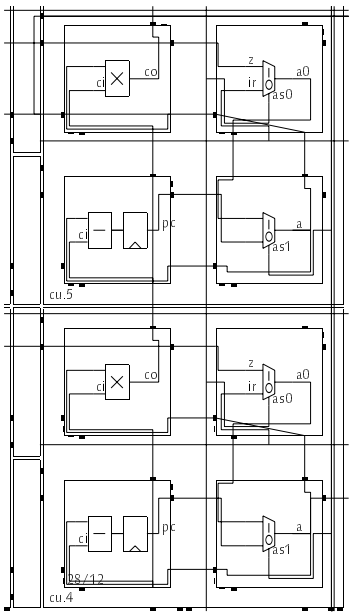

Figure F.4: Control Unit Slice

instructions shown in Table F.1. Rows with an “i” indicate instructions with or without an immediate operand and rows with an “n” indicate instructions which do or do not negate one operand.

F.1.5 Sequencer

The state machine controlling the operation of the microprocessor has three phases (states) shown in Figure F.5. In phase 0 (ph0'), ALU instructions and branches are executed and a new instruction is loaded into the instruction register (fetch). If the current instruction is a load or a store instruction it is executed in phase 1 (ph1), during which memory is accessed. After phase 1, a new instruction is loaded in phase 2 (ph2).

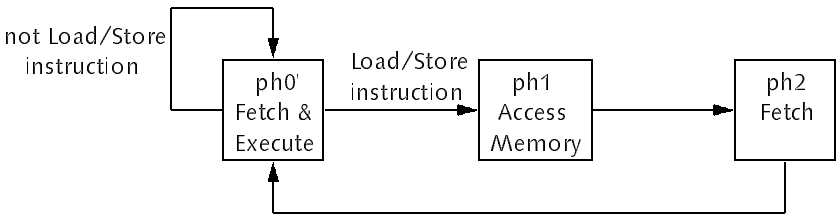

Figure F.5: State Machine

F.2 Lola Code

In the following, the complete Lola HDL description of Wotan together with placement code and more detailed explanations are given.

The ALU consists of 24 bit-slices. Each slice contains an ALU with 2 inputs x and y and output z. The inputs come from the 8 registers. The first register R0 is special, the others

Instruction	IR.23 .. IR.18
ADD	0i 0n00
AND	0i 0n01
OR	0i 0n10
XOR	0i 0n11
SHL	0i 1n00
SHR	0i 1n01
LD	10 0nxx
ST	10 1nxx
BR	11 1000
BEQ	11 0001
BNE	11 1001
BLT	11 0010
BGE	11 1010
BLE	11 0011
BGT	11 1011
BCS	11 0100
BCC	11 1100
BSR	11 0110
RET	11 0111

Table F.1: Wotan Instructions

are implemented as RegCells with two output muxes. Figure F.6 shows the input, output and control signals of one ALU-slice.

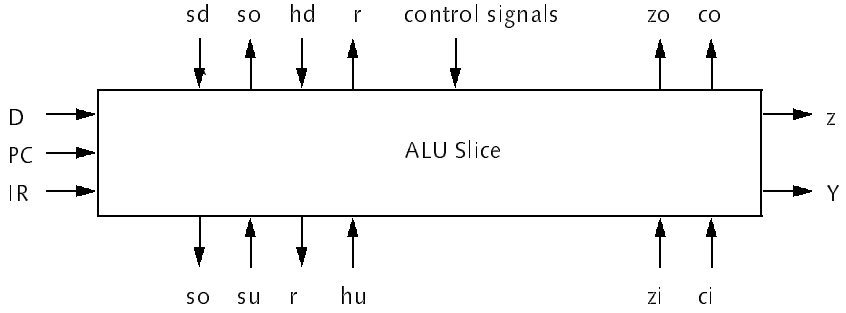

Figure F.6: ALU Slice Signals

RegCell: din = data input to register d from input bus; loadD = register load enable; xin, yin = input to output muxes; wrX, wrY: output mux selectors, 0 = pass xin, yin input to x, y output, 1 = feed register value to x, y.

The register input din flows through a mux controlled by ds (data select), selecting either the ALU output or input from memory (D), yielding so, and then through a shift mux controlled by shen. shen = shift enable; shd = shift down; sd, su = shift mux inputs from next higher or lower bit-slice. The output d of the shift mux goes to the registers.

A second shift path is used for multiply and divide instructions, in which register 0 r plays an exceptional role: The second shift-mux has inputs hd and hu for shifting up or down.

The ALU part implements inversion, exclusive and inclusive OR, and AND. Controls are: neg: complement y input; xor: select XOR; or: select OR; and: select AND.

The y input is selected to be either the register output y (called Y) or to come from the IR-register (immediate operand).

ci and co are the carries, and zi and zo the chain of AND-gates to determine whether all ALU outputs are zero. A multiplexer at the ALU-output allows to feed in the PC-value (for branch-subroutine).

