In-System FPGA Prototyping of an Itanium Microarchitecture

Roland E. Wunderlich and James C. Hoe
Computer Architecture Laboratory at Carnegie Mellon
{rolandw, jhoe}@ece.cmu.edu

Abstract

We describe an effort to prototype an Itanium microarchitecture using an FPGA. The microarchitecture model is written in the Bluespec hardware description language (HDL) and supports a subset of the Itanium instruction set architecture. The microarchitecture model includes details such as multi-bundle instruction fetch, decode and issue; parallel pipelined execution units with scoreboarding and predicated bypassing; and multiple levels of cache hierarchies. The microarchitecture model is synthesized and prototyped on a special FPGA card that allows the processor model to interface directly to the memory bus of a host PC. This is an effort toward developing a flexible microprocessor prototyping framework for rapid design exploration.

1. Introduction

Until recently, only relatively simple microprocessor designs have been prototyped using FPGAs [5, 6, 7, 11, 12, 14]. This can be attributed to the limited capacity of older FPGAs and the high level of effort associated with conventional hardware design flows. In this paper, we describe an FPGA processor prototyping effort that leverages both high-level hardware design technologies and the growing capacity of new FPGAs. In this effort, we used the Bluespec HDL [2] to prototype a subset of the Merced Itanium microarchitecture. The prototyped microarchitecture is mapped onto a special FPGA hardware that permits the processor model to execute in a real PC system environment.

Our case study illustrates the costs and capabilities of hardware prototyping as compliments to existing microarchitecture design techniques. While simulation-based studies are effective tools, they are not capable of predicting all microarchitectural issues related to final physical design. We were able to evaluate relative circuit area and cycle time metrics for design alternatives using our implementation of a functioning Itanium processor. The processor model was developed with tractable design effort, and its primary components were calibrated for effective performance modeling.

We present the details of this work in the following sections. Section 2 describes the Bluespec HDL used for microarchitecture modeling. Section 3 presents the Itanium microarchitecture model in detail. Section 4 describes our FPGA prototyping platform and its operation. Finally, Section 5 presents an assessment of the prototyped processor model and the results from a microarchitecture design study. We offer our conclusions in Section 6.

2. Bluespec HDL

Bluespec is a synthesizable high-level HDL for rapid ASIC development [2]. A key advantage of Bluespec is the ability to describe hardware designs clearly and concisely, leading to fewer errors and faster design capture. Despite the language's use of high-level synthesis, Bluespec produces high quality output comparable with hand coded RTL [1]. Two important features of Bluespec are its operation-centric semantics and functional programming constructs.

2.1. Operation-centric semantics

In a Bluespec hardware description, state elements such as registers, arrays, and FIFOs, are declared explicitly. Unlike standard synchronous RTL languages, however, operation-centric state transitions are abstractly represented as a collection of atomic predicated actions known as "rules." Each Bluespec rule is comprised of a guarding predicate condition and a set of actions that update affected state elements. When a rule's predicate is satisfied, all of the rule's actions are carried out simultaneously and instantaneously; if multiple rules' predicates are simultaneously satisfied only one (nondeterministically chosen) rule's actions are carried out. Thus, an execution of a Bluespec description corresponds to discrete steps of atomic rule applications, where each rule produces a state that satisfies the next rule's predicate condition.

A simple example below shows how the operation-centric semantics can be helpful in our description of the Itanium pipeline. A rule can specify far-reaching actions that logically pertain to the same event but are physically distributed over the datapath. In a pipelined processor design, actions associated with a branch misprediction recovery entail state modifications to numerous portions of the pipeline. In Bluespec, all such actions can be gathered and stated in a single rule guarded to take place only when encountering a branch misprediction. Given the atomic semantics of rule execution, the consequences of this rule are easy to understand, even in the context of other rules with conflicting actions.

2.2. Language features

Bluespec's original syntax[1] was derived from the Haskell functional programming language [10]. Bluespec applies object-oriented programming concepts to support clean and composible modular design partitioning. Functional programming constructs in Bluespec further allow concise specifications of combinational logic in a rule's predicate condition and state update expressions. For example, the use of *list* structures and *lambda* expressions allow more flexible and powerful combinational logic descriptions than the simple loops provided by standard HDLs. Moreover, Bluespec offers an extensive type system that is significantly more comprehensive than conventional HDLs. This type system enables the declaration of elaborate state-storage elements. Static type checking during compilation helps to eliminate a large class of errors at compile time.

