FPGA Prototyping of a RISC Processor Core for Embedded Applications

Michael Gschwind, *Senior Member, IEEE*, Valentina Salapura, and Dietmar Maurer

***Abstract*—Application-specific processors offer an attractive option in the design of embedded systems by providing high performance for a specific application domain. In this work, we describe the use of a reconfigurable processor core based on an RISC architecture as starting point for application-specific processor design. By using a common base instruction set, development cost can be reduced and design space exploration is focused on the application-specific aspects of performance.**

An important aspect of deploying any new architecture is verification which usually requires lengthy software simulation of a design model. We show how hardware emulation based on programmable logic can be integrated into the hardware/software codesign flow. While previously hardware emulation required massive investment in design effort and special purpose emulators, an emulation approach based on high-density field-programmable gate array (FPGA) devices now make hardware emulation practical and cost effective for embedded processor designs.

To reduce development cost and avoid duplication of design effort, FPGA prototypes and ASIC implementations are derived from a common source. We show how to perform targeted optimizations to fully exploit the capabilities of the target technology while maintaining a common source base.

***Index Terms*—Application-specific processors, embedded systems, field-programmable gate array (FPGA) prototyping, hardware/software codesign, processor design, processor emulation, reconfigurable processor core, target-specific HDL optimization for FPGAs.**

I. Introduction

In OUR work, we have designed an MIPS RISC processor [1] core as a starting point for hardware/software codesign space exploration [2]. The choice of this architecture was based on a number of factors, such as the provision of a clean starting point for application specific extensions and the architecture's popularity in the embedded control market. By choosing an existing target architecture, we are able to leverage a number of system tools, such as assemblers, compilers, and simulators, with little or no adaptation.

We have used this processor core in a number of hardware/software codesign experiments to explore the design space and generate optimized implementations for specific application requirements [3]. Our experiments have included a number of application areas, such as logic programming [4], [2], vector processing [5], [2], and fuzzy processing [6], [7]. All of these processor variations are based on the original MIPS-I architecture, adding only a limited amount of functionality to achieve high performance for a given problem set.

The processor core which serves as a starting point for these experiments has been described in the VHDL hardware description language (HDL). The design uses a modular design structure with control logic implemented as a set of communicating state machines.

As event-based or cycle-level simulation becomes increasingly inadequate to verify a significant execution trace for a given problem, this software-based simulation approach may not allow all aspects of a design's functionality to be exercised. To solve this problem, we consider prototype-based emulation. Rapid prototyping substitutes real time hardware emulation for slow simulator-based execution. By using this emulation technique, the time to market can be reduced because testing of the full system can be started early in the design cycle.

Previous work [8], [9] has shown how to prototype high-end processors on complex programmable logic emulation systems, but required a significant investment in resources, design adaptation, and designer time. While this may be appropriate for top-of-the-line processors for which such emulation has generally been employed, it may not be available for embedded applications. By reducing the logic capacity requirements for hardware emulation and using high density field-programmable gate array (FPGA) devices, such strategies can be made accessible for the development of embedded processors as well.

To facilitate such FPGA prototype implementations, we have investigated how to describe the processor architecture to achieve good synthesis results for both ASIC and FPGA implementations while maintaining a common source base for both target architectures. To do so we have continued our previous research about synthesis for FPGA targets [10] with the focus on restricting source code modification for efficient FPGA synthesis to only a few target-specific modules.

Such optimizations allow the implementation of a medium complexity CPU in a single high-density field-programmable gate array device, significantly reducing the investment in emulation hardware and support tools.

The purpose of such an FPGA implementation of the processor core is not to use such implementations as actual reconfigurable computational platforms which have been discussed in the literature [11]–[14]. Instead, prototyping is integrated in the hardware/software codesign flow and serves the sole purpose of architecture validation.

This paper is organized as follows: we discuss the development of the RISC core and its usage for generating application-specific processors in Section II and describe its architec-

Manuscript received October 29, 1997; revised October 16, 1998, December 13, 1998, and April 26, 1999. This work was performed while the authors were with Technische Universität Wien.

M. Gschwind and V. Salapura are with IBM, T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (mkg@us.ibm.com).

D. Maurer is with Technische Universität Wien, Vienna, Austria.

Publisher Item Identifier S 1063-8210(01)00700-4.

ture in more detail in Section III. Section IV describes how the processor core has been used to generate application-specific processor architectures, and Section V discusses the verification of the processor description. The architecture of field-programmable gate arrays and synthesis issues for FPGA implementation of the processor core are described in Section VI. Section VII analyzes the impact of the HDL coding style on the resource utilization of target FPGAs. Section VIII shows how special purpose FPGA logic can be efficiently accessed from an HDL description, and the usage of on-chip RAM capabilities is discussed in Section IX. We discuss synthesis results in Section X, and draw our conclusions in Section XI.

II. A RISC Core for ASP Development

We have developed a processor core for hardware/software codesign experiments based on the MIPS RISC architecture to generate optimized application-specific processors (ASPs) for embedded applications. The MIPS-I instruction set architecture was chosen as a starting point because it offers a simple, efficient execution engine for embedded applications. The processor core has been described using a synthesizable subset of the VHDL hardware description language and was synthesized using logic synthesis.

A key design requirement for the processor core was a modular structure to facilitate experiments with a number of extensions for different application areas. Source code modification should be restricted to those modules whose functionality is enhanced. Typically, instruction set extension involves modification of the instruction decode and execution stage modules. More complex instructions may affect other modules as well. An example of more complex behavior is a memory prefetching scheme which will be discussed in more detail in Section IV.

The modular structure provides a flexible framework with well-defined interfaces when changes to the processor architecture are to be made. A monolithic design would make design experiments difficult [15]. The reconfigurable processor core provides mechanisms to add or modify functionality without a major redesign effort. This is achieved with a modular design style with well-defined interfaces between modules and the use of a flexible and extendible pipeline control design.

To simplify reconfiguration of the processor, we describe it at a high abstraction level (detailed behavioral/register transfer level). Logic synthesis then maps high-level operators to module generators (such as Synopsys DesignWare [16]), which can select different implementation styles, based on timing, area, and power consumption constraints for different core-based ASPs. In contrast, choosing one particular implementation style using structural VHDL at the source level would prevent optimization to the requirements of a specific ASP target.

Since the HDL description is at a high abstraction level, it does not imply the use of a particular target technology. Modules used in the description can then be mapped to either an ASIC target for final chip production or to an FPGA target for rapid prototyping. This allows a common HDL source base for both implementations, reducing the risk of design errors and avoiding duplication of design effort.