The control unit consists of 16 PC-slices. Each slice contains a bit of the PC register and two muxes. The address output used to address memory is determined by the selectors as1 and as0 as shown in Table F.2. The control unit also contains the program counter. Its next value is also shown in Table F.2.

as1/as0	Adr Out	PC
0x	PC	PC+1
10	IR	IR (call)
11	ALU.z	ALU.z (return)

Table F.2: Control Unit

```
MODULE Wotan;
    (* register d and simulated register buses x, y *)
    TYPE RegCell(Place);
        (* data in, x/y bus in, load register, write bus x/y *)
        IN din, xin, yin, loadD, wrX, wrY: BIT;
        OUT x, y: BIT;                  (* x/y bus *)
        VAR d: BIT;                     (* register *)
    BEGIN
        d := REG(loadD, din);
        x := MUX(wrX: xin, d);          (* conditionally write d to x bus *)
        y := MUX(wrY: yin, d);          (* conditionally write d to y bus *)

        IF Place = 1 THEN d :: 1, 0; x :: 0, 0; y :: 1, 1 END
    END RegCell;

    (* alu bit slice with 8 registers *)
    TYPE ALUslice(Place);
        CONST Regs := 8;                (* 8 registers *)
        IN
            din, ir, pc: BIT;           (* data in, inst. reg, prog. counter *)
            su, sd, hu, hd: BIT;        (* shift data up/dn, mult/div up/dn *)

            ci, zi: BIT;                (* carry in, zero in *)
            (* data select, immediate select, pc select, shift enable, shift up/dn *)
            ds, im, pcs, shen, shd: BIT;
            (* and, or, xor, negate *)
            and, or, xor, neg: BIT;
            (* register load enable, output mux selectors *)
            en, xs, ys: [Regs] BIT;
        OUT
            z, Y, so, r, co, zo: BIT;   (* r for register 0 *)
        VAR
            d, x, y, u, zero, pz, shift, shiftr, rin, ren, r0in,
            Y1, sum, a, o, Xor, And: BIT;
            (* register bitslices, 0th register special (r) *)
            R: [Regs-1] RegCell(Place);
    BEGIN
        pz := MUX(pcs: z, pc);          (* select alu output or pc *)
        so := MUX(ds: pz, din);         (* select alu output, pc or data input *)

        shift := MUX(shd: su, sd);      (* shift up or down *)
        d := MUX(shen: so, shift);      (* shift select *)
        shiftr := MUX(shd: hu, hd);
        rin := MUX(shen: d, shiftr); ren := en.0+shen;
        r := REG(ren, rin);             (* reg 0 *)
        zero := '0; r0in := ys.0*r;
        R.0(d, zero, r0in, en.1, xs.1, ys.1);  (* reg 1 *)
        FOR i := 1..6 DO                (* reg 2-7 *)
            R.i(d, R[i-1].x, R[i-1].y, en[i+1], xs[i+1], ys[i+1])
        END ;
        x := R.6.x; Y := R.6.y;         (* x and y register bus *)
        Y1 := MUX(im: Y, ir);           (* select instruction or normal register *)
```

```
    y := Y1-neg;                    (* negate *)
    u := x-y;                       (* half sum *)
    (* alu operations *)
    sum := u-ci; a := x*y; o := x+y;
    Xor := MUX(xor: sum, u); And := MUX(and: Xor, a);
    z := MUX(or: And, o);
    co := MUX(u: x, ci);            (* carry out *)
    zo := ~z*zi;                    (* zero chain *)

    IF Place = 1 THEN
        FOR i := 0 .. 6 DO R.i :: 4+i*2, 0 END;
        so :: 0, 0; shift :: 0, 1;
        d :: 1, 0; rin :: 1, 1;
        shiftr :: 2, 0; ren :: 2, 1;
        r :: 3, 0; r0in :: 3, 1;
        zero :: 4, 1;
        x :: 16, 0;
        Y :: 17, 1;
        Y1 :: 18, 0; y :: 18, 1;
        co :: 19, 0; u :: 19, 1;
        sum :: 20, 0; Xor :: 20, 1;
        a :: 21, 0; And :: 21, 1;
        o :: 22, 0; z :: 22, 1;
        pz :: 23, 0; zo :: 23, 1
    END
END ALUslice;
```

```
(* incrementable program counter *)
TYPE PCslice(Place);
    IN ir, z, ci, as1, as0: BIT;        (* instr. reg, alu output, carry in, selectors *)
    OUT pc, co, a: BIT;                 (* pc, carry out, address *)
    VAR a0: BIT;
BEGIN
    (* increment pc *)
    pc := REG(a - ci); co := a * ci;
    a0 := MUX(as0: ir, z);              (* instr. reg or alu output *)
    a := MUX(as1: pc, a0);              (* instr. reg, alu output or pc *)

    IF Place = 1 THEN pc :: 0, 0; co :: 0, 1; a :: 1, 0; a0 :: 1, 1 END
END PCslice;

(* 3:8 decoder *)
TYPE Decoder(Place);
    IN a: [3] BIT;
    OUT y: [8] BIT;
BEGIN
    y.0 := (~a.2 * ~a.1) * ~a.0;        (* 000 *)
    y.1:= (~a.2 * ~a.1) * a.0;          (* 001 *)
    y.2 := (~a.2 * a.1) * ~a.0;         (* 010 *)
    y.3:= (~a.2 * a.1) * a.0;           (* 011 *)
    y.4:= (a.2 * ~a.1) * ~a.0;          (* 100 *)
    y.5:= (a.2 * ~a.1) * a.0;           (* 101 *)
    y.6:= (a.2 * a.1) * ~a.0;           (* 110 *)
    y.7:= (a.2 * a.1) * a.0;            (* 111 *)

    IF Place = 1 THEN FOR i := 0 .. 7 DO y.i :: 2*i, 0 END END
END Decoder;

(* 3:8 decoder with enable *)
TYPE EnDecoder(Place);
    IN en: BIT; a: [3] BIT;
    OUT y: [8] BIT;
BEGIN
    y.0:= (~a.2 * ~a.1) * (~a.0 * en);  (* 000 *)
    y.1:= (~a.2 * ~a.1) * (a.0 * en);   (* 001 *)
    y.2:= (~a.2 * a.1) * (~a.0 * en);   (* 010 *)
    y.3:= (~a.2 * a.1) * (a.0 * en);    (* 011 *)
    y.4:= (a.2 * ~a.1) * (~a.0 * en);   (* 100 *)
    y.5:= (a.2 * ~a.1) * (a.0 * en);    (* 101 *)
