A Lab Course on Computer Architecture

Pedro López José Duato

Depto. de Informática de Sistemas y Computadores
Facultad de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n, 46071 - Valencia, SPAIN
E-mail: {plopez,jduato}@gap.upv.es

Abstract

In this paper, the laboratory exercises for a course on computer architecture in the Computer Engineering degree at the Universidad Polit´ecnica de Valencia are presented. They have been designed for a course length of 30 hours and they only require standard personal computers and some simulation tools.

1 Introduction

The Computer Engineering degree at the Universidad Polit´ecnica de Valencia is organized as 5 academic years, each one consisting of two semesters. Each year accounts for 75 credits (1 credit: 10 hours). At the end of the 5th year, a final year project (equivalent to a master thesis) must be carried out. Computer architecture and related matters are covered in five courses: Computer Fundamentals (1st semester), Computer Organization I (2nd semester) and II (3rd semester) and Computer Architecture I (7th semester) and II (8th semester). We will focus on the last two courses. The authors have taught these courses for the last 8 years.

The students of Computer Architecture I and II know the assembly language of a modern MIPS-like computer. They also learnt logic design and the basics of computer organization. The approach we follow in these courses is based on the quantitative one found in [1]: design choices should be oriented toward achieving the maximum performance for a given cost. The courses syllabus includes performance analysis, instruction set design, simple and advanced pipelining and the basics of vector computers and multiprocessors.

Computer Architecture I and II account for 4.5 credits each one divided into 3 lecture credits plus 1.5 laboratory credits. Since a semester lasts for 15 weeks, each course must be taught for 2 lecture hours and 1 lab hour per week. As usual, the aim of laboratory exercises is to put into practice and complement the concepts explained in the lectures. However, there are also some constraints that must be taken into account. First, we wanted the lab course to be representative of almost all the topics of the course. Considering 4 to 5 different lab exercises, and taking into account the time dedicated to laboratory (15 hours), an average of three hours per lab exercise should be considered. As a consequence, lab work should be precisely delimited in order to take advantage of this somewhat limited time. With respect to class scheduling, we arranged laboratory exercises

in such a way that they are taught immediately after the corresponding lecture. Finally, there was a cost constraint. The required equipment should be the standard one for an educational laboratory: personal computers. This can be accomplished with the use of simulators. These tools allow the experimentation with a variety of systems at a very low cost. At the same time, simulators also hide the complexity of real machines, highlighting only those aspects that have to be analyzed. On the other hand, there are more than 200 students per year for each course, with an average of 30 people in each laboratory session. Thus, it is not feasible to organize a lab course where the student has to design some hardware and then implement, analyze and debug it using a logic analyzer, or to learn a complex programming environment to design parallel programs. Of course, these things are also very interesting, but we neither have time nor enough resources to do it. These things can be accomplished in the final year project.

In section 2, the syllabus for Computer Architecture I and II courses is presented. In section 3, the laboratory course developed is presented and commented.

2 Computer Architecture I and Computer Architecture II syllabus

Computer Architecture I syllabus consists of the following topics:

- Definition of computer architecture. Classic approach. Quantitative approach.
- Performance evaluation of architectures. Definition of performance. Popular measures of performance. Execution time. Quantitative principles of computer design.
- Instruction set design. Instruction set architectures taxonomy. Memory addressing. Operations. Operands. Register organization.
- Pipelining. Concept. Speed-up. Clock cycle. Reservation tables.
- Arithmetic pipelines. Pipelined adders. Pipelined multipliers. Pipelined dividers.
- Instruction pipelines. Instruction cycle. Instruction cycle pipelining. Structural hazards. Data hazards. Control hazards. Dealing with interrupts.
- Advanced memory design. Memory subsystem performance. Improving cache-memory performance. Improving main memory performance.

