Superscalar Out-of-Order Demystified in Four Instructions

James C. Hoe
Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213
jhoe@ece.cmu.edu

Abstract—This paper describes a processor design project intended to illustrate the detail inner workings of modern superscalar out-of-order processors. In the project, the students implement a cycle-accurate RTL-level model of an out-of-order core—including rename, issue, execute, completion and retirement stages—based on the MIPS R10000. The processor core only supports four instruction types. First, the basic integer subtract instruction is included to exercise the mechanisms related to register-renamed out-of-order execution. Second, two types of branch instructions, resolving correctly and incorrectly respectively, exercise speculative execution and branch rewind capabilities. Lastly, an exception-triggering instruction tests the support for precise exceptions. The project is designed to be completed in six weeks by a team of two to three students with solid background and strong interest in computer architecture and digital design. This project has been used twice in an advanced graduate computer architecture course (CMU 18-744 Hardware Systems Engineering) *and has received favorable feedback from students and industry recruiters. The project handout and required Verilog source files can be downloaded from* http://www.ece.cmu.edu/~jhoe/superscalar.

I. INTRODUCTION

Implementing an n-stage in-order pipelined processor is the staple design project in undergraduate introductory computer architecture courses. Such a hands-on project is very instructive in that the students walk away with an in-depth understanding of not just the abstract principle of pipelining but also the exact mechanisms that make a real instruction pipeline work well (e.g., stalling, squashing and forwarding). Modern superscalar out-of-order microarchitecture, on the other hand, is a central topic in most graduate-level computer architecture courses. Unfortunately, due to the complexity of the subject, the material is often presented at a fairly high level; rarely are students required to work out a coherent, complete datapath in a hands-on fashion. It is our contention that most students are only able to walk away with a "warm-and-fuzzy" understanding of how an out-of-order core really operates. In other words, many students may comprehend the basic principle of microdataflow instruction scheduling, but very few students would be able to accurately describe the intricacies in the instruction issue and data forwarding logic that permit two instructions with read-after-write dependency to execute in back-to-back cycles.[1]

In this paper, I described a project designed to impart students with *precise* and *accurate* understanding of modern superscalar out-of-order processor design. In this project, the students implement a cycle-accurate RTL-level model of an out-of-order core, i.e., rename, issue, execute, completion and retirement stages. To stay within a reasonable workload, students only need to support the execution of four simple instruction types. First, the integer subtract instruction is sufficient to exercise all of the mechanisms involved in register-renamed out-of-order execution. Second, two types of branch instructions, resolving correctly and incorrectly respectively, exercise speculative execution and branch rewind capabilities. Lastly, an exception-triggering instruction tests the support for precise exceptions. The RTL model produced by the student must be both simulatable and synthesizable. The students' final RTL implementations are evaluated both in terms of IPC performance and hardware cost.

[1] Data dependence between a pair of dependent single-cycle ALU instructions is resolved in the same cycle when the producer instruction is selected for issue. This way, the dependent instruction itself is eligible for scheduling in the next cycle while the producer instruction is still being executed.

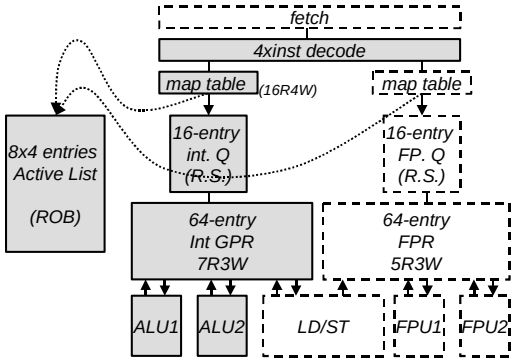

Fig. 1. MIPS R10000 Block Diagram

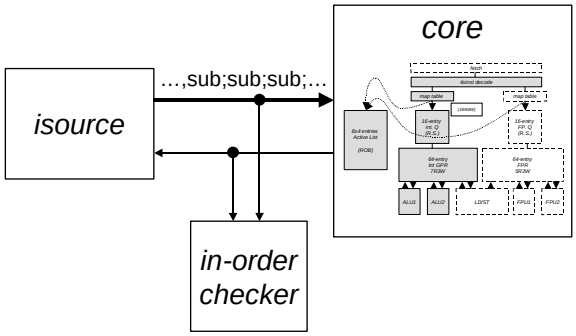

Fig. 2. The Simulation Environment

The project can be completed in six weeks by a team of two to three graduate (or advanced undergraduate) students who have solid background and strong interest in computer architecture and digital design. This project has been used twice in an advanced graduate computer architecture course (*CMU 18-744 Hardware Systems Engineering*, Spring 2002 and Spring 2003). CMU 18-744 is a depth course in our ECE department's graduate computer architecture curriculum. This course has as prerequisite our first-year graduate computer architecture course (*CMU 18-741 Advanced Computer Architecture*). The project is liked by the students who have taken the class and has gotten positive comments from industry recruiters who talked to the students.