    y.6:= (a.2 * a.1) * (~a.0 * en);    (* 110 *)
    y.7:= (a.2 * a.1) * (a.0 * en);     (* 111 *)

    IF Place = 1 THEN FOR i := 0 .. 7 DO y.i :: 2*i, 0 END END
END EnDecoder;

CONST
    DataN := 24; AddrN := 16;
    Place := 1; AluXOff := 4; AluYOff := 4;
IN
```

```
        AOE': BIT;
    INOUT
        A: [AddrN] TS;
        D: [DataN] TS;
    OUT
        RAMWE': [4] BIT;
        RAMOE': BIT;
    VAR
        IR: [DataN] BIT;
        alu: [DataN] ALUslice(Place);
        cu: [AddrN] PCslice(Place);
        Dx, Dy: Decoder(Place);
        Dz: EnDecoder(Place);
        N, Z, C: BIT;
        ds, im, pcs, shen, shd, and, or, xor, neg: BIT;
        (* state machine *)
        LS, ph0', ph1, ph2: BIT;
        (* controls *)
        ren, iren, cond, as0, as1: BIT;
        zero, one, we': BIT;
        zeroes: [DataN-AddrN] BIT;
BEGIN
    (* instruction register and decoders *)
    FOR i := 0..DataN-1 DO IR.i := REG(iren, D.i) END;
    Dz(ren, IR[15..17]);                (* destination register select *)
    Dx(IR[12..14]);                     (* src register 1 select *)
    Dy(IR[0..2]);                       (* src register 2 select *)
    (* ALU slices *)
    zero := '0; one := '1;
    alu.0(D.0, IR.0, cu.0.pc, alu.23.r, alu.1.so,
        zero, alu.1.r, neg, one,
        ds, im, pcs, shen, shd, and, or, xor, neg, Dz.y, Dx.y, Dy.y);
    FOR i := 1 .. 11 DO
        alu.i(D.i, IR.i, cu.i.pc, alu[i-1].so, alu[i+1].so,
            alu[i-1].r, alu[i+1].r, alu[i-1].co, alu[i-1].zo,
            ds, im, pcs, shen, shd, and, or, xor, neg, Dz.y, Dx.y, Dy.y)
    END;
    FOR i := 12 .. AddrN-1 DO
        alu.i(D.i, IR.11, cu.i.pc, alu[i-1].so, alu[i+1].so,
            alu[i-1].r, alu[i+1].r, alu[i-1].co, alu[i-1].zo,
            ds, im, pcs, shen, shd, and, or, xor, neg, Dz.y, Dx.y, Dy.y)
    END;
    FOR i := AddrN .. DataN-2 DO
        zeroes[i-AddrN] := '0;
        alu.i(D.i, IR.11, zeroes[i-AddrN], alu[i-1].so, alu[i+1].so,
            alu[i-1].r, alu[i+1].r, alu[i-1].co, alu[i-1].zo,
            ds, im, pcs, shen, shd, and, or, xor, neg, Dz.y, Dx.y, Dy.y)
    END;
    zeroes[DataN-1-AddrN] := '0;
    alu[DataN-1](D[DataN-1], IR.11, zeroes[DataN-1-AddrN],
        alu[DataN-2].so, alu[DataN-1].co,
        alu[DataN-2].r, alu.0.so, alu[DataN-2].co, alu[DataN-2].zo,
```

```
    ds, im, pcs, shen, shd, and, or, xor, neg, Dz.y, Dx.y, Dy.y);

(* status flags *)
N := REG(ren, alu[DataN-1].z);          (* ALU.z negative? *)
Z := REG(ren, alu[DataN-1].zo);         (* ALU.z zero? *)
C := REG(ren, alu[DataN-1].co);         (* carry set? *)

(* PC and address generation *)
cu.0(IR.0, alu.0.z, iren, as1, as0);
A.0 := AOE' | cu.0.a;
FOR i := 1..AddrN-1 DO
    cu.i(IR.i, alu.i.z, cu[i-1].co, as1, as0);
    A.i := AOE' | cu.i.a
END;
FOR i := 0..DataN-1 DO D.i := RAMOE' | alu.i.Y END;

(* control signals *)
ds := ph1 * ~IR.21;                     (* data select, load instruction *)
im := IR.22;                            (* immediate operand *)
(* select pc for register file: branch subroutine and link *)
pcs := IR.23 * IR.22 * ~IR.21 * IR.20 * IR.19 * ~IR.18;
shen := ~IR.23 * IR.21;                 (* shift instruction *)
shd := IR.18;
and := ~IR.23 * ~IR.19 * IR.18;
or := ~IR.23 * IR.19 * ~IR.18;
xor := ~IR.23 * IR.19 * IR.18;
neg := IR.20;

(* state machine *)
LS := IR.23 * ~IR.22;                   (* load or store *)
ph0' := REG(~iren);
ph1 := REG(~ph0'*LS);                   (* load or store phase *)
ph2 := REG(ph1);

iren := ~ph0'*~LS + ph2;                (* instruction reg. enable *)
ren := ~ph0'*~IR.23 + ph1*~IR.21;       (* reg. enable: ALU instr. or load *)

RAMOE' := ~(iren + ph1*~IR.21);         (* instruction fetch or load *)
we' := ~(ph1 * IR.21);                  (* store *)
RAMWE'.0 := we';
RAMWE'.1 := we';
RAMWE'.2 := we';

(* condition set (IR.21 = 0) or cleared (IR.21 = 1) *)
cond := (C*IR.20 + N*IR.19 + Z*IR.18) - IR.21;
(* return or previous was load/store *)
as0 := IR.18 + ph1;
(* branch or return or previous was load/store *)
as1 := IR.23*IR.22 * (cond + IR.20*IR.19) + ph1;

IF Place = 1 THEN
    FOR i := 0 .. DataN-1 DO alu.i :: AluXOff, AluYOff+2*i END;
```

```
        FOR i := 0 .. AddrN-1 DO cu.i :: AluXOff+24, AluYOff+2*i END;
        FOR i := AddrN .. DataN-1 DO
            zeroes[i-AddrN] :: AluXOff+24, AluYOff+2*i
        END;
        Dx :: AluXOff+2, AluYOff-4;
        Dy :: AluXOff+3, AluYOff-2;
        Dz :: AluXOff+3, AluYOff+2*DataN;
        zero :: AluXOff+2, AluYOff-1; one :: AluXOff+23, AluYOff-1;
        C :: AluXOff+19, AluYOff+2*DataN;
        N :: AluXOff+22, AluYOff+2*DataN;
        Z :: AluXOff+23, AluYOff+2*DataN;
        as0 :: AluXOff+25, AluYOff+2*AddrN;
        as1 :: AluXOff+25, AluYOff-1;
        ds :: AluXOff, AluYOff-1;
        shen :: AluXOff+1, AluYOff-1;
        xor :: AluXOff+20, AluYOff-4;
        and :: AluXOff+21, AluYOff-3;
        or :: AluXOff+22, AluYOff-2;
        pcs :: AluXOff+23, AluYOff-4;

        FOR i := 0 .. DataN-1 DO IR.i :: 1, 2*i+1 END;
        iren :: 1, 2*DataN;
        LS :: 3, 2*DataN-3;
        ph0' :: 3, 2*DataN-2;
        ph1 :: 3, 2*DataN-1;
        ph2 :: 3, 2*DataN;
    END
END Wotan.
```