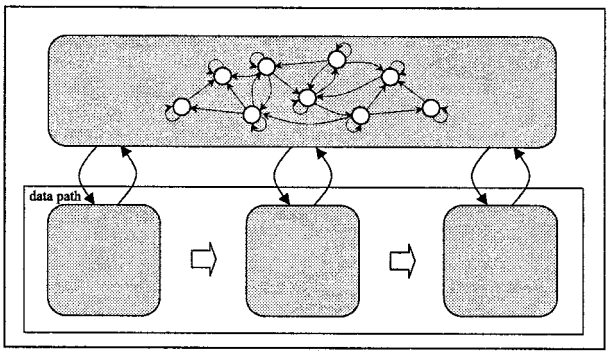

Fig. 1. Control unit in a conventional processor design: the data path and the control unit are separate blocks. A single, monolithic control unit controls the entire data path.

Using a high abstraction level also facilitates design exploration, as different design choices can be described and evaluated without major redesign of unrelated processor components and does not require a significant investment in the detailed implementation of each design point. Thus, this approach offers previously unachieved freedom in design space exploration which we have already applied to several problems. The ability to prototype these designs rapidly using FPGAs allows the evaluation of significant problem sets in real time.

Using the processor core in our hardware/software codesign experiments, we have developed a number of specialized processors for various areas, such as logic programming [2], vector processing [5], and fuzzy processing [6]. All these specialized processors are generated from the original MIPS-I architecture, adding only a limited amount of functionality to achieve high performance for a given problem set.

III. The Pipeline Control of the Processor Core

In many processors, the control unit is centralized and controls all CPU functions, introduces pipeline stalls, quashes bubbles, handles exceptions, etc. However, especially for pipelined architectures, this control unit is one of the most complex parts of the design, even for processors with fixed functionality. When the architecture is designed to be extendible, designing such a state machine is not feasible without some partitioning.

In this design, we have decided to break up the unstructured control unit into small, manageable units. Instead of a centralized control unit, which aims to control the complete data path (see Fig. 1), the control unit is integrated with the pipelined data path. Thus, each pipeline stage is controlled by its own, simple control unit (see Fig. 2). In this control design, each distributed state machine corresponds to exactly one pipeline stage, and this stage is controlled exclusively by its corresponding state machine. Overall flow control of the processor is implemented by cooperation of the control units in each stage based on communicating state machines.

Each pipeline stage is connected to its immediate neighbors and indicates whether it is able to supply or accept a new instruction. Communication with adjacent pipeline stages is performed using two asynchronous signals, `ready` and `accept`. When a stage has finished processing, it asserts the `ready` signal to indicate that data is available to the next pipeline stage (see Fig. 3).

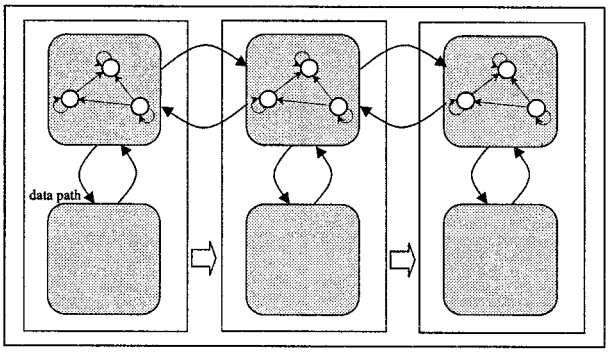

Fig. 2. Decentralized control unit: the control unit has been partitioned to reflect the different data path segments. The control units communicate using asynchronous handshake signals to implement the global control functions such as flow control.

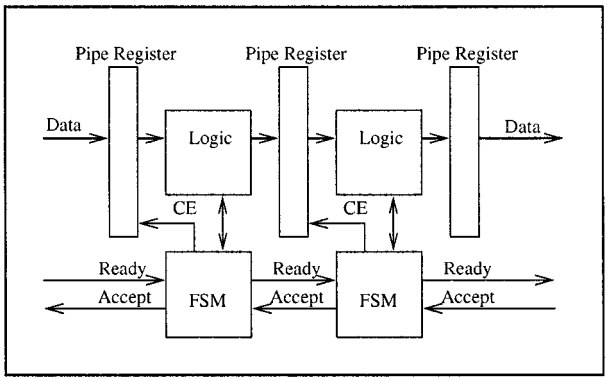

Fig. 3. Pipeline synchronization using cooperating state machines.

The next pipeline stage will then indicate whether it can accept these data by using the accept signal.

Since all data necessary to determine what actions are to be taken next are available in the pipeline stage (current operation status and synchronization signals from neighboring stages), computing the next state is simple. The state transitions of a single pipeline stage are shown in Fig. 4. Each pipeline stage can be in one of three basic states:

- *WORK*—The stage executes an instruction.
- *STALL*—The stage has computed a result but cannot pass it to the following stage.
- *IDLE*—The stage is idle, i.e., it does not compute any instruction, nor is it stalled.

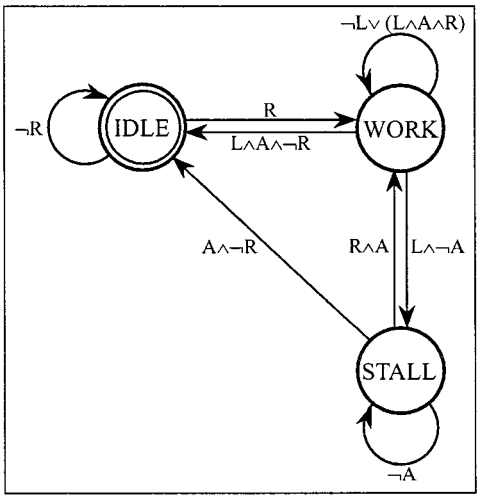

Fig. 4. FSM transition diagram for pipeline synchronization. The following conditions are evaluated: next pipeline stage can accept data (A), previous pipeline stage can supply data (R), and last cycle of computation (L).

This basic state machine is extended to cover the operational requirements of each stage, by dividing the WORK state into substates as needed. This is equivalent to using hierarchical substates for the WORK superstate in the Statecharts formalism [17]. An example is the implementation of the memory access stage, where the WORK state is refined into substates for cache access, main memory access, processor bus busy, cache update, and fixup.

Using these communicating state machines to control data flow in a pipeline partitions complex control logic into smaller modules which are easier to debug and extend.