Paper Outline: Following this introduction, the remainder of the paper is organized as follows. Section II gives an overview of the project specification. Section III and IV provide details in the project setup and execution, respectively. Section V explains the project's stated objectives and acceptance criteria. Section VI suggests ways to extend or expand the project in the future. Section VII concludes with a few remarks regarding our experience in running this project.

II. Project Overview

The project calls for the Verilog RTL implementation of a superscalar speculative out-of-order core based on the MIPS R10000 microarchitecture. Figure 1 gives a high-level sketch of MIPS R10000's out-of-order core. The details of the microarchitecture are described extensively in [5]. The MIPS R10000 microarchitecture is selected for the project because similar microarchitectural arrangements—most notably the use of a common physical register pool to serve as both rename registers and architectural registers—are employed by most of the recent superscalar out-of-order processors.

The scope of the project only covers the mechanisms for renaming, microdataflow scheduling, data forwarding, branch rewind and exception recovery. Datapath elements relevant to this project are highlighted in gray in Figure 1. Memory and floating-point portions of the datapath are left out in the current version of the project. For the purpose of testing and simulation, a synthetic randomized instruction source that mimics the instruction fetch stage provides test stimulus to the out-of-order core.

The synthetic instruction source emits a randomized instruction stream composed of 4 instruction types in the MIPS ISA [3]. The out-of-order core only needs to support the execution of the integer subtract instruction (SUB rd, rs, rt). A sequence of SUB instructions with randomized source and destination registers is sufficient to exercise the hardware related to register renaming, microdataflow scheduling, and data forwarding.

The synthetic instruction source also emits ADD, BNE and BEQ instructions. The semantics of these instructions, however, have been *redefined* for the purpose of exercising the mechanisms for precise exception and branch rewind.

- The execution of an ADD should always lead to an exception. The ADD instruction itself cannot be finished. The state of the out-of-order core should rewind to just before the ADD instruction prior to continuing with the corrected instruction fetch stream.
- The execution of a BEQ should always confirm its corresponding branch prediction, requiring no change to the incoming instruction stream.
- The execution of a BNE should always reverse its corresponding branch prediction. The state of the out-of-order core should rewind to just after the BNE prior to continuing with the corrected instruction fetch stream.

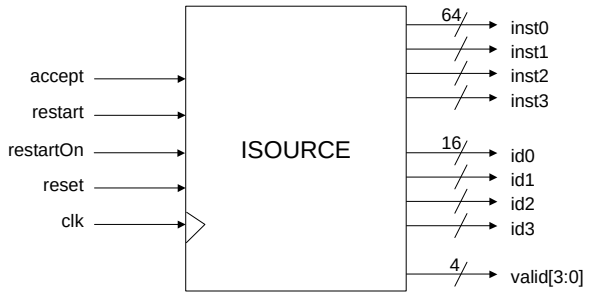

Fig. 3. The isource Module Interface

The frequency of these special instructions can be adjusted as necessary. The out-of-order core implementation is not allowed to take advantage of the special semantics of the ADD, BEQ and BNE instructions except when the instructions are being executed. In other words, until ADD, BEQ and BNE reach the execution unit, they must be treated normally as if they were expected to complete; similarly, the speculatively fetched wrong-path instructions following an ADD or BNE must also be treated normally (although they must be later discarded) until the exceptional condition is determined in the execution unit.

III. PROJECT SETUP

The students are provided with two behavior-level Verilog modules that constitute the testbench environment for developing their core (Figure 2). The first is a synthetic instruction source, and the second is a checker module.

A. Instruction Source Module

The isource module mimics a 4-wide instruction fetch buffer. The interface to the isource module is depicted in Figure 3. On each cycle, the isource module presents a sequence of 0 to 4 randomly generated instruction words on its four inst ports. The corresponding bits in the 4-bit valid mask indicate the validity of individual instruction words. If less than four instructions are valid, the valid instructions are always clustered together toward inst0. The output of the inst and valid ports do not change until the accept input port is asserted on a clock edge. (See example waveform in Figure 4). In other words, the instruction fetch stream can be stalled by deasserting accept. The instructions that follow a BNE instruction are, by our redefinition, wrong-path instructions and hence must be discarded after the BNE instruction is later executed. After branch