F.3 Layout Synthesis

As can be seen from the Lola code, most of the gates are placed manually. This is necessary to achieve a dense and fast layout. Using the automatic placer of Hades without placement hints, one ALU slice has a bounding box of 20x12 cells with a utilization of 18%. The design does not fit into a XC6216. After manually optimizing the ALU slice, it has a bounding box of 24x2 with a utilization of 88%. The layout is shown in Figure F.7. The remaining logic is preplaced as well, to ensure a routable design. Floor-planning is essential with this design, as control signals, such as the shift control signal shen, need to be in the correct column. To get a correct, routable layout using the placer and router interactively, about 12 hours were needed. This includes the time to understand the design.

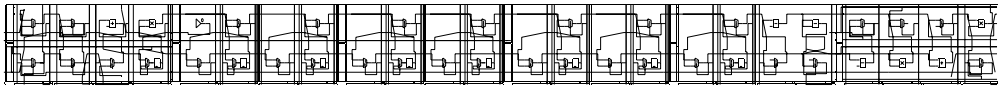

Figure F.7: ALU Slice

The quick response from the Hades tools were a prerequisite to try out different placements of the ALU slice, to see whether the resulting layout was routable or not. Table F.3 compares the performance of the Hades tools with the XACT step Series 6000 software from Xilinx. As already shown in Chapter 6, Hades is an order of magnitude faster.

First, we let XACT place the design automatically. This took 5 minutes and resulted in a

layout with components left unplaced. Then, we placed the design by inserting hints into the Lola code. Note that this process still took much longer than when using Hades, because some gates were left unplaced. The router took about 40 times as much time and resulted in twice as many unrouted nets, despite utilizing the Magic resources. We tried the automatic ripup and reroute feature of XACT. After 10 iterations, which took nearly 3 hours, there still were 52 unrouted nets. This experience undermines our request for a fast, interactive and iterative design cycle.

1247 Cells and 3931 Nets	Hades	XACT high
Compile	0.5 s	-
Map & Place	0.5 s	160 s
Bounding Box	34 x 54	34 x 54
Route	23.1 s	1076 s
Unroutes	85	168
Route using Script	41.2 s	-
Unroutes	0	-
10x Ripup & Reroute	-	9900 s
Unroutes	-	52
Total (M+P+R)	41.7 s	1236 s / 10060 s
Speedup of Hades		29.6 / 241.2

Table F.3: Wotan Place & Route Times

Clearly, the requirement of our tools to specify the location of nearly every cell is an undesirable feature, but it is necessary to achieve a compact, routable and fast design. The sophisticated placement algorithm of XACT fails to find a satisfactory solution and the user has to give hints as well.

Wotan has a critical path of 163 ns and should therefore run at about 6 MHz. Since most instructions have two phases (except for load and store), it executes about 3 MIPS. The resulting design is shown in Figure F.8. Note that the size of the bounding box in Table F.3 does not include the address drivers on the right side of the chip. With these, the bounding box would be 64x54 cells.