Although this self-arbitrating approach has been proven to be a straightforward solution for data flow problems [18], it has to be adapted for CPU control. In a CPU, features such as branches, delay slots, forwarding and especially exception handling are more complex and require additional control for communication and synchronization between nonadjacent pipeline stages. For example for exception handling, we have implemented an exception controller which collects exception information from all pipeline stages. This controller is responsible for quashing the output of pipeline stages to maintain precise exceptions and initiate the fetching of the exception handler.

An issue with distributed pipeline management is the time stall information may take to propagate through multiple state machines in the pipeline in order to synchronize control flow. Since communication occurs only after all pipeline stages have completed execution, this may impact achievable cycle time. We have not found this to have a significant impact in the current implementation, but this may be an issue in some cases.

Communicating state machines are a modeling abstraction for the controller design, and the actual implementation does not have to implement the communicating state machines explicitly. Instead, CAD software can merge the multiple state machines and perform global optimization during synthesis. This optimization is supported by some CAD tools and is possible because all state machines operate with the same clock and reset signals.

IV. Design of Application-Specific Processors

We have used our processor core as the basis for designing several application-specific processors. The main issue in the design of such application-specific processors is the evaluation of the instruction set architecture.

To optimize processor performance for a particular application, a common approach is to extend the instruction set by application-specific instructions. Many instruction set extensions have been proposed, such as for signal processing or multimedia processing.

Adapting an instruction set to a particular problem is a difficult task, as many unknown issues have to be explored.

Due to the many factors involved in performance optimization, suggested optimization solutions often minimize only the number of instructions necessary to solve a problem, or at best the number of cycles.

We have used the MIPS processor core presented here to explore several application specific instruction set extensions using *hardware/software co-evaluation*. This approach is based on software evaluation to establish cycle count and synthesis of the ASP to establish cycle time and chip area. Software evaluation was performed using a cycle-accurate instruction set simulator described in the C programming language. For hardware evaluation, we performed logic synthesis to an ASIC technology for implementation information.

Using this co-evaluation approach, we have evaluated several application-specific instruction set extensions to implement a memory prefetching mechanism [5] and other performance enhancing extensions, including tag support for dynamically typed languages such as Prolog [2], and support for fuzzy calculation [6].

In the course of our design exploration for each of these problem areas, architecture extensions have ranged from adding only a few new operations in the execution unit to significant enrichment of the memory access mechanism. This shows the versatility afforded by this core-based ASP design approach—the processor core accommodates adaptation ranging from enriching the set of ALU operations to modifications of the branch and memory access logic.

Processor extensions for embedded applications are generated by adapting the processor core HDL description. The new functionality is added to the HDL modules affected, or for more complex extensions, new modules are described at a detailed behavioral/register-transfer level.

Examples of architecture extensions we have performed as a result of hardware/software co-evaluation of the design space for these application areas include the following.

fuzzy processing

To increase rule evaluation throughput, we have investigated support for subword parallelism at the fuzzy rule level, as well as other functions to enhance fuzzy processing, such as minimum, maximum and multiply-accumulate instructions. Implementing these instructions has required changes to the execution stage only [6].

logic programming

We have investigated some new ALU instructions to support stack management and garbage collection. In addition, we have added several instructions to perform multiway branches and table lookups based on tags in order to support dynamic type systems.

vector and list processing

To support vector and list processing, we have added support for variable-stride prefetching. This has required more significant additions to the processor core. Additional register files are used to store stride and prefetched memory values, and a prefetch unit performs the actual memory access. Changes to the processor pipeline have involved adding scoreboarding for the prefetched memory values in the instruction decode/register fetch stage, computing memory addresses for prefetch accesses in the execution stage, and issuing prefetch commands to the prefetch unit during the memory access stage [2].

V. Verification of the Processor Core

Verification of the processor implementation is performed at multiple abstraction levels by integrating several approaches. The aim is to validate the functionality as provided by the processor (including any application-specific instruction set extensions) and used by application programs, as well as the correct implementation in hardware. Integration of these verification steps is important to ensure that hardware implementation matches the design specification both in functionality and required performance level.

Design verification of a reconfigured RISC core is performed in four phases which are based on the same HDL description.

instruction selection

Candidate instructions are defined and their software impact is evaluated using a cycle-accurate instruction set simulator.

functional simulation

This phase is performed by simulating the register-transfer level HDL description. To simplify the verification process, we have developed a front end which graphically displays the pipeline state, the register file, and other information that may be necessary to verify the functionality of the processor.

rapid prototyping

FPGA prototypes are used to exercise a description after synthesis has been performed. Because the processor description can be exercised in actual hardware, extensive tests can be performed to verify functionality.

gate-level simulation

Gate level simulation is necessary to validate the actual timing and provide detailed information about the design.

A. Cycle-Level Simulation

Cycle-level simulation of the instruction set architecture is used to verify and evaluate application execution at the instruction level. We have used a cycle-level simulator based on SPIM [20] with a graphical user interface allowing users to view the pipeline state, memory, and register contents.

Different application specific instructions are evaluated by extending the cycle-level simulator to support this added functionality. This is performed by describing bit-accurately new instructions in the C programming language. The simulator modules are linked with these new modules generating a new simulator for the application-specific processor.[1] Once the new instructions have been included in the cycle-level simulator, we can evaluate software aspects of the embedded processor and the application performance even before the hardware is developed.

B. Functional Simulation

To simplify functional verification at the register transfer level, we have developed the graphical front end tool RISCview which displays the internal state of the processor model. This is performed by parsing traces generated by an HDL simulator

[1]For simple extensions, a common intermediate language could be used to automatically generate both the VHDL and C description.

