

High-Level Modeling and FPGA Prototyping of Microprocessors

Joydeep Ray
Advanced Micro Devices, Inc
5204 East Ben White Blvd.
Austin, Texas 78741
joydeep.ray@amd.com

James C. Hoe
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
jhoe@ece.cmu.edu

ABSTRACT

Emerging high-level hardware description and synthesis technologies in conjunction with field-programmable gate arrays (FPGAs) have significantly lowered the threshold for hardware development. Opportunities exist to integrate these technologies into a tool for exploring and evaluating microarchitectural designs. This paper presents a case study in developing the synthesizable high-level model of a superscalar processor and producing a working prototype in FPGA. Using an experimental operation-centric hardware description language, we have created the synthesizable model of a superscalar speculative out-of-order core for the integer subset of SimpleScalar PISA. A prototype implementation is produced by synthesizing the high-level model for the Spyder FPGA prototyping board. In addition, we have modified the baseline processor model to create derivative processor designs that add newly proposed experimental mechanisms. The derivative models are useful both in testing the completeness and correctness of new mechanisms and in assessing the mechanisms' impact on implementation area and cycle time.

Categories and Subject Descriptors

C.1 [Computer Systems Organization]: Processor Architectures

General Terms

Design

Keywords

prototyping, evaluation, microprocessor, microarchitecture, operation-centric, FPGA

1. INTRODUCTION

High-level design tools and field-programmable gate arrays (FPGAs) significantly reduce the effort, cost and risk of hardware implementation. These technologies can be incorporated into a manageable and affordable prototyping framework—a VLSI-scale “breadboard”—for exploring and evaluating new microprocessor designs. This approach provides another middle ground between paper design and full-scale implementation efforts where an architect can quickly test preliminary ideas without committing undue resources. In current practices, this evaluation step is normally carried out using functional and performance “C” simulators (e.g., SimpleScalar [3]).

For the right questions, software microarchitecture simulators can produce as convincingly an answer as actually building hardware, but developing a prototype provides a working proof-of-concept and directly addresses questions in design complexity and implementability. However, in order for prototyping to be an effective technique, the cost, risk and effort required should not greatly exceed those needed by simulations. By combining high-level hardware description technologies and FPGAs, researchers can prototype preliminary ideas without risking a great setback if the trial fails. The ability to quickly iterate design refinements also allows for a more interactive design approach in finalizing a microarchitecture. As a supplement to simulation, an FPGA prototype is a powerful demonstration because it forces a designer to produce a *complete* and *precise* description of a design; this undertaking helps to uncover many issues glossed over by a software simulator.

In this paper, we present our experiences in developing the synthesizable high-level models of superscalar out-of-order processors and producing working FPGA prototype implementations. Using an operation-centric high-level hardware description language, one student in significantly less than one year has created an FPGA implementation of a 4-way superscalar speculative out-of-order core for the integer subset of SimpleScalar PISA. The high-level model of the processor is amenable to modifications that add or change microarchitecture features. The model is also extensible with performance counters for data gathering. To demonstrate the utility of this infrastructure, the baseline processor model is modified to include a newly proposed scheme for soft-error tolerance that calls for dynamic redundant executions of the instruction stream [17]. We show how a prototyped-based evaluation allows us to test the performance and correctness of a new microarchitecture proposal

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'03, February 23–25, 2003, Monterey, California, USA.
Copyright 2003 ACM 1-58113-651-X/03/0002 ...\$5.00.

and assess its impact on implementation metrics such as cycle time and area.

Following this introduction, the remainder of the paper is organized as follows. Section 2 discusses two current approaches to preliminary or exploratory microarchitecture design study. Section 3 describes our technologies for high-level description and FPGA prototyping. Section 4 describes our experience in developing the prototype implementations of two processors: a baseline superscalar processor and a derivative design with soft-error tolerance. Section 5 presents the results of the design evaluations. Section 6 summarizes the key points we learned from this work.

2. BACKGROUND AND RELATED WORK

In this section, we first discuss the use of software simulators to study processor microarchitectures. Next, we describe applications of FPGAs and hardware emulators in microprocessor development.

2.1 Software Simulators

A large number of proprietary and public simulators have been developed to help investigate microprocessor design issues. These simulators can be broadly grouped into functional versus performance simulators. A functional simulator provides a virtual implementation such that the outward functionality of a design is emulated. A performance simulator models the inner workings of a design, in only as much detail as necessary, to extract the desired quantitative measures of some dynamic behavior.