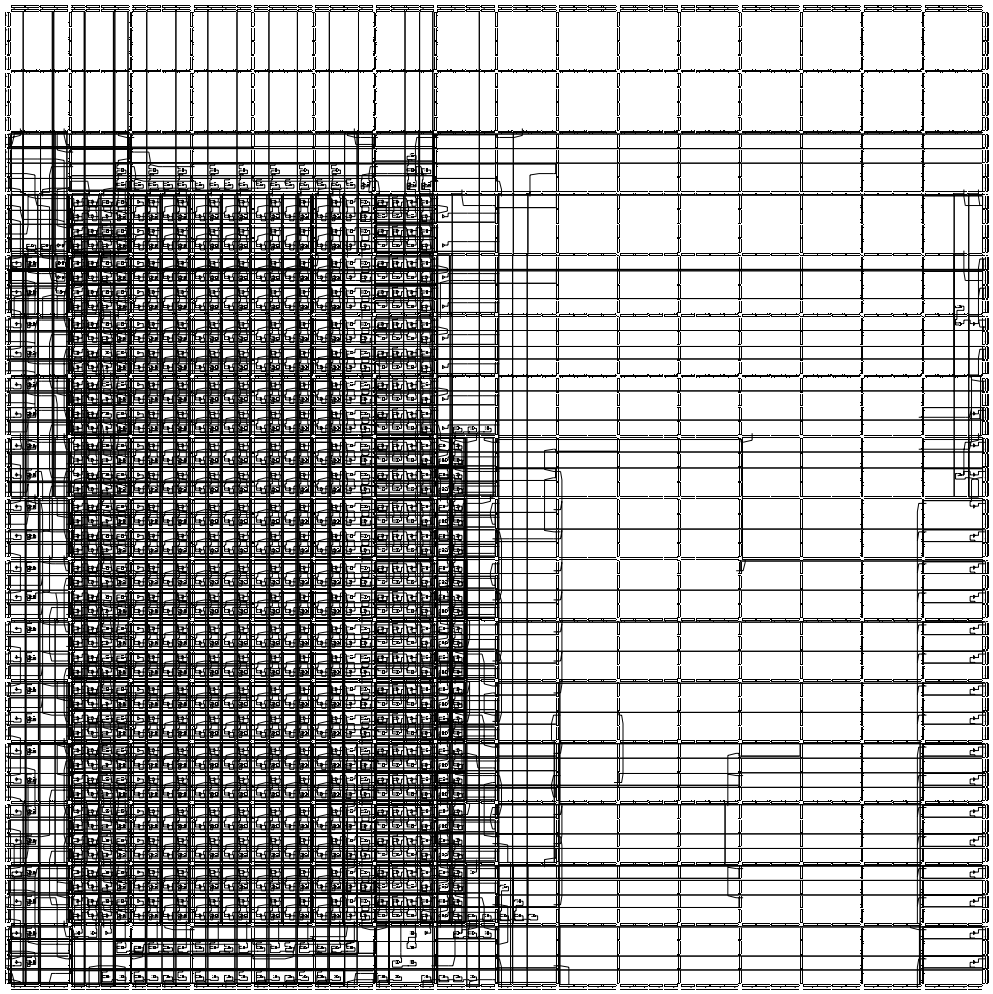

Figure F.8: Wotan Microprocessor on XC6216 FPGA

G Resources on the Web

This section lists a few useful URLs pointing to pages with information about Trianus and Hades, as well as RC boards. These URLs are believed to be current. However, since the World Wide Web is an ever-changing environment, we also list AltaVista queries (http://-altavista.digital.com) below the URLs, which should return the most recent pointer to these pages. More URLs can also be found in the Bibliography.

http://www.cs.inf.ethz.ch/cs/group/wirth/projects/cad-tools/
+hades +lola +trianus
: Page of the Institute for Computer Systems, ETH Zürich, on CAD tools for hardware design. From this page, the Trianus/Hades system can be downloaded.

http://www.inf.ethz.ch/publications/diss.html
+host:inf.ethz.ch +dissertations
: Page of the Department of Computer Science, ETH Zürich, listing available dissertations.

http://www.eee.bham.ac.uk/James-RoxbyP/reconfig.htm
+james-roxby +birmingham +reconfigurable
: Philip James-Roxby's page about reconfigurable computing, containing Lola examples, screen-shots of Hades in action and more pointers to other resources.

http://www.vcc.com
+VCC +6200
: Page of Virtual Computer Corporation offering a PCI-card featuring an XC6200 FPGA.

http://www.xilinx.com
+xilinx +6200 +product +literature
: Page with data-sheet for the XC6200.

Bibliography

[Act95] Actel. *ACT Family Field-Programmable Gate Array Data Book*, 1995.

[AMD95] Advanced Micro Devices. *Programmable Logic Data Book*, 1995.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

[ACG95] M. Alexander, J. Cohoon, J. Ganley, G. Robins. Performance-Oriented Placement and Routing for Field-Programmable Gate Arrays. *Proc. European Design Automation Conference.* IEEE Computer Society Press, 1995.

[Alg90] Algotronix. *CAL-1024 Data Sheet*, 1990.

[Alg91] Algotronix, *Configurable Array Logic User Manual*, 1991.

[Alt96] Altera. *Data Book*, 1996.

[ACC95] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, G. Snider. Teramac — Configurable Custom Computing. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1995.

[ACC96] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, G. Snider. Plasma: An FPGA for Million Gate Systems.*Proc. Intl. Symposium on Field Programmable Gate Arrays.* ACM, 1996.

[ABD92] J. M. Arnold, D. A. Buell, E. G. Davis. Splash 2. *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[AS93] P. M. Athanas, H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, Vol. 26, No. 3, March, 1993.

[Atm95] Atmel. *Configurable Logic: Design & Application Book*, 1995.

[BRV89] P. Bertin, D. Roncin, J. Vuillemin. Introduction to Programmable Active Memories. *Systolic Array Processors.* Prentice Hall, 1989.

[Ber93] P. Bertin. *Mémoires actives programmables: conception, réalisation et programmation.* (Programmable Active Memories: Conception, Realization and Programming.) Dissertation, Paris University, 1993.

[BT94] P. Bertin, H. Touati. PAM Programming Environments: Practice and Experience. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1994.

[Bli96] T. Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis.* Dissertation 11894, ETH Zürich, 1996.

[BM77] R. S. Boyer, J. S. Moore. A Fast String-Searching Algorithm.*Communications of the ACM*, Vol. 20, No. 10, October 1977.

[BRA96] Berkeley Reconfigurable Architectures, Systems and Software Research Group. *BRASS Research Group Homepage*. http://www.cs.berkeley.edu/projects/brass-/index.html, 1996.