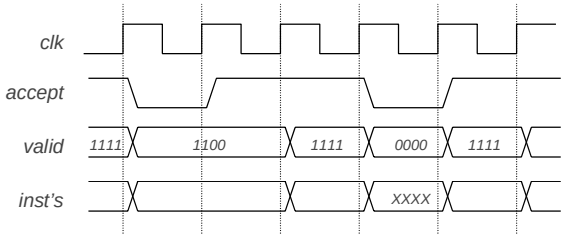

Fig. 4. Stalling Fetch by Deasserting accept

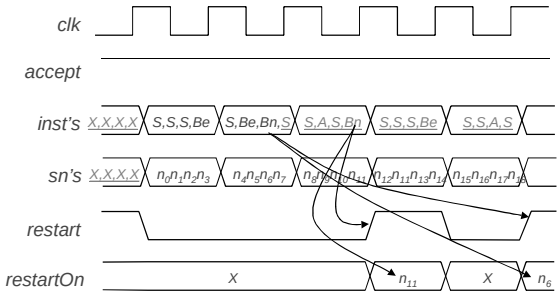

Fig. 5. Restarting Fetch by Asserting restart

rewind, the correct instruction stream is resumed by asserting the restart input port for 1 clock edge. (See example waveform in Figure 5.) The new instruction stream begins immediately on the following cycle. Instruction fetch is restarted in the same way following a precise exception caused by an ADD instruction. The isource module generates a sequencing ID for each instruction in the stream. The sequence ID of the exceptional instruction must accompany the assertion of the restart signal to properly resolve the situation when nested branch mispredictions are resolved out of program order.

B. Checker Module

The checker module maintains a shadow copy of the architecture register file. The checker module passively monitors the activities on all input and output ports of the isource module. The checker module computes the correct in-order-state of the register file according to the observed instruction stream. The checker module executes the instructions in order. Instruction processing is skipped following an ADD or BNE instruction until the restart signal is asserted for the correct exceptional instruction. The checker module provides a reference state to verify the execution of the out-of-order core. The checker module also collects and displays basic performance and instruction stream statistics (e.g., IPC, instruction mix, and the number of exceptions) during

simulation.

IV. Project Execution

Four major milestones demarcate the different phases of the project. These milestones both help pace the students' effort and also steer the students' attention.

- **Phase 1:** Develop a one-instruction-wide out-of-order core for just the SUB instruction. The core only needs to handle one instruction per cycle in each of the decode, dispatch, execute and writeback stages. The emphasis in this step is to develop the register renaming and dataflow algorithms. This step is allotted 2.5 weeks, which includes allowance for getting up to speed on the MIPS R10000 microarchitecture.
- **Phase 2:** Extend the one-instruction-wide core to support branch instructions (BNE and BEQ) and branch rewinds.
- **Phase 3:** Extend the one-instruction-wide core to also support precise exceptions. Step 2 and 3 are together allotted 2 weeks.
- **Phase 4:** Extend the fully-capable core from one-wide to superscalar operations in all stages. This step is allotted 1.5 weeks.

An appropriately restricted isource module is provided for in each phase to facilitate testing of the restricted core in the first three phases.

The project can run alongside of a normal lecture sequence. However, a part of each lecture should be reserved to discuss and clarify project related issues. As necessary, a number of the lectures can also be devoted to covering the more subtle details of the MIPS R10000 design. Another option is to have the students take turn presenting different aspects of the MIPS R10000 core, as described in [5].

The RTL models produced for the project must not only simulate correctly but also be synthesizable. Synthesizabilty is a project acceptance criterion to ensure the students do not include unrealistic hardware structures in their processor models. The students can only implement the core using the synthesizable subset of Verilog Hardware Description Language [2]. In this regard, students with RTL design experience have a significant advantage. Therefore, it is important each team includes at least one member who is familiar with the RTL design flow.

Students are encouraged to follow a top-down design flow where they begin with a very high-level, possibly behavioral, model for the major datapath structures. Next, they can refine the datapath structures piecewise from behavioral Verilog down to synthesizable RTL code. After each refinement step, the students can immediately simulate against the testbench environment to ensure the correctness of the most recent changes.

Although the core must be implemented using the synthesizable subset of Verilog, students are encouraged to incorporate behavioral-level Verilog code for monitoring and debugging purposes. These out-of-band behavioral debugging code can dynamically examine the processor state cycle-by-cycle and compute elaborate runtime invariant conditions. In our experience, carefully designed runtime invariants are very powerful debugging aids. These invariants help flag an erroneous operation in a timely manner such that they allow much better localization of the origin of that error.

V. Project Objectives

A. General Implementation Guidelines

It is suggested that the students base their core on the MIPS R10000. As a general guideline, their RTL models should capture all details explicitly mentioned in [5]. Nevertheless, the students also need to improvise in several places where design decisions are not spelled out explicitly. In addition to the general guideline above, the following set of specific criteria must be met by the students' RTL models.