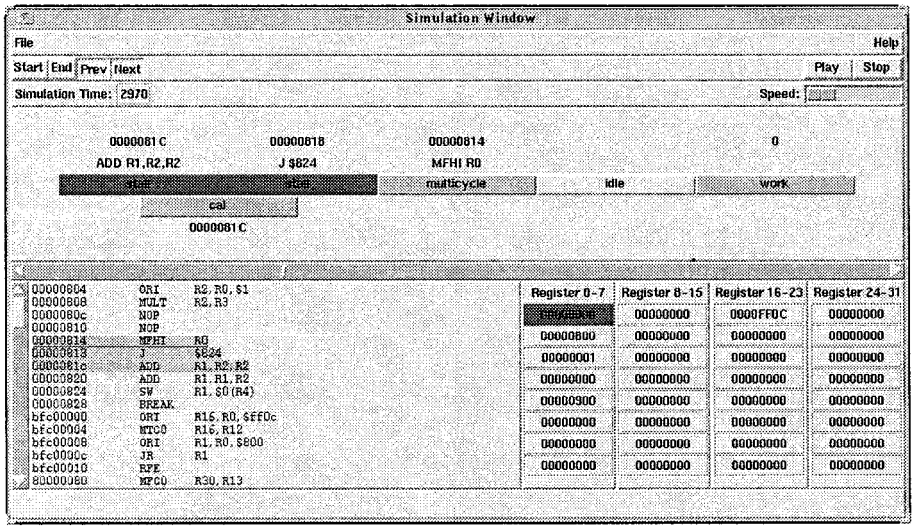

Fig. 5. Snapshot of the graphical validation tool: The pipeline displayed executes an instruction in the write-back stage, and the memory stage is idle because the execute stage operates on a multicycle operation. This operation stalls the instruction decode and address translation phases, while the instruction fetch is allowed to complete (the stage is in state CAL, i.e., cache access load). This view of the processor state was generated automatically by parsing waveform data supplied by the VHDL simulator.

to extract information about the CPU state. The state displayed includes: active machine instructions, the state of each pipeline stage, the register file, and other information necessary to verify the functionality of the processor.

This approach allows monitoring program execution as each instruction passes through the pipeline and identifies functional problems by tracing processor state changes. This method does not replace other verification techniques but reduces the amount of trace data which has to be scanned to identify problems in the design. The tool can be used with both the original HDL description and the post-synthesis net list, as the only data used are traces generated by the HDL simulator.

Most notably, the following functionality can be verified directly using a graphical display of traces:

- synchronization/communication between pipeline stages;
- state machines;
- basic computation and data flow;
- exception handling mechanism;
- processor state (general and special purpose registers).

Fig. 5 shows a snapshot of the graphical validation tool: the top button bars allow single stepping, tracing, and scrolling forward and backward through program execution. The main window displays the pipeline state graphically: each pipeline stage is associated with its current FSM state, the instruction executed in the pipeline, and the instruction address. A summary of the register file status and the program code are also given. Active instructions are highlighted in the program code window.

C. Rapid Prototyping

While simulation allows very accurate tracing of each individual signal in a design, simulation speeds are too slow to simulate significant programs. Thus, we have turned to developing an FPGA prototype in order to allow real time testing of the RISC processor (and proposed instruction set extensions) before actual fabrication. Prototyping of the processor core is described in detail in the next several sections of this paper.

In addition to higher simulation speeds, FPGA prototyping allows the verification of functionality which is otherwise difficult and time-consuming to verify if using simulation. A further advantage of FPGA-based hardware emulation is that actual peripheral components can be attached to the processor core. Not all processor designs are amenable to this type of verification, due to limitations in the complexity of available FPGA devices. Currently available field-programmable devices offer logic capacities of 100k gate equivalents, but this number is increasing rapidly. However, for embedded applications, the current complexity level of FPGAs is sufficient to emulate most designs.

D. Gate-Level Simulation

High-level verification based on RISCview is mainly used to verify functional correctness. Low-level problems such as timing violations cannot be verified directly using only RISCview although many problems can be uncovered because of their impact on program correctness.

To ensure correctness of low-level implementation details, interfaces and timing, we use gate-level simulation for ensuring compliance with design specifications. Gate-level simulation validates the actual timing and provides detailed information about the design. Gate-level simulation is based on established practice using a standard HDL simulator.

VI. FPGA Synthesis of the RISC Core

FPGAs consist of two resource types, a set of logic resources (CLBs) implementing the logic functions in a design and interconnect resources (SWMs) which combine logic resources to form more complex functions (see Fig. 6).

The complexity and structure of logic resources is one of the distinguishing features of different FPGA families. In the Xilinx XC4000 FPGA family, each CLB contains three

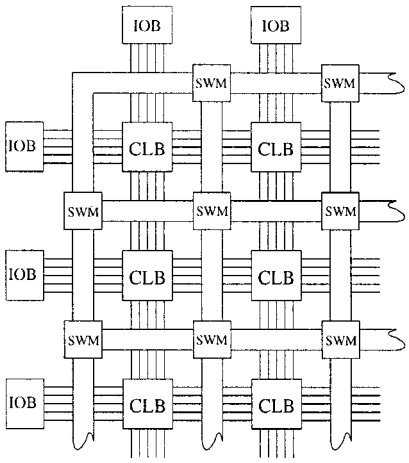

Fig. 6. The internal structure of the Xilinx XC4000 FPGA architecture: devices consist of configurable logic blocks (CLBs), interconnection resources (controlled by switching matrices SWM), and I/O blocks (IOB) for accessing pins.

user-programmable lookup tables, two flip–flops, and has nine inputs and two outputs. In addition, each CLB contains special purpose logic, such as fast carry logic, for the implementation of fast arithmetic. The look-up tables can also be used as RAM blocks.

The Xilinx XC4000 architecture provides a bit-oriented architecture, where each logic resource is programmed individually. This approach offers maximum versatility but incurs considerable cost for data path oriented applications, where control information is duplicated for each bit of the data paths.

Data path-oriented architectures (such as the Lucent ORCA architecture) allow each CLB to operate on multiple bits simultaneously. This approach allows sharing of lookup tables and simplifies control logic, thereby increasing device density. This is balanced by a higher cost for control logic, which is bit oriented and cannot take advantage of the multibit function cells. When targeting a design to data path oriented architectures, resource efficiency is a function of how well a description can be mapped to the multibit data path structure.

Using an HDL-based design flow, a prototype implementation can be obtained from the same HDL description as the final implementation. Logic synthesis is targeted at FPGAs to exercise the design for verification purposes. Final implementation is performed using logic synthesis to an ASIC technology.

To better estimate the impact of the description style on FPGA efficiency and to improve key design parts, we have explored logic synthesis for FPGAs. The focus of this exercise was not to completely rewrite the entire VHDL description (which would be an arduous endeavor and contradict the aim of using the same VHDL source for ASIC and FPGA technologies), but to optimize critical design parts either for speed or resource utilization.

The optimization of VHDL models for FPGAs can be grouped into two distinctive approaches as follows.

1) Coding Style:

The HDL coding style is adapted to the structure of the target technology. Because the hardware structure is directly inferred from the HDL description, the HDL coding style has significant impact on the type and number of resources required. Design components typically affected by coding style are sequential elements or finite state machines. Coding style also determines whether a description can be mapped onto multibit data path logic resources in data path oriented architectures to exploit available logic resources efficiently.