[Bre96] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[BG95] G. Brebner and J. Gray. Use of Reconfigurability in Variable-Length Code Detection at Video Rates. *Proc. 5th Intl. Workshop on Field-Programmable Logic and Applications.* Springer, 1995.

[Bre77] M. A. Breuer. Min-Cut Placement. *Journal of Design Automation and Fault Tolerant Computing*, Vol. 1, 4 (Oct.), 343-362, 1977.

[BFR92] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publishers, 1992.

[BKV96] S. Brown, M. Khellah, Z. Vranesic. Minimizing FPGA Interconnect Delays. *IEEE Design & Test of Computers*, Vol. 13 (4), 1996.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, Vol. C-35, 6 (Aug.), 677-691, 1986.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, Vol. 24, 293-318, 1992.

[Buc96] I. Buchanan. Xilinx Development Corporation, Scotland. Personal Communication, 1996.

[BJL92] M. Burrows, C. Jerian, B. Lampson, T. Mann. *On-Line Compression in a Log-Structured File System.* Digital Systems Research Center Report No. 85, 1992.

[Cas96] S. Casselman. Virtual Computer Corporation, USA. Personal Communication, 1996.

[Chr95] Chromatic Research, *MPact Media Engine*, 1995.

[CKW95] S. Churcher, T. Kean, B. Wilkie. The XC6200 FastMap^{TM} Processor Interface. *Proc. 5th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 975, Springer, 1995.

[CH96] D. A. Clark, B. L. Hutchings. The DISC Programming Environment. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[Con96] D. Conroy. Digital Systems Research Center, California. Personal Communication, 1996.

[Coo71] S. Cook. The Complexity of Theorem Proving Procedures. *Proc. Third Annual ACM Symposium on the Theory of Computing*, 1971.

[CLR90] T. H. Cormen, C. E. Leiserson, R. L. Rivest.*Introduction to Algorithms.* The MIT Press, 1990.

[COO93] W. B. Culbertson, T. Osame, Y. Otsuru, J. B. Schackleford, M. Tanaka. The HP Tsutsuji Logic Synthesis System.*Hewlett-Packard Journal, August*, 1993.

[CAC96] W. B. Culbertson, R. Amerson, R. J. Carter, P. Kuekes, G. Snider. Exploring Architectures for Volume Visualization on the Teramac Custom Computer.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[Cyp95] Cypress. *Programmable Logic Data Book*, 1995.

[DeH96] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing.* Dissertation, A.I. Technical Report No. 1586, Artificial Intelligence Laboratory, MIT, 1996.

[DGR87] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang. Technology Mapping in MIS. *Proc. IEEE Conference on Computer Aided Design*, 1987.

[Dio87] J. Dion. Fast Printed Circuit Board Routing. *Proc. 24th Design Automation Conference*, ACM/IEEE, 1987. Also available as Digital Western Research Laboratory Report No. 88/1, 1988.

[DM95] J. Dion, L. M. Monier. *Contour: A Tile-Based Gridless Router*. Digital Western Research Laboratory Report No. 95/3, 1995.

[DR97] A. R. Disteli, P. Reali. Combining Oberon with Active Objects. *Proc. Joint Modular Languages Conference*, 1997.

[ECF96] C. Ebeling, D. C. Cronquist, P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[Ebe87] H. Eberle. *Development and Analysis of a Workstation Computer.* Dissertation 8431, ETH Zürich, 1987.

[EH94] J. G. Eldredge, B. L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1994.

[FM82] C. M. Fiduccia, R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. *Proc. 19th Design Automation Conference*, ACM/IEEE, 1982.

[FB72] R. A. Finkel, J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1), 1974.

[Fie95] C. A. Fields. The Proper Use of Hierarchy in HDL-Based High Density FPGA Design. *Proc. 5th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 975, Springer, 1995.

[Fou93] P. W. Foulk. Data-Folding in SRAM Configurable FPGAs. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1993.

[FRV91] R. J. Francis, J. Rose, Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Tabled-Based FPGAs. *Proc. 28th Design Automation Conference*, ACM/IEEE, 1991.

[Gal95] J. Galloway. The Transmogrifier-C Hardware Description Language and Compiler for FPGAs. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1995.

[GJ79] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Fremann, 1979.

[Geh97] S. Gehring. *An Integrated Framework for Structured Circuit Design with Field-Programmable Gate Arrays.* Dissertation to appear, ETH Zürich, 1997.

[GLW94] S. Gehring, S. Ludwig, N. Wirth. A Laboratory for a Digital Design Course Using FPGAs. *Proc. 4th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 849, Springer, 1994.

[GL96] S. Gehring, S. Ludwig. The Trianus System and its Application to Custom Computing.*Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[GHK90] M. Gokhale, W. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, P. Olsen. SPLASH: A Reconfigurable Linear Logic Array. *International Conference on Parallel Processing*, 1990.

[Gol89] D. E. Goldberg. *Genetic Algorithms.* Addison-Wesley, 1989.

[GS95] M. Gschwind, V. Salapura. A VHDL Design Methodology for FPGAs.*Proc. 5th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 975, Springer, 1995.

[Guc94] S. Guccione. *List of FPGA-based Computing Machines*. http://www.io.com-/˜guccione/HW-list.html, 1994.

[GG95] S. Guccione and M. Gonzalez. Classification and Performance of Reconfigurable Architectures. *5th Intl. Workshop on Field-Programmable Logic and Applications.* Springer, 1995.

[Guc95] S. Guccione. *Programming Fine-Grained Reconfigurable Architectures.* Dissertation, University of Texas at Austin, 1995.

[Gut97] J. Gutknecht. Do the Fish Really Need Remote Control? *Proc. Joint Modular Languages Conference*, 1997.