2.3. Bluespec compiler

The Bluespec Compiler (BSC) produces the RTL-level Verilog description of an optimized synchronous implementation. Bluespec's atomic and sequential abstract semantics do not preclude a correct implementation from executing multiple rules per cycle. During compilation, the Bluespec compiler identifies "conflict-free" rule pairs (i.e. rules that can be safely executed in the same clock cycle to produce a combined state transition that is correct with respect to Bluespec's atomic and sequential semantics) [9]. The Bluespec compiler then synthesizes a synchronous implementation that executes as many conflict-free rules as possible each cycle. BSC Verilog output can be integrated with other components described by conventional HDLs for simulation and synthesis. BSC can also generate a cycle-accurate C simulator of the design.

[1] The current Bluespec revision [3] extends support to SystemVerilog.

3. Itanium model development

The goal of this project is to model a realistic processor microarchitecture for FPGA prototyping and design explorations. One of the key challenges is maintaining a balance between the level of modeling detail and the implementation effort. Below, we first give an overview of the Itanium microarchitecture and then describe the details of our Bluespec model.

3.1. Overview of the Intel Itanium architecture

The Intel Itanium Architecture is a 64-bit "EPIC" (explicitly parallel instruction-set computing) architecture [8]. Reminiscent of Very Long Instruction Word (VLIW) architectures, Itanium instructions are formatted as 128-bit bundles of three RISC-like instructions. The Itanium instruction set architecture (ISA) supports explicit demarcation of data-independent groups (ranging from one instruction to several bundles in length) in the instruction sequence. This explicit data-dependence encoding allows the Itanium ISA to be efficiently supported by straightforward VLIW-like microarchitectures. The Itanium ISA also incorporates other distinctively VLIW-like features such as a large rotating register file and predicated instructions. The first generation Intel Itanium processors (code named Merced) ran at 733 and 800 MHz and use an 8-stage core pipeline [16]. This first-generation microarchitecture are based on a "2-bundle" wide datapath; that is, the datapath can decode and issue up to two bundles, or six instructions, per cycle.

3.2. Modeled ISA subset

Presently, we support only a subset of the Itanium ISA in our Bluespec microarchitecture model. The chosen subset constitutes approximately one-third of the instruction encodings in the Itanium ISA (not including IA-32 compatibility modes). This subset concentrates on user-level integer, memory, and control flow instructions, which constitute the vast majority of integer instructions generated by the Intel Itanium C++ compiler. For example, the subset is sufficient to execute the Dhrystone integer benchmark [17]. Examples of omitted user-level integer instructions include vector operations, multiprocessor related operations, speculative and advanced loads, etc. Floating-point instructions, multimedia instructions, and privileged instructions are also not currently supported. The model obeys the true bit-encodings defined by the Itanium ISA. This requirement introduces some complexity in the decoding stages of the microarchitecture model, but we deem this necessary for a faithful prototype. This also allows us to use stock Itanium assemblers to produce executable binaries.