In the last few years, the SimpleScalar processor simulation package [3] has gained popularity in the microprocessor research community. This software package contains several different parameterizable performance and functional simulators plus the supporting infrastructures for compilation, execution and monitoring. Researchers can either use a stock simulator or, as in most cases, extend the simulator source code to model the new mechanism that the researcher is studying. Other popular simulation environments in recent years include SimOS (full system simulator) [18], RSim (parallel architecture simulator) [15], and SimICS (full system simulator) [4]. Together, these simulators have investigated topics ranging from high-level issues such as memory consistency model [8] to low-level issues such as power and energy [1].

The dramatic savings in time, effort and expense offered by simulations have enabled many more hardware ideas to be researched, but it has two main limitations. First, a software simulator cannot provide feedback about a new idea's impact on implementation (cycle time, area and power, etc.). Second, modeling errors are often introduced into a software simulator when important but subtle details are mistakenly simplified in an effort to save simulator development time or simulation run time. These "abstraction" errors lead to incorrect or unrealistic measurements, but they often go undetected if they do not break the simulator's functional correctness.

2.2 Hardware Emulators

FPGA-based prototypes are routinely used in ASIC developments today to allow rapid design iteration and to enable early development of the surrounding system. In the microprocessor arena, until recently, the capacity of FPGAs has been quite limited, and consequently FPGAs have only

been used to prototype relatively simple processor architectures [2, 5, 9, 11, 14, 16].

Large scale hardware emulation systems (e.g., Quickturn and Ikos) have been used to prototype commercial microprocessor designs in the developments of Intel Pentium, SUN UltraSPARC-I [7], AMD K5 [6] and K6. A design as complicated as a microprocessor must be partitioned into small chunks and distributed over a large number (hundreds to thousands) of interconnected FPGAs. In these industry applications of hardware emulation, the primary goal is to increase the speed of RTL design simulations of a few million to tens of million gates. Detailed RTL design representations are compiled for cycle-accurate functional and performance simulation. These costly big-iron hardware emulators currently face growing competition from greatly improved software RTL simulator performance and high-density FPGAs.

These examples of prototyped processors were synthesized from detailed schematics or RTL design descriptions. In these cases, although FPGA and hardware emulators eliminate the long turn-around time of a full IC design flow, the efforts required to develop the low-level synthesis model itself can be prohibitive as a preliminary design evaluation technique.

3. PROTOTYPING TECHNOLOGIES

In this section we describe our technologies for accelerating processor modeling and prototype implementation. We begin by describing the *operation-centric* hardware description framework for modeling.

3.1 High-level Description and Synthesis

We choose an experimental operation-centric language [10] over mainstream RTL-based languages (e.g. RTL subset of Verilog or VHDL) to deal with the complexity of microprocessor designs. The operation-centric language/abstraction allows us to describe a complex design more concisely and with less chance for bugs. In an operation-centric description, state elements are declared explicitly as in RTL, but the behavior of the system is decomposed into a collection of predicated (a.k.a. guarded) operations. (This is in contrast to RTL descriptions which decompose a system's behavior into a collection of per-state-element next-state logic.) An operation can control the next-state value of any number of registers (and other types of state elements), but an operation is only relevant for a given instant when its predicate condition is satisfied.

The entire effect of an operation is considered atomic, that is an operation "reads" the entire state of the system in one step, and if the operation is enabled, the operation updates the state in the same step. If several operations are enabled in a step, any one (but only one) of the operations can be selected to update the state in one step, and afterwards a new step begins with the updated new state. The atomic semantics of operations permits the designer to formulate each operation under the assumption that the rest of the system is not changing at the same time—the designer does not have to worry about race conditions with other potentially concurrent operations. This permits an unambiguous and apparently sequential description of the hardware behavior. An execution is interpreted as a sequential interleaving of operations such that each operation produces a new state that enables the next operation. This sequential conceptual interpretation, however, does not preclude an implementation

that in reality overlaps the execution of multiple operations for better performance

In this work, we use an experimental operation-centric hardware description language TRSpec and its compiler. The TRSpec language is an adaptation of Term Rewriting Systems (TRS) [12]. In Section 4.1.1, we present examples from our TRSpec description of a superscalar processor. Here, we give an overview of TRSpec with the help of a simple two-stage pipelined processor example. In TRSpec, the collective values of the processor state elements can be symbolically represented as a *term*. In this case, a two-stage pipelined processor can be represented by terms of the form $\text{Proc}(\text{pc}, \text{rf}, \text{bf}, \text{imem})$. The four fields in a processor term are *pc* the program counter, *rf* the register file (an array of integer values), *bf* the pipeline buffer (a FIFO of fetched instructions), and *imem* the instruction memory (an array of instructions). The operation of instruction fetching in the fetch stage can be described by the TRSpec rule:

Fetch Rule:

$$\begin{aligned} & \text{Proc}(\text{pc}, \text{rf}, \text{bf}, \text{imem}) \\ \rightarrow & \text{Proc}(\text{pc}+1, \text{rf}, \text{bf.enq}(\text{imem}[\text{pc}]), \text{imem}) \end{aligned}$$

The handling of an Add instruction in the execute stage is given by the rule:

Add Rule:

$$\begin{aligned} & \text{Proc}(\text{pc}, \text{rf}, \text{bf}, \text{imem}) \text{ where } \text{Add}(\text{rd}, \text{r1}, \text{r2}) = \text{bf.first}() \\ \rightarrow & \text{Proc}(\text{pc}, \text{rf}[\text{rd} := (\text{rf}[\text{r1}] + \text{rf}[\text{r2}])], \text{bf.deq}(), \text{imem}) \end{aligned}$$

Each TRSpec rule specifies an operation. The left-hand-side (LHS, i.e., left of the rewrite symbol \rightarrow) of a rule specifies a pattern and other predicate conditions that must be satisfied for the rule to apply; the right-hand-side (RHS) describes the state changes when a rule is applied. The Fetch Rule follows *pc* to fetch from consecutive instruction memory locations and enqueues the fetched instructions into *bf*. The operationally-isolated Fetch Rule is not concerned with what happens if a branch is taken or if the pipeline encounters an exception. The Add Rule, on the other hand, would process the next pending instruction in *bf* as long as it is an Add instruction. Next consider the two possible executions of the Bz (branch if zero) instruction,

Bz Taken Rule:

$$\begin{aligned} & \text{Proc}(\text{pc}, \text{rf}, \text{bf}, \text{imem}) \text{ if } (\text{rf}[\text{rc}] = 0) \\ & \quad \text{where } \text{Bz}(\text{rc}, \text{ra}) = \text{bf.first}() \\ \rightarrow & \text{Proc}(\text{rf}[\text{ra}], \text{rf}, \text{bf.clear}(), \text{imem}) \end{aligned}$$

Bz Not-Taken Rule:

$$\begin{aligned} & \text{Proc}(\text{pc}, \text{rf}, \text{bf}, \text{imem}) \text{ if } (\text{rf}[\text{rc}] \neq 0) \\ & \quad \text{where } \text{Bz}(\text{rc}, \text{ra}) = \text{bf.first}() \\ \rightarrow & \text{Proc}(\text{pc}, \text{rf}, \text{bf.deq}(), \text{imem}) \end{aligned}$$

Although the Fetch Rule and the Bz Taken Rule both affect *pc* and *bf*, the sequential semantics of rules allows the formulation of the Bz Taken Rule to ignore contentions with the Fetch Rule, making it easy to independently verify that the Bz Taken Rule correctly handles a taken branch instruction. The complete behavior of this processor can be specified by additional TRSpec rules corresponding to the remaining op-codes, but notice that the specifications of new operations do not affect the correctness of the already existing rules.

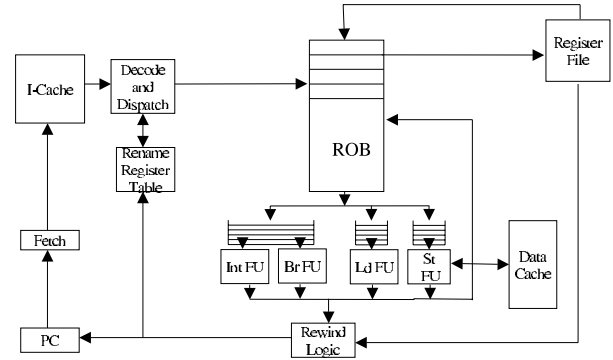


Figure 1: Baseline microarchitecture.

The convenient sequential and atomic interpretation of an operation-centric description taken by designers does not prevent a legal implementation from executing several operations in the same clock cycle. In fact, in the simple pipelined processor example above, the Fetch Rule and the execute stage rules must be executed concurrently for the implementation to be truly pipelined. In a correct implementation, the concurrent execution of multiple operations in the same clock cycle must still lead to a state that corresponds exactly to the result of some sequential and atomic execution of those same operations. To synthesize such an implementation, the TRSpec compiler analyzes dependence and interference relationships between all operations of a design to discover parallelism that can be safely exploited in hardware. The processed dependence information is encapsulated into an arbitrator that dynamically decides when to select multiple enabled but non-conflicting operations to execute in the same clock cycle¹. The current version of the TRSpec compiler compiles a TRSpec description to a synthesizable Verilog RTL description. We use the generated Verilog for both simulation and synthesis. For prototyping, the Verilog description is compiled using commercial RTL synthesis tools such as Xilinx ISE to target FPGAs. The Verilog description can also be synthesized against gate-array or standard-cell libraries to extract the area and timing of a hypothetical implementation in those technologies.