Computer Architecture II syllabus consists of the following topics:

- Multicycle operations. Instruction pipelines with multicycle operations. Hazards. Dealing with interrupts.
- Dynamic instruction scheduling. ILP. Dependencies. Special compilation techniques. Dynamic instruction scheduling. Dynamic branch prediction.
- Speculation. Motivation. Conditional instructions. Compiler speculation. Hardware speculation.
- Superscalar computers. Superscalar processors. Superpipelining. VLIW processors.

- Vector computers. Motivation. Basic vector architecture. Vector length. Chaining. Vector access sequences. Conditionally executed sentences. Measures of vector performance.
- Multiprocessors. Multiprocessor basics. Taxonomy.
- Shared memory multiprocessors. Interconnection networks. The cache coherence problem. Cache coherence protocols.
- Multicomputers. Interconnection networks. Switching. Topology. Routing.

3 Laboratory Course

The laboratory course for Computer Architecture I and II consists of the following lab exercises:

1. Performance evaluation. Amdahl's law.
2. Instruction set measurements.
3. Pipelined design.
4. Instruction pipeline design.
5. Programming pipelined computers.
6. Increasing ILP with compilation techniques.
7. Dynamic scheduling: Tomasulo's algorithm.
8. Speculation.
9. Vector computer programming.
10. Multicomputer programming.

Computer Architecture I includes the first 5 exercises, whereas Computer Architecture II comprises the last 5 ones.

3.1 Performance evaluation. Amdahl's law.

The performances of the laboratory computers are analyzed and compared. Two real programs (an Occam language compiler and a multicomputer simulator) and the well-known Dhrystone and Whetstone benchmarks are used. Performance is compared using several measures: individual programs, total execution time, weighted execution time and geometric means of normalized execution times. A comparison is also performed between real programs and benchmarks measures. In addition, in this experience, Amdahl's law is used in an unusual way. We want to analyze if it is interesting to buy some enhancement. The speed-up of the enhancement is known, but the fraction of time F that applications spend using the enhancement is unknown. In particular, we consider buying a faster disk as the enhancement. The solution in this case is simple. Run the program using the available hard disk, and using another available disk, for instance the floppy disk. Then, the speed-up of the program is obtained. If the speedup of the hard to floppy disk is known (or obtained using a test program that only does disk I/O), it is easy to obtain an estimation of F. Then, it is easy to apply Amdahl's law with the new disk.

3.2 Instruction set measurements.

We measure some instruction set features usage of a 80x86-based architecture running MS-DOS programs: top-ten instructions, taken/untaken branches, branch displacement distribution, PC addresses distribution, etc. We set the trace bit of the 80x86 processor and change the corresponding interrupt vector to point to a procedure that analyzes the instruction just fetched. This procedure is linked with the object code. This allows a precise analysis of executed programs, but it only works if the source code is available. Although these routines can be kept resident prior to the execution of programs, the problem is that MS-DOS real applications reset trace bit when they begin execution. In order to analyze programs for which the source code is not available, we use the following trick. We modify the timer interrupt service routine adding some lines of code to it to set trace mode again. This allows the students to obtain approximate statistics for real programs.

3.3 Pipelined design.

A pipelined 4-bit adder written in the VHDL hardware description language using an structural style is given to the students. They must analyze its behavior by simulation. Then, the effect of real latches delay is analyzed. Clock cycle must be increased in order to keep the pipeline working. The Earle latch is then introduced and compared against the conventional one. Some clock skew is also introduced, leading to an additional increase in clock cycle and a constraint in the lower bound for stage delay. Finally, a new pipelined unit with feedback is considered, being unable to initiate a new operation every clock cycle. The collision-free initiation hardware must be designed and tested.