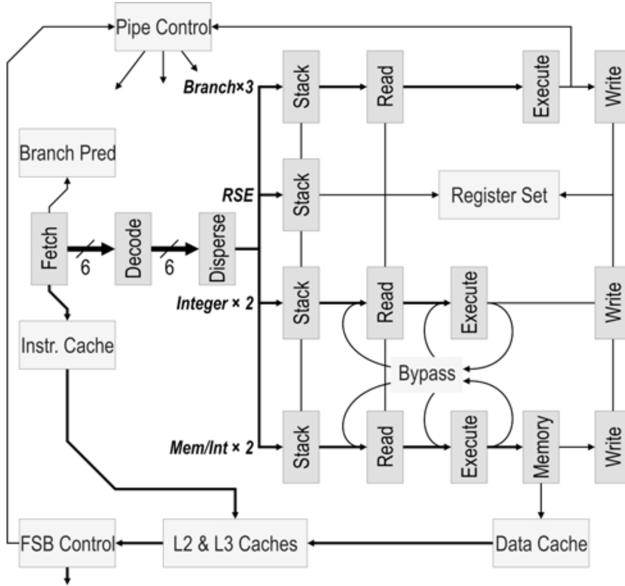

Figure 1. Bluespec Itanium microarchitecture model

3.3. Modeled microarchitecture details

The Itanium microarchitecture captured in our Bluespec model is primarily based on published descriptions of the first-generation Itanium microarchitecture [16]. A block diagram of the modeled microarchitecture is shown in Figure 1. Certain stages in the front of the pipeline differ from the published details to permit a more straightforward description under the Bluespec framework. For example, our model utilizes a single decoding stage, rather than distributing the decoding functionality across expand and rename stages. Below, we briefly describe the microarchitectural details included in our Bluespec model.

The fetch stage uses a two-level adaptive branch predictor to generate instruction fetch addresses to the instruction cache. This predictor has a 512-entry, 4-way associative per address BHT of 4-bit entries, 128 16-entry per address PHTs of 2-bit saturating counters, and a 64-entry branch target buffer. An instruction cache hit returns up to two bundles (depending on alignment) to the decoding stage. The decoding stage passes partially decoded instructions in the same instruction group to the dispersal stage. The dispersal stage attempts to issue as many instructions from the same instruction group as possible. The decode and dispersal stages can process up to 2 bundles per cycle, but only 1 instruction group per cycle. Like in a real Itanium, there is no out-of-order instruction issue.

The stack stage renames the effective register name to a real register name according to a simple offset in the circularly indexed rotating register file. We implement a simple register stack engine that blocks the pipeline to service compulsory stack spills and fills. Next, the register read stage fetches operand values from the register files. Instructions are stalled in the register fetch stage if non-bypassable read-after-write hazards are detected between instructions in different instruction groups.

The execution stages comprise of three types of execution pipelines: branch, integer, and memory/integer. The branch units determine the outcome of a branch instruction based on a predicate register value, possibly forwarded from the integer execution units. The branch outcome is compared to the predicted outcome, and mispredictions cause the pipeline control module to be notified. The resulting control module action flushes the wrong-path instructions, and corrects branch predictor entries and the current instruction pointer. The integer execution units (including the memory/integer units) are fully bypassed and support 64-bit arithmetic and logical operations as well as fixed-point multiply. The bypass control is predicated allowing speculative execution and forwarding of predicated results. The memory units perform reads and writes against the L1 data cache. Finally, the write-back stage commits register updates to the appropriate register file.

The cache hierarchy consists of three levels, separate 16 KB L1 instruction and data caches, a 96 KB L2 unified cache, and a 4 MB L3 cache. The L1 caches are both 4-way set associative with 32-byte lines and 2 cycle load latencies. The L2 is 6-way set associative, has 64-byte lines, and a 6-cycle load latency. The L3 is 4-way set associative with 64-byte lines and a 21-cycle load latency. Main memory latency is implemented to be 100 cycles.

3.4. Model development

The behavior of the microarchitecture model is described as 90 rules in about 9,500 lines of Bluespec code. Approximately one-third of the description is devoted to the decode and execution stages. Both stages are conceptually simple, but require extensive descriptions due to the elaborateness of the Itanium ISA encodings.

The operation-centric semantics and functional language syntax of Bluespec are very effective in reducing the development time of the Itanium microarchitecture model. Language features such as static type checking and terse functional language descriptions of combinational logic lead to fewer bugs. The atomic descriptions of control operations, as well as the ability to compose operations easily, made the Bluespec description simpler and clearer for corner case behaviors.