3.2 FPGA Prototyping Board

The availability of low-cost, high-density FPGAs have made rapid prototyping of complex processor microarchitectures possible. The current generation of FPGAs significantly out-price and out-perform the multi-million-dollar hardware emulators of just a few years back. The latest FPGAs cost on the order of several thousand dollars and can support million-gate logic designs (or upward of 8 million gate equivalents if 30% of the FPGA’s reconfigurable logic look-up tables are employed as SRAM storage). Commercial prototyping boards bundle FPGAs with memory and I/O interfaces to streamline accessibility and prototype development. A selection of commercial prototyping boards with differing capabilities and software support are available (see [20] for examples). Our prototypes are based on the Spyder PCI card, containing a Xilinx XCV2000E-6C FPGA and two 1-MByte SRAM banks [19]. The synthesized FPGA config-

¹Please refer to [10] for additional details on the synthesis and scheduling algorithms.

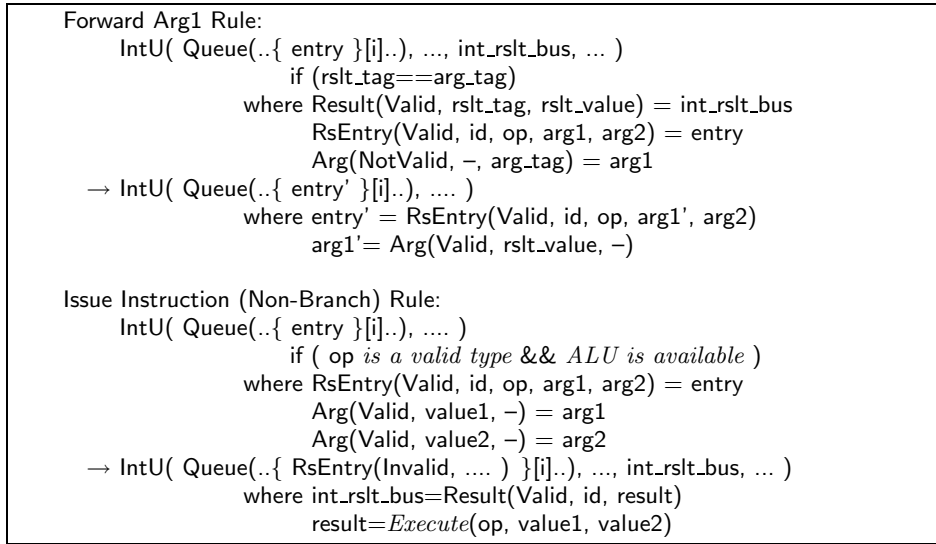


Figure 2: Examples of two TRSpec rules that describe the instruction issue operations of the reservation stations. The second rule is simplified for brevity.

uration bitstream can be downloaded to the FPGA through the PCI bus by the host computer’s processor. Software on the host processor can access user-defined registers in the FPGA and the contents of the SRAMs by memory-mapped I/O.

4. PROCESSOR MODELING AND PROTOTYPING

In this section, we describe our experience in developing prototype implementations of superscalar out-of-order processors on FPGAs. Using TRSpec, we have created the detailed model of a superscalar speculative out-of-order processor core with a small BTB (branch-target-buffer), I-cache and D-cache. To evaluate the effectiveness of reusing the model to derive other microarchitecture models, we have also developed a version of the soft-error-tolerant superscalar microarchitecture proposed in [17]. Here, we first describe the level of detail captured in our high-level processor models and then discuss the issues in modeling and prototyping.

4.1 Processor Models

4.1.1 Baseline

The baseline model is a 4-way superscalar speculative out-of-order processor that executes the integer subset of SimpleScalar’s PISA (portable instruction set architecture). With few exceptions, the processor model is specified in an operation-centric fashion using TRSpec. Only small sections of the arbitration and interface glue logic at the periphery of the processor are coded directly in Verilog as a top-level wrapper. The block diagram of the baseline microarchitecture is shown in Figure 1. Below we describe the salient characteristics of this superscalar core, including examples of TRSpec rules regarding the operations of the reservation stations (RS).

Instruction Fetch: Up to four instructions in the same cache line are fetched from the instruction cache in each

clock cycle. A 256-entry BTB, using 1-bit predictors, predicts the next fetch address. Unpredicted “absolute jump” instructions are caught and corrected in the first stage of the two-stage decode.