2) Usage of Special-Purpose Devices on FPGA:

Many FPGA devices include special-purpose circuitry to improve device effectiveness in some application area. Typical examples of such circuitry are dedicated arithmetic functions (such as fast carry logic), RAM functionality, or wide decoders. For HDL design to be effective, this special purpose logic has to be exploited wherever possible. In FPGAs, many of these features are accessed by using parameterizable megafunction cores, which are typically accessible as module generators to the synthesis tools.

Optimizing the VHDL description to exploit the strengths of the target technology is of paramount importance to achieve an efficient implementation. This is particularly true for FPGA targets, where a fixed amount of each resource is available and choosing the appropriate description style can have a high impact on the final resource efficiency [10]. For typical FPGA features, choosing the right implementation style can cause a difference in resource utilization of more than an order of magnitude.

Synthesis efficiency is influenced significantly by the match of resources implied by the HDL description and resources present in a particular FPGA architecture. When an HDL description implies resources not found in a given FPGA architecture, those elements have to be emulated using other resources at significant cost. Such emulation can be performed automatically by EDA tools in some cases but may require changes in the HDL description in the worst case, counteracting the aim of a common HDL source code base.

In this work, our experiments and the results presented are based on the Xilinx XC4000XL architecture [21]. The Xilinx XC4000XL architecture is a medium-grained FPGA architecture, with a good tradeoff between routability and logic capacity. We have used the Synopsys Design Compiler [22] as synthesis tool and vendor-supplied software for FPGA placement and routing.

VII. Coding Style for FPGAs

The initial structure of synthesized logic is directly inferred from the structure of the hardware description. Thus, the quality of the final hardware very much depends on the description style used at a higher level. To account for this, the high-level description has to be adapted to guide the synthesis tool to choose the appropriate implementation.

The coding style can be used to influence the choice of different implementation methods for functionality such as signal selection from multiple sources, the choice of sequential elements or the encoding of the state vector in finite state machines.

Signals are often selected in VHDL from a number of sources by such VHDL constructs as conditional statements and indexing of an array. From these source level statements, logic synthesis normally generates multiplexers to select an input. While this implementation is appropriate for ASIC processes, wide multiplexers can result in considerable resource usage in

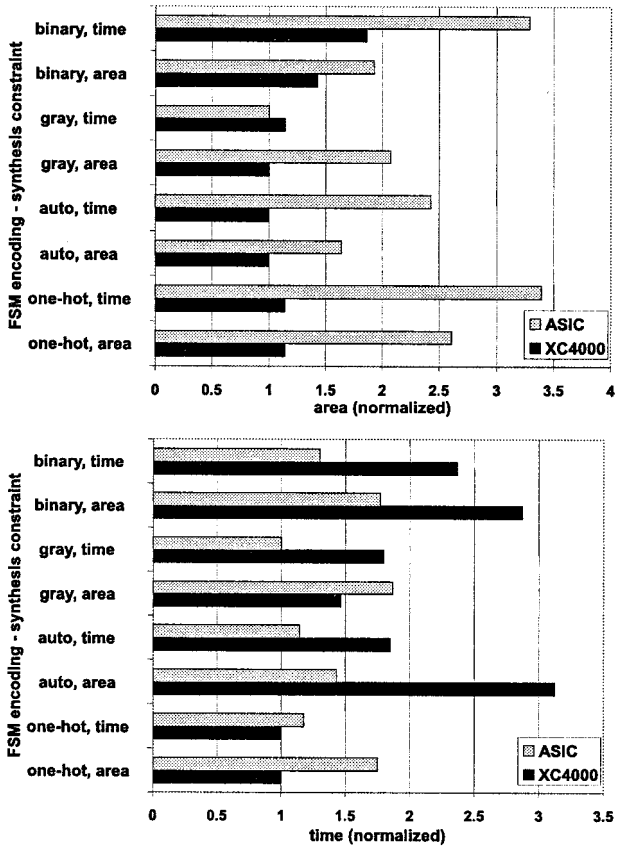

Fig. 7. Resource usage and timing for state machine synthesis using different encoding schemes.

FPGAs requiring many CLBs. For FPGAs supporting tristate resources, these expensive multiplexers can often be replaced by using tristate busses. This alternative implementation uses considerably less logic resources on an FPGA.

The design of state machines is also affected heavily by the coding style used to represent the state machine. By using different state vector encodings, the speed/area tradeoff is affected significantly for different target technologies. Advanced CAD tools allow the choice of a particular state machine encoding at synthesis time, facilitating the maintenance of a common HDL source base.

Fig. 7 compares the encoding tradeoffs for a simple finite state machine for the Xilinx XC4000 FPGA and an ASIC library using four encoding techniques: one-hot encoding, a solution adapted to the particular FSM (auto), gray code encoding, and binary encoding of states.

When ASICs are the target process, fully encoded representations such as binary or gray code encoding of states lead to space efficient designs, whereas the one-hot encoding scheme consumes more resources. This is different for FPGA devices with a large number of flip–flops, where a decoded representation reduces decoding logic at the cost of typically underutilized flip–flops.

Since the resource distribution of the one-hot encoded state machine maps well to the resource distribution of the target architecture, it is a very compact representation. Since decoding logic is more expensive on FPGAs, a decoded representation has a more pronounced performance advantage for such targets.

VIII. Special Purpose FPGA Logic

Many FPGA devices include special-purpose circuitry to improve device effectiveness in some application areas. For HDL design to be effective, this special purpose logic has to be exploited wherever possible to achieve maximum speed and resource efficiency.

A typical example of such special purpose circuitry is the provision of optimized arithmetic functions to simplify the implementation of fast adders, subtracters, counters, and other related function blocks. When such special purpose logic is available, an optimal implementation is based on the usage of such hardware. In general, no algorithmic advantages can be gained by substituting a superior description for such a function block, as the special purpose hardware is implemented at the hardware level ("dumb hardware is faster than smart software").

Thus, using high-level operators which map to module generators optimized for the target platform offer superior results (and are easier to maintain). Using such module generators makes the HDL description independent of the target technology, as module generators can optimize each function for the target technology without requiring different HDL descriptions. Some synthesis systems can invoke these generators automatically when a module is inferred from the VHDL description while others require that these modules be inserted by the designer.