In return for the convenience of Bluespec's high-level design abstraction, we give up some control over the exact implementation details. This presents a number of problems in prototype development. First, a hidden

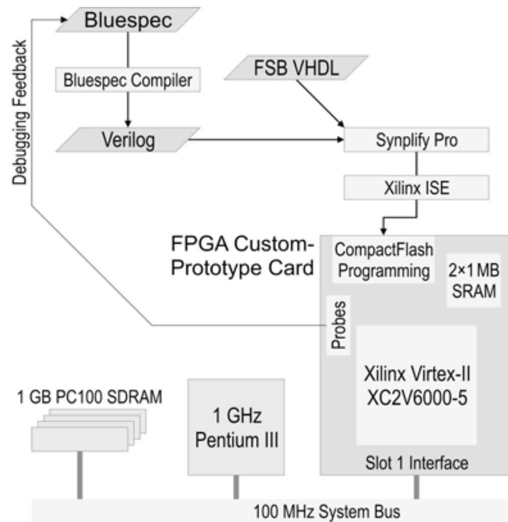

Figure 2. FPGA prototype system and workflow

danger in (all) high-level synthesis is the ability to imply huge combinational logic blocks unintentionally from seemingly short and simple expressions. This is particularly problematic when several valid coding options lead to drastically different circuit sizes (revealed only after synthesis). Similarly, since the description's high-level constructs are further removed from the final synthesized physical designs, it is more difficult to trace timing and space problems from the synthesized circuit back to a particular site in the Bluespec source code. Finally, Bluespec's abstract timing model improves operation composibility within the language environment but demands special care when attempting to interface with synchronously-timed external systems.

4. FPGA prototyping

Our FPGA prototyping platform enabled rapid development of a functional Itanium model with low latency and high bandwidth access to large memory resources. Figure 2 gives an overview of the key components and workflow of our FPGA prototyping system.

4.1. FPGA prototyping hardware

Our Itanium model is prototyped on a custom FPGA board provided by Intel for research use. The FPGA board connects a Xilinx Virtex-II XC2V6000-5 to the front-side bus (FSB) of a Pentium III motherboard via a Slot 1 edge connector [13]. Through the FSB, the FPGA can directly reference the main memory of a host PC. Operating at 100 MHz, the FSB interface provides the FPGA with a peak memory bandwidth of 800 MB/sec and a typical round-trip time of less than 150 ns.

The FPGA board has two banks of 256K×4 byte synchronous SRAM modules. The two-cycle pipelined SRAM modules operate at up to 166 MHz, for a sustained bandwidth of 667 MB/sec.

In our current setup, the FPGA board replaces one of the Pentium III processors in a dual-processor PC host. The remaining Pentium III processor runs a uniprocessor Linux operating system out of the lower half of the physical memory (1 GB). The upper 512 MB of physical memory is used by the FPGA-prototyped processor. The host Pentium III can read and write the upper 512 MB region through a special /dev/mem file handle. Thus, the Pentium III can communicate with the FPGA-prototyped processor via uncached shared-memory operations. The Pentium III can also issue low-level control messages to the FPGA by writing to special physical addresses snooped by the FPGA's FSB interface logic.

4.2. FPGA design instantiation

To instantiate a Bluespec microarchitecture model on the FPGA, the BSC-generated Verilog source code is combined with a VHDL wrapper code that implements the FSB interface. The FSB interface code enables the FPGA to participate as a master in uncached read and write bus transactions and as a passive snoop agent.

The combined Verilog and VHDL source files are compiled using Synplify Pro 7.3 to produce EDIF files required by Xilinx's Integrated Software Environment (ISE) 5.2. Xilinx ISE is used to map, place-and-route, and generate the FPGA configuration bit stream. We configure the FPGA using the ACE CompactFlash interface. By providing standalone power and configuration clocking to the FPGA board, we are able to program the FPGA prior to powering up the PC-host; this greatly simplifies the host boot-up processes.

In addition to the raw read-write FSB interface described in VHDL, we implement a three-level cache hierarchy model in Bluespec. The L1 caches are implemented using the Xilinx Virtex's internal block select RAMs. To achieve a dual-ported 4-way set associative L1 data cache we implemented the cache logic in a separate clock domain that ran at double the speed of the rest of the processor model. The 100 MHz FSB clock drove the cache clock domain directly, and a clock divider fed the remainder of the processor model with a 50 MHz clock signal.