[GMN96] B. Gunther, G. Milne, L. Narasimhan. Assessing Document Relevance with Run-Time Reconfigurable Machines.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[HH95] J. D. Hadley, B. L. Hutchings. Design Methodologies for Partially Reconfigured Systems.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1995.

[HWA76] M. Hanan, P. K. Wolff Sr., B. J. Agule. A Study of Placement Techniques, *Journal of Design Automation and Fault Tolerant Computing*, Vol. 1, 1 (Oct.), 28-61, 1976.

[HHC96] I. Harvey, P. Husbands, D. Cliff, A. Thompson, N. Jakobi. Evolutionary Robotics at Sussex. *Proc. Intl. Symposium on Robotics and Manufacturing*, 1996.

[Hee88] B. Heeb. *Design of the Processor-Board for the Ceres-2 Workstation*. Technical Report Nr. 93, Dept. Informatik, ETH Zürich, 1988.

[Hee93] B. Heeb. Debora: *A System for the Development of Field Programmable Hardware and its Application to a Reconfigurable Computer.* Dissertation, ETH Zürich, 1993.

[HN91] B. Heeb and I. Noack. *Hardware Description of the Workstation Ceres-3*. Technical Report Nr. 168, Dept. Informatik, ETH Zürich, 1991.

[HP92] B. Heeb and C. Pfister. Chameleon: A Workstation of a Different Colour. *2nd Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 705, Springer, 1992.

[HP96] J. L. Hennessy, D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1996.

[HW96] J. P. Heron, R. F. Woods. Architectural Strategies for Implementing an Image Processing Algorithm on XC6000 FPGA. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[Hig69] D. Hightower. A Solution to Line Routing Problems on the Continuous Plane. *Proc. Design Automation Workshop*, 1969.

[Hof96] D. Hofmann.*Minterms: A Program for Logic Minimization*, 1996.

[IEE87] IEEE Standard 1076-1987, *IEEE Standard VHDL Language Reference Manual*, Institute for Electrical and Electronic Engineers, 1987.

[ICS96] Institute for Computer Systems. *The Oberon Archive.* ftp://ftp.inf.ethz.ch/pub-/software/Oberon.

[Ise96] C. Iseli. *Spyder: A Reconfigurable Processor Development System*. Dissertation 1476, EPF Lausanne, 1996.

[IS95] C. Iseli, E. Sanchez. A C++ Compiler for FPGA Custom Execution Units Synthesis. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1995.

[JG93] H. Johnson and M. Graham. *High-Speed Digital Design: A Handbook of Black Magic.* Prentice Hall, 1993.

[Kar86] R. M. Karp. Combinatorics, Complexity, and Randomness.*Communications of the ACM*, Vol. 29, No. 2, February 1986.

[Kea89] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. Thesis CST-62-89, Univ. of Edinburgh, 1989.

[KG89] T. A. Kean, J. Gray. Configurable Hardware: Two Case Studies of Micro-Grain Computation. *Systolic Array Processors*. Prentice Hall, 1989.

[Kea96] T. A. Kean. Xilinx Development Corporation, Scotland. Personal Communication, 1996.

[KB92] T. A. Kean, I. Buchanan. The Use of FPGA's in a Novel Computing Subsystem, *Proc. 1st Intl. ACM/SIGDA Workshop on FPGAs.* ACM Press, 1992.

[KNS96] T. Kean, B. New, B. Slous. A Fast Constant Coefficient Multiplier for the XC6200. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[KL70] B. Kernighan, S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, Vol. 49, February, 1970.

[Keu87] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. *Proc. 24th Design Automation Conference*, ACM/IEEE, 1987.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecci. Optimization by Simulated Annealing. *Science*, Vol. 220, May, 1983.

[Kni96] G. Knittel. A PCI-compatible FPGA-Coprocessor for 2D/3D Image Processing. *FPGAs for Custom Computing Machines* '96. IEEE Computer Society Press, 1996.

[Kri84] B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. *IEEE Trans. on Computers*, Vol C-33, No. 5, May 1984.

[Lat96] Lattice. *Lattice Data Book*, 1996.

[Lee61] C. Y. Lee. An Algorithm for Path Connections and its Applications. *IRE Trans. Electronic Computer*, Vol. EC-10, September 1961.

[LS88] B. Liskov, L. Shrira, Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *SIGPLAN '88 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices 23 (7), 1988.

[Log91] Logical Devices. CUPL PLD/FPGA Language Compiler, 1991.

[Lud94] S. Ludwig. *Conventions for Programming.* Internal Memo, Institute for Computer Systems, ETH Zürich, 1994. Also available in the Oberon System 3 distribution.

[Lud96] S. Ludwig. The Design of a Coprocessor Board Using Xilinx's XC6200 FPGA - An Experience Report. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[LSC96] W. Luk, N. Shirazi, P. Cheung. Modelling and Optimising Run-Time Reconfigurable Systems. *FPGAs for Custom Computing Machines '96.* IEEE Computer Society Press, 1996.

[LD93] P. Lysaght, J. Dunlop. Dynamic Reconfiguration of FPGAs. *More FPGAs: Proc. 1993 Intl. Workshop on Field-Programmable Logic and Applications*, 1993.

[MD95] L. M. Monier, J. Dion. Recursive Layout Generation. *Proc. 16th Conference on Advanced Research in VLSI.* IEEE Computer Society Press, 1995.

[MD96] E. Mirsky, A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[Moo59] E. F. Moore. Shortest Path Through a Maze. *Annals of the Computation Laboratory of Harvard University.* Harvard University Press, 1959.

[MW91] H. Mössenböck, N. Wirth. The Programming Language Oberon-2. *Structured Programming.* Vol. 12, No. 4, 1991.

[Mot95] Motorola. *Fast Static RAM. Databook*, 1995.