To compare different modeling styles and their implementation in hardware, we have described an adder module with several different algorithms:

- *rip* a ripple-carry adder;
- *srip* a structured ripple-carry adder built from 1-bit full adder modules;
- *cla* a carry look-ahead adder;
- "+" using the VHDL "+" operator linked to a module generator.

To compare these design styles, we have compiled these adders with and without target-specific module generators. Resource usage and timing of these implementations of an eight bit adder are shown in Fig. 8.

The synthesized circuit based on the target-specific module generator is significantly smaller and more efficient. Even when using generic target-independent module generators, the "+" operator was slightly better than an open-coded version because the synthesis tool contains a powerful library of module generators. Implementations not based on the fast arithmetic functionality of the FPGA have similar resource usage which is much higher than the technology-specific version, and their performance is slower by a factor of 2 to 3.5.

IX. Storage Structures

Sequential elements are normally provided in FPGAs either in the form of flip–flops or latches. Depending on the nature of sequential elements, HDL descriptions should choose to use the provided elements to avoid the penalty of simulating one sequential element with another.

While memory structures can be built from these basic sequential elements and appropriate addressing logic implemented as combinatorial logic, such implementations are

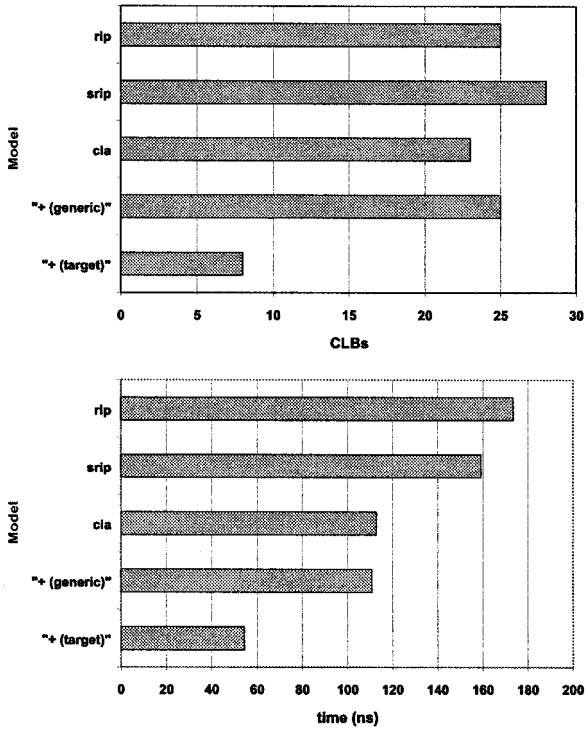

Fig. 8. Resource usage and timing of different implementations for an eight bit adder.

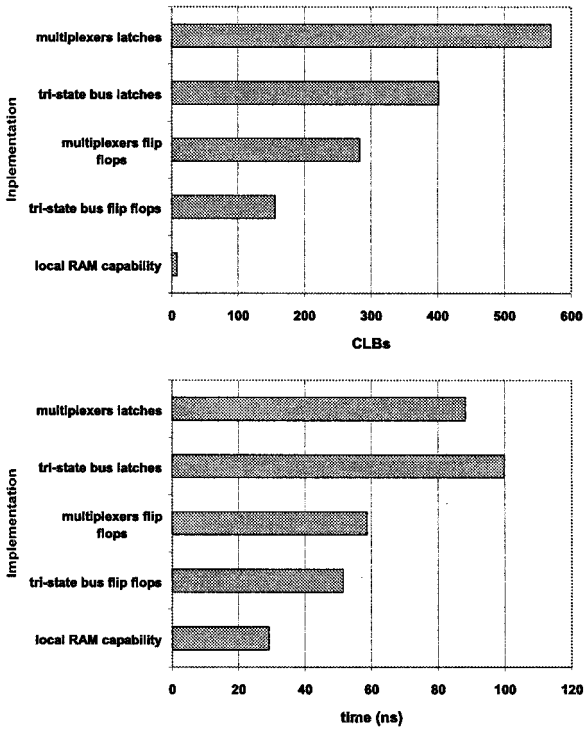

Fig. 9. Resource usage and timing of a 16×16 register file using five different VHDL coding strategies for implementing RAM structures.

highly inefficient. To efficiently implement local memory or register files in FPGAs, some FPGA families allow logic blocks to be configured as RAM blocks. Larger RAM blocks can be built by combining several of these blocks. For implementing large RAM blocks, external SRAM is a better solution.

Fig. 9 compares five different FPGA implementations for a 16×16 RAM. Depending on the VHDL description, either flip–flops or latches are generated as storage elements using either a MUX-based signal selection scheme or tri-state buses. The first four architectures show how coding style in VHDL can affect resource consumption yielding a four-fold improvement for implementing a register file.

A fifth, alternative design is based on the usage of the embedded RAM supported by the FPGA device. This implementation exploits decoder logic already present in the lookup tables of the CLBs and thus incurs no delay due to building a decoder with multiple levels of configurable logic blocks. To use the on-chip RAM provided by FPGAs, VHDL descriptions can include RAM components from the target FPGA's library. These modules map directly to the features of the FPGA family. Isolating such target-specific descriptions in a distinct VHDL entity will enhance portability. By using the on-chip RAM functionality, the final design uses only 1.4% of the logic resources compared to the original design.

Using an implementation based on the embedded RAM available in the target FPGAs (the Xilinx XC4000XL series), we have implemented a 32×32 bit general purpose register file with two read ports and a single write port found in the MIPS-I architecture. The original register file implementation based on a latch array required over 1000 CLBs, i.e., two thirds of the available logic capacity of an XC4044XL device. Using the on-chip RAM capability, the final synthesis result of the optimized register file contains only 69 CLBs (64 CLBs for the register file and an additional 5 CLBs for steering logic).

X. Results and Discussion

Table I summarizes the synthesis results of our MIPS-I processor core for the FPGA XC4000 family targets and for the Matra Harris (MHS) 0.6μm ASIC technology. The complexity of each processor module is given as the number of CLBs for the XC4000 FPGAs and as MHS cell count for the ASIC technology. For the MHS ASIC technology, the register file was not synthesized using Synopsys, as regular memory structures cannot be efficiently generated using logic synthesis.

Fig.10 shows a floorplan of the placed and routed design. The modules are superimposed upon the floorplan. Boundaries between modules are gradual, reflecting the global place and route decisions made by the placement software. (No constraints were set on the placement of individual modules.)