1) The execution of the out-of-order core must correspond to the reference behavior of the checker.
2) In the out-of-order core, pending instructions must issue as soon and as fast as possible after (true) data dependence and structural hazards have been cleared.
3) Back-to-back dependent instructions must be capable of issuing on consecutive cycles (in the absence of structural hazards).
4) Branch rewind must be fast, i.e., taking O(1) time, independent of the number of instructions to rewind.[2] A branch rewind must start and complete as soon as possible, without waiting for older instructions to retire.
5) Exception rewind can be slow, i.e., taking O(n) time where n is the number of instructions to rewind.
6) The core cannot make use of the special semantics of ADD and BNE instructions until they are in the functional unit.

[2] This criterion forces the student to implement a "branch rewind stack".

Other aspects of this design project are essentially left to the students' whim.

B. Evaluations

The quality of the resulting RTL model is judged on correctness, IPC performance and hardware cost. For the early milestones, the acceptance criterion is simply for the out-of-order core to simulate correctly against the test environment (isource and checker) for an agreed upon number of instructions. During simulation, the checker keeps count and reports the progress. After a sufficient number of instructions has elapsed, instruction fetch from isource is stopped, and the out-of-order core is switched into the "drain" mode. The students can next verify that the out-of-order core's committed register file state agrees with the reference register file state in the checker.

In Phase 4, the out-of-order core must also achieve a minimum IPC performance. The IPC lower bound helps diagnose performance bugs. For example, we had seen a case where one team did not handle back-to-back execution of dependent instruction. The consequence of this oversight is obvious in their abysmal IPC relative to the other teams. During Phase 4, the different teams in the class are routinely informed of each other's latest IPC achievements. This effectively turned the last phase of the project into a competition. The students were self-driven to fine-tune things like the issue priority logic and the branch rewind process to stay ahead of the rest of the class.

Although the RTL models produced by the students are synthesizable, we do not use the synthesis outcome to evaluate hardware cost. Synthesized storage structures (RAM, CAM, and FIFO) are much less efficient that the custom blocks instantiated in real processor implementations. As a compromise, we compute an estimate manually using empirical values. We assume each bit of storage (a bit cell) costs 1 unit and the integer ALU costs 50,000 units. To compute the final cost of a storage structure, the raw bit-cell cost must be further multiplied by the number of normal read or write ports and by two times the number of associative lookup ports. (For logical structures that require different types of references in its different columns, the students would have to break the logical structure into its physical components to get an accurate estimate.) This very coarse grain model does not account for random logic, registers or routing congestions. Nevertheless, it does force the students to be aware of the tremendous cost and tradeoffs of adding additional ports and associative lookup.

VI. Project Extensions

The project as described is designed to be completed by groups of two to three students in a half semester (~six weeks). The duration and scope of the project can be extended by including other aspects of modern high-performance processors, such as branch prediction, aggressive load/store ordering and cache hierarchy. Here, we briefly suggest some specific ideas.

1. Instead of MIPS R10000, one could also retarget the project to be based on Alpha 21264 with clustered datapath. This microarchitecture is as described in detail in [3].
2. One could augment a baseline implementation of MIPS R10000 with support for Simultaneous Multithreading (SMT) [1]. The project would serve to clarify the implementation consequences of SMT support.
3. The project currently does not handle loads and stores. The memory subsystem in modern superscalar processors can easily be made into a similar but stand-alone project.
4. Similarly the project can also be extended to examine instruction fetch and prediction issues in modern superscalar processors (e.g. wide fetch using a trace cache).

The key is to carefully confine the scope of the project so the students are exposed to all of the important details but without unnecessary tedium.

VII. Conclusion

The intent of this project is for students to gain an in-depth understanding of modern superscalar microarchitecture through hands-on practice. In addition, the project also gives the students a chance to experience issues in project teaming and participate in a full engineering cycle of specification, implementation, validation and analysis. We believe this course is invaluable training for students who are headed for either industry or graduate research.

This project is challenging and time-consuming. Although the students often gripe about its load and difficulty, in our experience, the students really did enjoy spending the time to work out the details, especially for the moments when a fuzzy concept in their head suddenly becomes crystal clear in their implementation.

REFERENCES

[1] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol 17 no 5, pp 12-19, Sep/Oct 1997.

[2] *HDL Compiler for Verilog Reference Manual*, Synopsys, Inc., 2000.

[3] G. Kane and J. Heinrich, *MIPS RISC Architecture*, New Jersy:Prentice Hall, 1991.

[4] R. E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro, vol 16, no. 2, pp 24-36, Mar/Apr 1999.

[5] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Mic*ro, vol. 16, pp. 28-41, Apr 1996.