The two lower levels of the cache hierarchy were too large to be implemented onboard the FPGA. Thus, we used the external SRAM to provide tag and data storage for the L2 cache, and tag storage for the L3 cache. The L3 cache model consults this SRAM tag storage to determine hit or miss status, but always fetches cache lines directly out of main memory. On a L3 hit, the cache line is be available to the core within 8 processor cycles

(at 50 MHz), thus the request is artificially delayed to mimic the desired 21 cycle load latency. Main memory requests are also delayed before being placed on the FSB to allow modeling of the correct relative speed between the processor core and main memory.

Our current Itanium microarchitecture model utilizes approximately 40% of the resources available on the Xilinx XC2V6000, while the FSB interface wrapper consumes an addition 4% of available resources. The current Itanium microarchitecture model and FSB interface synthesizes to 50 MHz.

4.3. Prototyped processor execution

The FPGA prototyped processor executes Itanium binary executables out of the reserved upper 512 MB region of the host-PC's main memory. We use the Intel Itanium C++ Compiler 7.0 to generate object files from C++, C, and Itanium assembly programs. The object files are decoded using the *objdump* utility software and reformatted by a Python script to produce executables for the FPGA prototyped processor.

Prior to execution, the memory space for the FPGA prototyped processor is initialized by the Pentium III host processor. The host processor uses uncached writes to load the executable binary and program data to a known location in the FPGA processor's address space. The host processor initiates the FPGA prototyped processor's execution by writing to a memory-mapped address snooped by the FPGA's FSB controller. The FPGA prototyped processor terminates execution by writing out relevant processor state (register file contents, program counter, and performance counters) to physical memory where they can be examined by monitoring software running on the host processor.

5. Model calibration & experimental results

Our FPGA prototype is designed to realistically model the details of an Itanium processor and to support microarchitectural design explorations. We evaluate these two goals by first calibrating our model's IPC performance to a real Itanium processor. Next, we apply the calibrated model to investigate the performance and implementation tradeoffs from reducing the data bypass network.

5.1. Performance calibration

We used two types of software benchmarks to compare and tune the performance of our FPGA model to an Itanium processor. First, we specifically designed microbenchmarks to exercise various features of the processor pipeline (as in [3]). We also measured overall

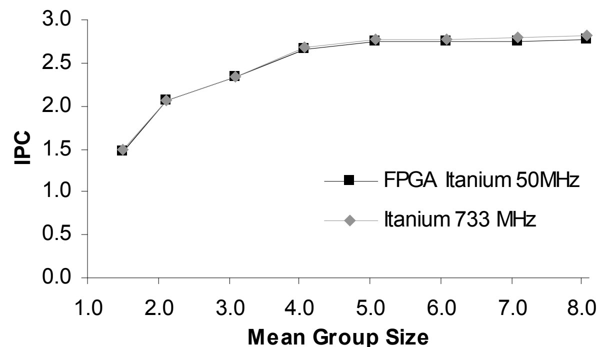

Figure 3. ALU microbenchmark performance

performance using the Dhrystone benchmark as an inclusive integer workload.

Execution pipeline. We first focused on the execution portion of the pipeline, verifying that the dispersal through write-back stages performed correctly for general ALU instructions. We generated Itanium assembly comprised of ALU instructions with random register operands. We varied the average size of the generated instruction groups (dependence free instructions) to exercise the different behaviors of the dispersal stage. Specifically, we tested how the dispersal stage handles split-issues when an insufficient number of execution pipelines are available. The results of executing this microbenchmark on our model and an Itanium processor are shown in Figure 3 as a plot of IPC vs. mean group size. Since this portion of the prototype model was based on detailed documentation of the Itanium processor, our model performed almost identically to the actual processor.

It is clear from Figure 3 that neither the Itanium nor our prototype model ever exceeds three instructions per cycle, regardless the of mean group size. This is a real phenomenon caused by Itanium's issue policy, which does not permit instructions decoded in different cycles to be issued in the same cycle. When presented with the first two decoded bundles of an infinitely long instruction group, a 2-bundle wide microarchitecture with four integer execution units can issue four independent integer instructions in one cycle. In the next cycle, however, only the remaining two instructions from the two decoded bundles are issued because the issue policy does not further consider subsequent bundles. This caps the peak IPC at 3.0 for program segments of purely integer instructions (unless carefully padded with NOP instructions). Our prototype model reflects this performance characteristic accurately.