Fig. 11 shows the achievable throughput which can be obtained from this design on different execution platforms. For the hardware platforms, we show the processor frequency. For comparison purposes, the VHDL simulator performance has been converted to an artificial "frequency rating" by dividing the simulator throughput by a cycle count of 1 CPI. This chart shows the benefits which can be derived from direct hardware execution using a prototype, when compared to processor simulation. The data used for this comparison are based on event-driven functional VHDL simulation. Postsynthesis gate-level simulation is significantly slower than the figures reported here.

With the advent of the complex high-density devices from a number of vendors, single-chip emulation becomes a realistic verification choice. While in the past, FPGA prototyping

TABLE I
OVERVIEW OF THE RISC CORE BASED ON THE MIPS-I ARCHITECTURE: THIS TABLE GIVES THE MODULE COMPLEXITY AS CLB COUNT WHEN SYNTHESIZED FOR THE XILINX XC4000XL FAMILY AND AS CELL COUNT FOR THE MATRA HARRIS 0.6μm PROCESS

Description	Module	CLBs	MHS
AT pipeline stage (PC, TLB access)	AT	56	1377
instruction fetch unit	IF	57	1160
instruction decode unit	ID	153	2472
execution unit (ALU)	EXE	332	4796
integer multiply/divide unit	MD	232	3367
data memory access	MEM	191	3169
register file writeback	WB	19	380
register file	RF	69	N/A
coprocessor 0 (exception handling, TLB)	CP0	198	3705
cache controller	CCON	25	496
glue logic	-	129	66
RISC core	R3K	1461	20988

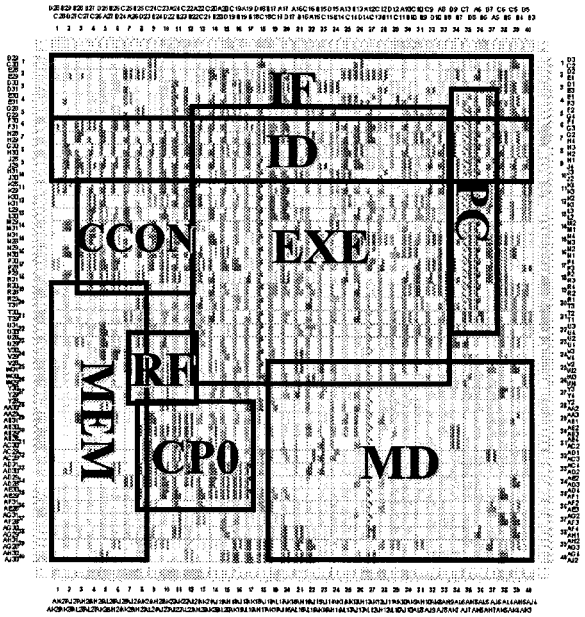

Fig. 10. Floorplan of the placed and routed processor core: the pipeline stages have been placed from the top right to the lower left corner of the FPGA. The modules show considerable overlap as logic is placed according to interconnect requirements.

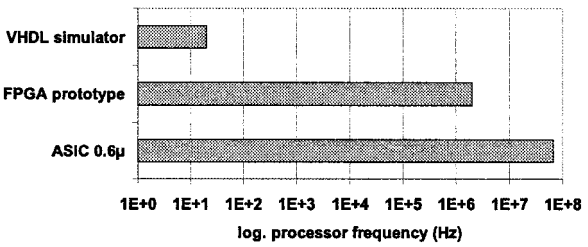

Fig. 11. Instruction throughput is a function of achievable processor frequency for hardware implementations of the MIPS-I model. Simulation speeds have been converted to a nominal frequency rating to facilitate comparison.

required design partitioning across multiple FPGAs and specialized emulation platforms, this is no longer necessary. This results in a simpler design flow and a higher effectiveness of FPGA emulation, since partitioned FPGA designs are pin-constrained, resulting in underutilization of the logic resources.

By optimizing key design parts to exploit the features of the target technology, VHDL designs can be retargeted for the design of a prototype implementation with only a few carefully selected changes to the design to exploit the specific features of the target architecture. These changes are localized and restricted to only a few places of the description, thus permitting to generate both ASIC and FPGA implementations from the same design description base.

Integrating FPGA prototyping into the design flow of a reconfigurable RISC processor core exploits the respective strengths of both FPGA and ASIC implementations. FPGA prototyping offers fast turn-around time and enables several design choices to be evaluated in a short time, which would be impossible using ASIC implementations due to long turn-around time and high cost. For a final implementation, ASICs offer superior performance and lower cost. With a common HDL source base, both FPGA and ASIC implementations can be used to leverage the respective strengths of both technologies without duplicating design effort.

XI. CONCLUSION

In this work, we have introduced a synthesizable MIPS RISC processor core. The design of this RISC core is based on communicating state machines and serves as the starting point for the development of application-specific processors. The core has been designed to be reconfigurable in order to support hardware/software co-evaluation of instruction sets for design space exploration.

Design space exploration is particularly important for achieving good cost/performance tradeoffs in embedded systems. Since embedded systems often require excellent performance on a particular application area, processors for such systems stand to profit the most from hardware/software codesign techniques. Using an extendible RISC core, a framework is available which can be extended for the actual application requirements without incurring excessive nonrecurring engineering costs during the evaluation cycle.

With the arrival of high-density FPGA devices, prototyping has become accessible to designers of embedded applications built around application specific processors. While in the past, specialized emulation hardware consisting of a large number

of FPGAs has been required to emulate microprocessors, high-density FPGAs facilitate effective and affordable real time testing of application-specific processor variants.

We have shown what factors contribute to efficient FPGA implementation. This has been demonstrated by targeting a processor core based on the MIPS-I instruction set architecture at a single, high-density FPGA. The processor prototype fits on a single FPGA device, thereby obviating the need to perform multichip partitioning which results in a loss of resource efficiency.

Through the use of design abstraction, it is possible to maintain a common HDL source base for both FPGA and ASIC implementations, so that final ASIC implementation information can be combined with FPGA real-time evaluation without duplicating design effort. This design flow relies on CAD software to choose optimal implementations for both target technologies. Only a few, clearly identified modules such as register files need to be specially optimized at the source level to achieve efficient resource usage.

ACKNOWLEDGMENT

The authors wish to thank the reviewers whose suggestions have helped to improve the quality of this work. The authors thank C. Mautner for his help with the final partitioning and placement. The Synopsys and Xilinx XACT tools have been made available to them through the EUROPRACTICE program of the European Commission.