3.4 Instruction pipeline design.

A DLX-like pipelined computer simulator written in the C programming language is given to the students. It is able to simulate a subset of the DLX instruction set. Care has been taken to use the same events, signals, logic functions and style of description as in the lectures and textbook [1]. Branch instructions update the PC register in the 4th pipeline stage, leading to 3 cycles of branch delay slot. The students must modify the logic to reduce the branch delay slot to only one clock cycle. Then, the logic to deal with RAW hazards must be designed, adding it to the simulator. First, a simple RAW between arithmetic instructions that can be solved with forwarding is considered. Then, a RAW hazard in which the first instruction is a load must be solved, being necessary to stall the pipeline and also forward results between stages. Notice that we use C as the hardware description language, since it is well known to the students, and we are not interested in timing analysis.

3.5 Programming pipelined computers.

A full-featured DLX simulator [2] is given to the students. It is based on the DLX simulator provided with the textbook but with a more friendly interface and some add-ons. In addition, a sample assembly code is given. The students have to modify this code to reduce the number of stalls, and fill the branch delay slot with a useful instruction. Finally, a completely new program must be developed. We use the convergence division algorithm that has already been studied in the lectures as a way to exploit a pipelined multiplier to perform divisions, analyzing its behavior as a function of the operands.

3.6 Increasing ILP with compilation techniques.

A simple DLX program loop that requires multi-cycle instructions is given to the students. The execution of the code leads to a very poor performance due to the high number of stalls. Then, the unrolled and software pipelined versions of the loop (also provided) are executed, observing the increase in performance over the conventional code. Then, a new loop (for instance DAXPY, $a*\vec{x}+\vec{y}$) has to be written, unrolling and software pipelining it, and doing a performance comparison among the three versions.

3.7 Dynamic scheduling: Tomasulo's algorithm.

In this lab exercise, an incomplete simulator written in the C programming language of a dynamic instruction scheduler that applies Tomasulo's algorithm is given to the students together with some sample DLX code. The student's job is to complete the code for Issue and Write Back stages, in order to have the simulator working. All the data structures and procedures to show them after each clock cycle are included in the provided code, and also the code for all the non-trivial required operations (for instance, to fetch a new instruction from the queue, to write a piece of data on the common data bus, etc).

3.8 Speculation.

In this lab exercise, an incomplete simulator written in the C programming language of a hardware speculative DLX is given to the students together with some DLX test programs. The provided code includes all the data structures, procedures to show them and the implementation of the instruction fetch, issue, execute and write back stages. The student's job is to write the code for the Commit stage in order to get a working simulator.

3.9 Vector computer programming.

By using an improved version of the DLX simulator that is able to run vector instructions [2], in this lab exercise a DLXV program (for instance the DAXPY loop) is written and executed, comparing its performance with the equivalent scalar code. The influence of chaining on performance is also analyzed. Then, in order to deal with vector length, stride and memory subsystem performance, a matrix is assumed as input data and a single row, column or the main diagonal is processed, respectively, comparing the obtained execution time. Then, a conditionally executed sentence is introduced in the original loop and a new version of the DLXV code must be written using indexed load/stores. Finally, a strip-mined version of the original loop that allows the processing of any vector length has to be written. By using this program, the measures of vector performance R_∞, $N_{\frac{1}{2}}$, and N_v have to be obtained, comparing them with the values computed using the model for vector performance studied in the lectures.

3.10 Multicomputer programming.

In this last exercise, the basics of the message-passing programming paradigm are presented to the students. In particular, a simple parallel program is written using the message-passing and SPMD (single program multiple data) model. We use a multicomputer simulator that executes the same

program on all the simulated nodes, conditionally running some pieces of code as a function of the processor identifier. Simple send and receive procedures are provided to allow parallel tasks to communicate. Topology and timing parameters of the node are configurable. Store and forward is used as the switching technique. A sample program is given to the students and they have to execute it using the simulator for different machine configurations. Finally, a new simple program has to be written.

References

[1] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffman, 1996.

[2] P. L´opez, R. Calpe, *DLXVSim: A DLXV vector computer simulator*, http://dlxv.disca.upv.es/tools/dlxv.html