Memory system. The second portion of our model that we calibrated was the memory system. We tested the

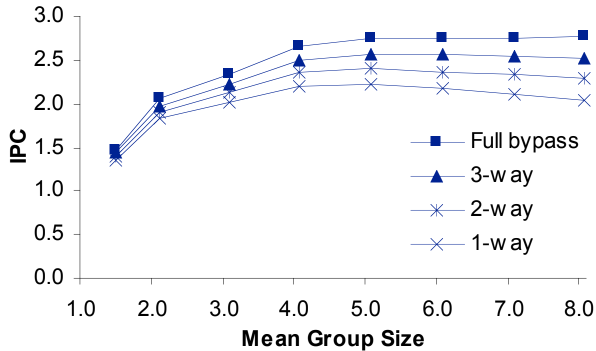

Figure 4. ALU microbenchmark performance

prototype model using a microbenchmark that performs loads and stores to memory regions of varying size for a spectrum of strides [15]. This benchmark produces a characteristic graph that reveals the details of the cache hierarchy when access latency is plotted as a function of stride and region size. Using this microbenchmark, we were able to verify that the basic caching behavior of our prototype model closely matches the Itanium processor.

System performance. To establish overall performance we compared the IPC of the Dhrystone benchmark on the Itanium processor with our prototype model. Initially, we had only implemented a single-ported L1 data cache to avoided introducing a second clock domain (see Section 3.3). For the Dhrystone benchmark, even with the calibrated execution pipeline, our prototype model with only a single-ported L1 data cache had a 34% error in IPC, relative to the real Itanium processor (0.94 vs. 1.43). Introducing the dual-ported L1 data cache reduces the IPC error to 11%.

There remain several aspects of the Itanium microarchitecture that we do not implement fully, which contributes to the remaining IPC error. For example, our prototype model currently only implements the primary branch predictors, but not other supporting mechanisms such as the target address registers, the multi-way branch prediction table, branch correction for modulo-scheduled loops and static prediction hints. To reduce the absolute IPC error further would require increasingly more details to be modeled precisely in the prototype, at which point the prototyping effort approaches that of a production development effort. Given the desire to reduce development effort, it is unrealistic to expect a prototype model to agree absolutely with a production design. What the prototype enables us to do is to quickly explore different design options and measure the effects in not only performance but also implementation metrics.

Table 1. Bypass network impact study

Bypass	Area reduction (mm^2)	Critical path (ns)	Dhrystone IPC
Full	—	4.29	1.27
3-way	0.35	3.99	1.21
2-way	0.53	3.76	1.15
1-way	0.68	3.57	1.09

5.2. Microarchitecture exploration

In this study, we use the prototype model to investigate the effects of abbreviating the bypass network that connects the integer pipelines. As in the real Itanium, our baseline model has a full bypass network that connects the output from the integer/memory execute stages to the operand inputs of both the read and execute stages. One could consider reducing the complexity of this network by allowing forwarding among a subset of the pipelines. Reducing the bypass network negatively affects the processor's IPC, but when moving to a wider microarchitecture a partially bypassed network is an important strategy to reduce implementation area and impact on cycle time.

Starting from our baseline prototype model, we can modify the high-level Bluespec description to derive alternative microarchitectures to investigate the impact of a partial bypass network. In a "1-way" configuration, each pipeline forwards results only to itself. The "2-way" and "3-way" configurations forward results to additional adjacent pipelines. To support a partial bypass network, we also have to modify the register scoreboard logic accordingly to account for the limited data forwarding.

Figure 4 plots the changes in IPC when the integer instruction microbenchmark is executed on prototype models with different bypass configurations. As expected, IPC performance decreases as the degree of bypass in network is reduced. An interesting behavior in the partial bypass networks is that group sizes beyond a threshold can actually come to negatively impact IPC. This behavior is caused by the increased probability of dependencies between consecutive groups in the microbenchmark. Notice that this evaluation is based on microbenchmarks of controlled instruction group sizes. A comprehensive study on the impact of partial bypass networks must also expose this microarchitecture feature to the compiler and consider the compiler interactions.