Decode and Dispatch: The baseline microarchitecture utilizes separate in-order architectural and out-of-order rename register files (where the renamed registers are logically associated with the reorder buffer (ROB)). A register alias table (RAT) helps rename the source operand registers. A source operand value, if ready, is fetched from either the renamed register file or the architectural register file. After renaming, instructions are dispatched to the appropriate functional unit’s RS, and, at the same time, they are also entered into the eight-entry ROB. The size of ROB and RSs can be changed by resetting their corresponding parameters in the TRSpec description. Up to four instructions (2 integer/branch, 1 load and 1 store) can be renamed and dispatched per cycle, and up to four instructions can retire from ROB in program order.

Issue and Execution: The microdataflow execution core consists of an integer unit, a branch unit, a load unit and a store unit. The integer unit and the branch unit share an eight-entry RS, while the load and store units each have their own four-entry RSs. The shared integer/branch RS can accept two instructions (any mix) per cycle and launch two instructions (1 integer/1 branch) per cycle. The load and store RSs each can accept one and launch one instruction per cycle. The load/store units have their own address generation logic. Stores are executed to memory in-order after it retires from ROB. After retirement, a store can continue to wait in the store buffer for its turn to access the cache. Loads and stores are never reordered, but loads can be reordered if there are no stores between them. The predicted branches are checked by the branch unit. Instruction refetch is triggered immediately after a branch misprediction, but decoding and renaming of new instructions are stalled until all committable instructions have retired from ROB.

Instructions pending in RSs are eligible for execution as soon as their data dependencies are satisfied. The data

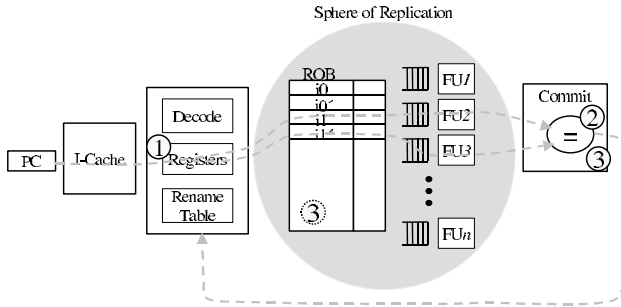


Figure 3: Soft error detection and recovery in a superscalar processor.

operands unknown at dispatch time are satisfied by forwarding. Figure 2 gives two TRSpec rules describing operations related to microdataflow instruction scheduling in the integer RS. The Forward Arg1 Rule describes how the result of a completing instruction is forwarded to the first operand of a data-dependent instruction in the integer RS. (There is a corresponding rule for forwarding to the 2nd operands.) In the rule’s LHS (left-hand-side), the terms “Queue(..{ entry }[i].)” and “int_rslt_bus” name the i ’th entry of the integer RS and the completion bus, respectively. The where clause of LHS restricts the rule to be applicable only when rslt_tag on int_rslt_bus matches the operand tag of a pending instruction in the i ’th entry of RS.² The RHS (right-hand-side) of the rule describes how the RS entry is updated if the LHS predicate conditions are satisfied. This rule is parameterized over the variable ‘ i ’ such that this one rule covers the forwarding operation of all integer RS entries. The Issue Instruction Rule in Figure 2 describes the operation of finding a readied integer instruction for execution.

Instruction and Data Caches: The operation-centric processor model includes a direct mapped on-chip I-cache for 512 64-bit PISA instructions in four-instruction-wide cache lines. The stated size of the I-cache is limited by the current FPGA capacity but can be adjusted easily as a parameter in the TRSpec model. The original model also includes a non-blocking D-cache that allows a load to bypass a load miss, but it is omitted from the FPGA-prototyped version due to capacity limitations.

Performance Counters: The fluidity of high-level models and quick prototype turn-around allow us to make interesting use of performance counters in debugging and performance evaluation. Counters to monitor specific events (elapse cycle, instruction retiring, I/D cache misses, branch mispredictions, etc.) can be added to the high-level model as required by experimentations.