REFERENCES

[1] G. Kane and J. Heinrich, *MIPS RISC Architecture: Reference for the R2000, R3000, R6000 and the New R4000 Instruction Set Computer Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[2] M. Gschwind, "Hardware/software co-evaluation of instruction sets," Ph.D. dissertation, Technische Universität Wien, Vienna, Austria, July 1996.

[3] ——, "Instruction set selection for ASIP design," in *Proc. 1999 Int. Workshop Hardware/Software Co-Design*, Rome, Italy, May 1999, pp. 7–11.

[4] A. Krall, "An extended Prolog instruction set for RISC processors," in *VLSI for Artificial Intelligence and Neural Networks*, J. G. Delgado-Frias and W. R. Moore, Eds. New York, NY: Plenum, 1991, pp. 101–108.

[5] M. Gschwind and T. Pietsch, "Vector prefetching," *ACM Computer Architecture News*, vol. 23, no. 5, pp. 1–7, Dec. 1995.

[6] V. Salapura and M. Gschwind, "Hardware/software co-design of a fuzzy RISC processor," in *Proc. Design, Automation Test Europe Conf. DATE '98*. Paris, France: EDAA, IEEE Computer Society Press, Feb. 1998, pp. 875–882.

[7] V. Salapura, "A fuzzy RISC processor," *IEEE Trans. Fuzzy Syst.*, vol. 8, pp. 781–790, Dec. 2000.

[8] W.-Y. Koe, H. Nayak, N. Zaidi, and A. Barkatullah, "Pre-silicon validation of Pentium CPU," in *Hot Chips V—Symp. Rec.*, Palo Alto, CA, Aug. 1993. TC on Microprocessors and Microcomputers of the IEEE Computer Society.

[9] J. Gateley, M. Blatt, D. Chen, S. Cooke, P. Desai, M. Doreswamy, M. Elgood, G. Feierbach, T. Goldsbury, D. Greenley, R. Joshi, M. Khosraviani, R. Kwong, M. Motwani, C. Narasimhaiah, S. J. Nicolino Jr., T. Ozeki, G. Peterson, C. Salzmann, N. Shayesteh, J. Whitman, and P. Wong, "UltraSPARC-I emulation," in *Proc. 32nd Design Automation Conf.* San Francisco, CA: IEEE, June 1995.

[10] M. Gschwind and V. Salapura, "A VHDL design methodology for FPGAs," in *Field-Programmable Logic and Applications—5th Int. Workshop FPL '95*, W. Moore and W. Luk, Eds. Berlin, Germany: Springer-Verlag, 1995, vol. 975 of Lecture Notes in Computer Science, pp. 208–217.

[11] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. 1995 IEEE Workshop FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, Eds. Napa, CA: IEEE, Apr. 1995, pp. 99–107.

[12] M. J. Wirthlin, B. L. Hutchings, and K. L. Gilson, "The Nano processor: A low resource reconfigurable processor," in *Proc. 1994 IEEE Workshop FPGAs for Custom Computing Engines*, D. A. Buell and K. L. Pocek, Eds. Napa, CA: IEEE, Apr. 1994, pp. 23–30.

[13] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Comput.*, vol. 26, pp. 11–18, Mar. 1993.

[14] A. Wolfe and J. P. Shen, "Flexible processors: A promising application-specific processor design approach," in *Proc. 21st Annual Workshop Microprogramming Microarchitecture*: IEEE Press, 1988, pp. 30–39.

[15] M. Gschwind and D. Maurer, "An extendible MIPS-I processor in VHDL for hardware/software co-design," in *Proc. European Design Automation Conf. EURO-DAC '96 EURO-VHDL '96*. Geneva, Switzerland: GI, IEEE Computer Society Press, Sept. 1996, pp. 548–553.

[16] Synopsys, *DesignWare User Guide*. Mountain View, CA: Synopsys, Inc., Nov. 1996, (Version 1997.01).

[17] D. Harel, "Statecharts: A visual formalism for complex systems," *Science Computer Programming*, vol. 8, pp. 231–274, 1987.

[18] T. H. Meng and S. Malik, *Asynchronous Circuit Design for VLSI Signal Processing*: Kluwer, 1994.

[19] G. de Micheli, "Computer-aided hardware-software codesign," *IEEE Micro*, vol. 14, no. 4, pp. 10–16, Aug. 1994.

[20] J. R. Larus, "SPIM S20: A MIPS R2000 simulator," Tech. Rep. 966, Univ. Wisconsin-Madison, Madison, WI, Sept. 1990.

[21] Xilinx, *The Programmable Logic Data Book*. San Jose, CA: Xilinx, Inc., 1996.

[22] Synopsys, *Design Compiler Family Reference*. Mountain View, CA: Synopsys, Inc., Nov. 1996, (Version 1997.01).

Michael Gschwind (M'91–SM'00) received the Ph.D. and M.S. degrees in computer science from Technische Universität Wien, Vienna, Austria.

He is a Research Staff Member in the high-performance VLSI architectures group at IBM T. J. Watson Research Center. At IBM, he contributed to several generations of binary translation architectures exploiting instruction-level parallelism, the evaluation of future microarchitecture options for current architectures, and the definition of the "Cell" next-generation high-performance system architecture. Before joining IBM in 1997, he was a faculty member (Univ.-Ass.) with Technische Universität Wien. He is the author of over 50 papers and holds several patents on high-performance computer architecture. His research interests include compilers, computer architecture, hardware/software codesign, application-specific processors, and field-programmable gate arrays.

Dr. Gschwind is a Senior Member of the IEEE Computer Society, and member of Phi Kappa Phi and the Fulbright Alumni Association.

Valentina Salapura received the M.S. degrees in computer science and electrical engineering from the University of Zagreb, Croatia, and the Ph.D. degree in computer science from Technische Universität Wien, Vienna, Austria.

She is with the advanced projects group at IBM T. J. Watson Research Center. At IBM, she is working on the BlueGene PetaFlops system and on network processor architectures. She is on sabbatical leave from her position as Faculty Member at Technische Universität Wien. She is the author of over 50 papers. Her research interests include application-specific processors, network processors, computer architecture, hardware/software codesign, and field programmable gate arrays.

Dr. Salapura is a member of the ACM.

Dietmar Maurer received the M.S. (Dipl.-Ing.) degree in computer science from Technische Universität Wien, Vienna, Austria, in 1996.

He is a Research and Teaching Assistant at the Technische Universität Wien. His research interests are in VLSI design, logic synthesis, and hardware/software codesign.