[Mul97] P. Müller. *Arithmetische Einheiten auf FPGAs.* (Arithmetic Units on FPGAs.) Term Project, Institute for Integrated Systems, ETH Zürich, 1997.

[NS88] National Semiconductor. *Series 32000 Microprocessors Databook*, 1988.

[OE95] E. Oertli, H. Eberle. Switcherland – An Interconnect for Workstations. *Proc. 21st EUROMICRO 95 Conference*. IEEE, 1995.

[Ohr84] R. Ohran. *Lilith: A Workstation Computer for Modula-2*. Dissertation 7646, ETH Zürich, 1984.

[PCI93] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.0*. PCI Special Interest Group, Portland, 1993.

[Pfi92] C. Pfister. *CALLAS: A Physical Design Framework for Configurable Array Logic*. Dissertation 9940, ETH Zürich, 1992.

[Phi95] Philips Semiconductors. *TriMedia Data Sheet*, 1995.

[PTS93] D. V. Pryor, M. R. Thistle, N. Shirazi. Text Searching on Splash 2.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Press, 1993.

[Qui94] QuickLogic. *Very High Speed FPGAs Data Book*, 1994.

[Raz94] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. Dissertation, Harvard University, 1994.

[RBS94] R. Razdan, K. Brace, M. D. Smith. PRISC Software Acceleration Techniques. *Proc. Intl. Conference on Computer Design*, 1994.

[RS94] R. Razdan, M. D. Smith. High-Performance Microarchitectures with Hardware-Programmable Functional Units. *Proc. 27th Annual IEEE/ACM Intl. Symposium on Microarchitecture*, 1994.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula.* Addison-Wesley, 1992.

[RS97] P. Roe, C. A. Szyperski. Lightweight Parametric Polymorphism for Oberon. *Proc. Joint Modular Languages Conference*, 1997.

[Sha96] M. Shand. *PCI Pamette V1*. http://www.research.digital.com/SRC/pamette/, 1996.

[SKC95] G. Snider, P. Kuekes, W. B. Culbertson, R. J. Carter, A. S. Berger, R. Amerson. The Teramac Configurable Compute Engine. *Proc. 5th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 975, Springer, 1995.

[Syn92] Synopsys Inc. *VHDL Compiler Reference Manual*, 1992.

[TI87] Texas Instruments. *The TTL Data Book*, 1987.

[Tho96] A. Thompson. An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics. *Proc. First Intl. Conference on Evolvable Systems: from Biology to Hardware*, LNCS, Springer, 1996.

[Tra95] D. W. Trainor. *An Architectural Synthesis Tool for VLSI Signal Processing Chips.* Dissertation, Queen's University of Belfast, 1995.

[TWM95] D. W. Trainor, R. F. Woods, J. V. McCanny. Architectural Synthesis of an Image Processing Algorithm Using Iris. *Proc. IEEE Workshop on VLSI Signal Processing*, 1995.

[TW96] D. W. Trainor, R. F. Woods. Architectural Synthesis and Efficient Circuit Implementation for Field Programmable Gate Arrays. *Proc. 6th Intl. Workshop on Field-Programmable Logic and Applications.* LNCS 1142, Springer, 1996.

[TM91] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[VSC96] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, B. Mangione-Smith. Configurable Computing Solutions for Automatic Target Recognition.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[VCC97] Virtual Computer Corp. http://www.vcc.com, 1997

[VBR96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age.*IEEE Trans. on VLSI Systems*, Vol. 4, No. 1, March 1996.

[WAL93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh. PRISM-II Compiler and Architecture.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1993.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[Wir95] N. Wirth. *Digital Circuit Design. An Introductory Textbook.* Springer, 1995.

[Wir96a] N. Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[Wir96b] N. Wirth. The Language Lola, FPGAs, and PLDs in Teaching Digital Circuit Design. *Proc. 2nd Intl. Andrei Eshov Memorial Conference.* LNCS 1181, Springer, 1996.

[WH95] M. J. Wirthlin, B. L. Hutchings. A Dynamic Instruction Set Computer.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1995.

[Woo96a] R. Woods. Answer to question asked during presentation of [WCG96], 1996.

[Woo96b] R. Woods. Queen's University of Belfast, Northern Ireland. Personal Communication, 1996.

[WCG96] R. Woods, A. Cassidy, J. Gray. VLSI Architectures for Field Programmable Gate Arrays: A Case Study.*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1996.

[WLH97] R. Woods, S. Ludwig, J. Heron, D. Trainor, S. Gehring. FPGA Synthesis on the XC6200 Using IRIS and Trianus/Hades (or from Heaven to Hell and Back Again).*Proc. IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, 1997.

[Xil96] Xilinx. *The Programmable Logic Data Book*, September 1996.

[ZK97] R. Zimmermann, H. Käslin, Cell-Based Multilevel Carry-Increment Adders with Minimal AT- and PT-Products.*To be published in IEEE Trans. on VLSI Systems*.

[ZR95] Zuken-Redac. *CadStar for Windows*, 1995.

Curriculum Vitae

Stefan Hans-Melchior Ludwig

May, 21 1966	born in Zürich, Switzerland, citizen of Schiers, Graubünden, son of Donat Dietegen Ludwig and Marlene Ludwig-Märki
1985	Matura Typus B, Kantonsschule Freudenberg, Zürich
1991	Diploma in Computer Science, Swiss Federal Institute of Technology, Zurich (ETH Zürich)
1991–1997	Research and teaching assistant, in the research group of Prof. Dr. N. Wirth, Institute for Computer Systems, Swiss Federal Institute of Technology, Zurich (ETH Zürich)
1997–	Member of research staff, Systems Research Center, Digital Equipment Corporation, Palo Alto, California