Besides IPC performance trends, the implementation impact of the design can be studied by synthesizing the prototype model. Table 1 reports the reduction in area and critical delay path (in the bypass network) when the different prototype models are synthesized for a 0.18μm standard cell library.

6. Conclusions

Simulation-based microarchitecture studies are powerful tools, but they can fail to uncover critical issues related to the final physical implementation. Prototyping helps to expose these issues and provides implementation estimates such as relative circuit area and cycle time metrics. In this paper, we described the FPGA prototyping of an Itanium processor. The processor model accurately recreates the primary components of the Itanium microarchitecture to provide an effective performance model. This prototype is also capable of providing supporting implementation metrics through synthesis. We intend to develop this platform further as a flexible prototyping system for application specific processor design.

Acknowledgment

We would like to thank Steve Haynal, Shih-Lien Lu, Konrad Lai, and Kevin Rudd for their feedback and assistance in this study. Funding for this work is provided by the Integrated Circuits and Systems Research program of the Semiconductor Research Corporation. We thank Bluespec Inc. for providing the Bluespec compiler and Intel Corporation for providing the FPGA platform.

References

[1] Arvind, R.S. Nikhil, D.L. Rosenband, and N. Dave, *High-level synthesis: An Essential Ingredient for Designing Complex ASICs*, Memo 473, Computation Structures Group, Massachusetts Inst. of Technology, 2004.

[2] L. Augustsson, J. Schwartz, and R.S. Nikhil, *Bluespec Language Definition*, Sandburst Corp., 2001.

[3] B. Black and J.P. Shen, "Calibration of Microprocessor Performance Models," *Computer*, vol. 31, iss. 5, May 1998.

[4] *Bluespec™ SystemVerilog Version 3.8 Reference Guide*, Bluespec Inc., 2004.

[5] R. Brown, J. Hayes, and T. Mudge, "Rapid Prototyping and Evaluation of High-Performance Computers," *Proc. Conf. on Experimental Research in Computer Systems, NSF Experimental Systems*, June 1996.

[6] J. Gaisler, *LEON/AMBA VHDL model description*, European Space Agency, 2000.

[7] M. Gschwind, V. Salapura, and D. Maurer, "FPGA Prototyping of A RISC Processor Core for Embedded Applications," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, Apr. 2001.

[8] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder and R. Zahir, "Introducing the IA-64 Architecture," *IEEE Micro*, vol. 20, iss. 5, Sept./Oct. 2000.

[9] J.C. Hoe and Arvind, "Synthesis of Operation-Centric Hardware Descriptions," *Proc. Int'l Conf. on Computer Aided Design (ICCAD-2000)*, Nov. 2000.

[10] S.P. Jones and J. Hughes, *Haskell 98: A Non-strict, Purely Functional Language*, tech. report YALEU/DCS/RR-1106, Yale Univ., 1999.

[11] Y.G. Kim and T.G. Kim, "A Design and Tool Reuse Methodology for Rapid Prototyping of Application Specific Instruction Set Processors," *Proc. IEEE Int'l Workshop on Rapid System Prototyping (RSP1999)*, June 1999.

[12] K. Oh, S. Yoon, and S. Chae, "Emulator Environment Based on an FPGA Prototyping Board," *Proc. IEEE Int'l Workshop on Rapid System Prototyping (RSP2000)*, June 2000.

[13] *P6 Family of Processors – Hardware Developer's Manual*; http://www.intel.com/design/PentiumII/manuals/244001.htm.

[14] W.B. Puah, B.S. Suparjo, R. Wagiran, and R. Sidek, "Rapid Prototyping Asynchronous Processor," *Proc. IEEE Int'l Conf. on Semiconductor Electronics (ICSE2000)*, Nov. 2000.

[15] R.H. Saavedra-Barrera, *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, PhD Dissertation, Univ. of California, Berkley, May 1992.

[16] H. Sharangpani and H. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, vol. 20, iss. 5, Sept./Oct. 2000.

[17] R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. of the ACM*, vol. 27, no. 10, Oct. 1984.