4.1.2 Soft-Error Tolerance (SET)

Future microprocessors will become increasingly susceptible to transient hardware failures (or soft errors). A simple extension that provides protection against soft errors in a

²In this example, we use the convention where terms beginning with uppercase characters are symbolic constants while terms beginning with lowercase characters are variables. Variables are bound by pattern matching either in the main LHS pattern or in the where clauses. The “_” symbol signifies “don’t care”.

superscalar processor datapath is described in [17]. As depicted in Step (1) of Figure 3, instructions fetched from a single stream can be dispatched redundantly as two data-independent threads, using pre-existing register renaming capabilities. In Step 2, the redundantly computed results from the two threads are checked against each other at the commit stage for error detection. In Step (3), any inconsistency between the redundant results triggers the also pre-existing execution-rewind mechanism to recover the program’s execution from the failed instruction. The performance of the SET scheme has been studied using a software simulator. Here, we extend our baseline superscalar model with the proposed changes to verify the hardware feasibility of the modifications. The derivative model is subjected to the same prototyping and evaluation as the baseline processor to verify the hypothesis that the modifications only minimally impact the implementation’s area and cycle time.

4.2 Experiences and Issues in Modeling

The initial baseline processor model is completed over the course of a semester by one student. The complete description contains 128 rules in about 6000 lines of text. Since a rule could have multiple application sites (as in the examples in Figure 2), the 128 rules actually specify about 7000 distinct operations in the out-of-order processor, most of which are in RS and ROB. The operation-centric style descriptions are helpful in managing the complexity of the description by allowing us to focus on designing one rule at a time and keeping interactions between the numerous rules to a minimum. Starting from a stable baseline model, the modifications to add SET requires only one working day to incorporate and another day to debug. The SET extensions only involve modifying or adding a total of 21 rules in the TRSpec source files.

The use of high-level hardware description language in this study simplifies implementation efforts but by the same token introduces a potential source of inaccuracy in the prototypes. As described in Section 3.1, state transitions in an operation-centric description are not sequenced by a global clock; rather, at design time the execution is conceptually viewed as a sequence of *steps*. The final allocation of steps to clock cycles is controlled by the TRSpec compiler which tries to discover parallelism in the description and allows multiple parallelizable operations to take place in the same cycle. Whether the quality of the resulting prototype is representative of a best-effort implementation of the intended design depends on the ability of the compiler to make reasonable design decisions. This is an inherent property of any high-level design framework that reduces development efforts by off-loading low-level design decisions from the designer. This type of modeling inaccuracies, however, is different from microarchitecture-level correctness which remains under a high-level designer’s control. A high-level synthesis flow still demands a complete and unambiguous description in order to produce a correctly working prototype.

4.3 Prototyping Flow

Figure 4 illustrates our prototyping flow from an operation-centric model to a prototype executing on the Spyder board. The operation-centric descriptions are synthesized to Verilog RTL and then combined with the top-level Verilog wrapper. The wrapper implements the interfaces to memory and the local bus. The combined Verilog

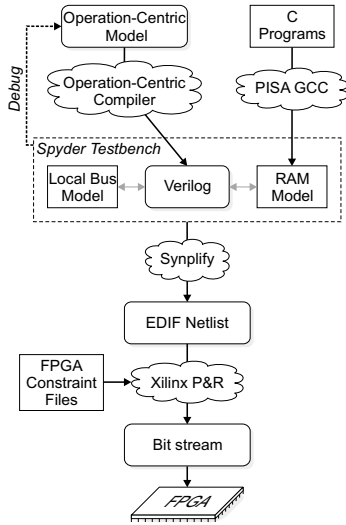


Figure 4: FPGA prototyping flow.

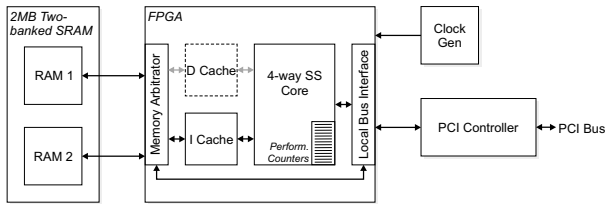


Figure 5: System architecture of a processor prototyped on Spyder.

descriptions are synthesized to create the FPGA configuration bitstream.

A Verilog simulation testbench based on a board-level model of the Spyder board provides a simulation and debugging environment. The testbench allows loading of PISA binaries (integer subset) into the simulated SRAMs and simulates the execution of the prototyped processor. For debugging, we use test programs compiled from C for general testing and hand-coded assembly segments for corner-case scenarios.

Next, the TRSpec generated Verilog descriptions are synthesized to an EDIF netlist using Synplify 7.0. In this step, Synplify automatically infers SRAM usage from Verilog’s two-dimensional array constructs. The synthesized netlist is then targeted for the Spyder board’s Xilinx XCV2000E FPGA using place-and-route tools from Xilinx ISE 4.1. Finally, the configuration bitstream is downloaded onto the FPGA using the Spyder device driver. Figure 5 illustrates the general architecture of a prototyped implementation and highlights the interfaces between the processor core and the rest of the system.

Currently, we are only executing small test programs. The size of executable benchmarks is limited by the 2-MByte SRAM “main memory” of the prototype system. Our goal is to expand the supporting infrastructure so we can execute complete benchmarks. The key challenge will be to support

system calls. For future work, we plan to use an FPGA card that plugs into the processor slot of a real system motherboard. Such a processor emulation environment can greatly enhance the usefulness of FPGA-based processor prototyping studies.

5. PROTOTYPE-BASED EVALUATIONS

In this section, we present implementation characteristics (area and cycle time) and architectural performance metrics (IPC and I-cache miss rate) for the prototype processor implementations. In addition, we use these results to evaluate the area and timing impact of the SET modifications. Implementation area and cycle time are generated by Xilinx ISE 4.1 tools on place-and-routed FPGA designs. The performance results are measured directly using the performance counters built into our models. The goal of this section is to motivate the additional types of information that could be extracted if prototyping becomes a viable option in microarchitecture studies.

5.1 FPGA Prototyping Capacity

Our initial baseline design with an on-chip D-cache cannot be place-and-routed for the Xilinx XCV2000E FPGA. A compromise is made to eliminate the on-chip D-cache such that the load and store units directly access the 3-cycle-latency external SRAMs on the Spyder board. This truncated design utilizes only 76% of the logic resources and can be place-and-routed in about 1 hour. This nominally 6-stage pipelined microarchitecture achieves a maximum frequency of 14.9 MHz. The critical path (61.17 ns) is attributed to the lengthy combinational path required for register renaming and operand fetch in the decode stage.

The capacity of the current generation of FPGAs are not yet completely adequate, but they are beginning to be able to support modestly complicated designs. One may argue that, due to approximately a factor of 10 loss in efficiency for reconfigurability, the modeling capacity of state-of-the-art FPGAs (currently equivalent to less than 10 million ASIC gates under the best assumptions) can never match the transistor count of state-of-the-art microprocessors (a few hundred million transistors), and thus any relevant processor design will never fit in a contemporary FPGA. This argument is not necessarily true. A great majority of modern processors’ transistor budget is expanded in caches. The second-generation Itanium processor (McKinley) spends only 25 out of the total 221 million transistors for logic [13]; the capacity needed by logic transistors is imminently reachable by a single FPGA within a couple of years. By mapping cache hierarchies to external memory devices, FPGAs can be a very capable prototyping platform for microarchitecture studies.

5.2 FPGA Synthesis Results

Table 1 summarizes the area and cycle time for four variations of the baseline superscalar design and the SET superscalar design. The four variations are 1. the baseline model (as described in Section 4.1.1); 2. a variation with a 16-entry ROB (instead of the 8-entry baseline); 3. a variation that only retires two instructions per cycle; 4. a variation which bypasses to allow the oldest instructions to retire in the same cycle it completes. All results are extracted from place-and-routed FPGA design files. The area is reported both in

Table 1: Area and cycle time of five FPGA prototypes.

Designs	Area (slice)	Area (gate)	Period (ns)
1. Baseline Superscalar	14768	425609	67.17
2. Baseline: 16-entry ROB	18255	—	—
3. Baseline: 2-way retire	12468	388357	62.91
4. Baseline: Bypassing	15346	432432	69.66
5. SET	15821	442173	69.74

Table 2: IPC and I-cache miss rate obtained using performance counters.

Program / HW	IPC	I-cache Miss/Try	Inst. Count	Cycle Count
Bubble Sort, n=100				
Baseline	0.53	7/94507	50751	95576
SET	0.35	8/143080	50751	143171
Fibonacci, n=10				
Baseline	0.48	10/5340	2618	5420
SET	0.34	13/7565	2681	7626

terms of slice³ (pre-place-and-route) and ASIC gate equivalents (post-place-and-route). The absolute values of these results are less important than the trends they exhibit. For example, it is determined that the SET extensions require only about 3.9% more area than the baseline superscalar, and it also only affects the cycle time by a small margin. The modifications increase the critical path in decoding and register renaming by about 3.8%. These results give empirical data to support that the proposed SET mechanism can be implemented on a superscalar datapath with only a small implementation overhead.

5.3 Microarchitectural Performance Simulations

The prototypes can also be used to execute benchmark programs directly to assess dynamic microarchitecture-level behaviors. Table 2 reports the IPC and I-cache miss rates of the baseline and the SET processors. The two toy benchmarks are a bubble sort on 100 random integers and a recursive Fibonacci number generator ($n = 10$). The programs are coded in C and compiled using SimpleScalar’s GCC PISA cross-compiler. The on-chip I-cache is initially empty for these measurements.

The processor models include performance counters that can be defined or reassigned, from run to run, to collect different statistics and to increase the observability of run-time hardware behaviors. Once a performance problem is uncovered by the performance counters, it can be diagnosed in detail in the Verilog testbench environment. For example, bubble sort’s low IPC is traced to a mismatch between its memory-intensive inner loop and the poor load/store performance of our prototyped processor designs.

³A slice is the basic building block of Virtex FPGAs. Each slice has 2 SRAM-based 4-to-1 lookup tables and 2 1-bit registers. XCV2000E contains over 19 thousand slices.

6. CONCLUSIONS

Ideally, an architectural investigation would begin with paper designs and simulations, but ultimately an idea should be tested in hardware. Unfortunately, the high up-front cost of hardware development limits the number of ideas that can be prototyped in VLSI. Granted, very good ideas will eventually be tried by the industry. But, if computer architecture research must rely on the industry for proof-of-concept, it would be very difficult to explore ideas that are outside of short-term development trends.

In this paper, we present our work in combining a high-level hardware design framework and FPGAs to develop prototype implementations of processor microarchitectures. We show that 1. interesting and non-trivial microarchitecture prototypes can be developed in a justifiable amount of time and effort and 2. prototyping can help provide answers that are not readily addressable through simulations. By bringing together the flexibility of a high-level design environment and FPGAs, we hope to achieve a processor prototyping flow that achieves that same flexibility and convenience as software development.

7. ACKNOWLEDGMENTS

This paper describes research done at the Computer Architecture Lab at Carnegie Mellon (CALCM) in the Department of Electrical and Computer Engineering. Funding for this work is provided in part by the Integrated Circuits and Systems Research program of the Semiconductor Research Corporation. We are thankful to Denali Inc. for access to their SRAM models.

8. REFERENCES

- [1] D. Brooks, V. Tiwari, and M. Martonosi. WATTCH: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, 2000.
- [2] R. Brown, J. Hayes, and T. Mudge. Rapid prototyping and evaluation of high-performance computers. In *Proceedings of the 1996 Conference on Experimental Research in Computer Systems*, pages 159–168, 1996.
- [3] D. Burger and T. M. Austin. The SimpleScalar tool set, Version 2. 0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Technical Report, 1997.
- [4] M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, P. S. Magnusson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.
- [5] J. Gaisler. *LEON/AMBA VHDL model description*. European Space Agency, November 2000.
- [6] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Proceedings of the 33rd annual Design automation conference*, pages 315–318, 1996.
- [7] J. Gateley, D. Greenley, M. Blatt, D. Chen, S. Cooke, P. Desai, M. Doreswamy, M. Elgood, G. Feierbach, and T. Goldsbury. UltraSPARC-I emulation. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 13–18, 1995.

- [8] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC+ILP=RC. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 162–171, 1996.
- [9] M. Gschwind, V. Salapura, and D. Maurer. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):241–250, April 2001.
- [10] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of International Conference on Computer Aided Design*, pages 511–518, 2000.
- [11] Y. G. Kim and T. G. Kim. A design and tool reuse methodology for rapid prototyping of application specific instruction set processors. In *Proceedings of the 1999 IEEE International Workshop on Rapid System Prototyping*, pages 46–51, 1999.
- [12] J. W. Klop. *Term Rewriting System*, volume 2 of *Handbook of Logic in Computer Science*. Oxford University Press, 1992.
- [13] S. D. Naffziger and G. Hammond. The implementation of the next-generation 64b Itanium microprocessor. In *Proceedings of 2002 IEEE International Solid-State Circuits Conference*, volume 2, pages 276–504, 2002.
- [14] K.-S. Oh, S.-Y. Yoon, and S.-I. Chae. Emulator environment based on an FPGA prototyping board. In *Proceedings of the 11th International Workshop on Rapid System Prototyping*, pages 72–77, 2000.
- [15] V. Pai, P. Ranganathan, and S. Adve. RSim: A simulator for shared-memory multiprocessor and uniprocessor systems that exploit ILP. In *Proceedings of the 3rd Workshop on Computer Architecture Education*, February 1997.
- [16] W. B. Puah, B. S. Suparjo, R. Wagiran, and R. Sidek. Rapid prototyping asynchronous processor. In *Proceedings of the 2000 IEEE International Conference on Semiconductor Electronics*, pages 223–227, 2001.
- [17] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 214–224, 2001.
- [18] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.
- [19] Spyder system. <http://www.x2e.de>.
- [20] Xilinx prototyping and evaluation boards. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=protoboards_protoboards_page.