Read, Use, Simulate, Experiment and Build : An Integrated Approach for Teaching Computer Architecture

Ioannis Papaefstathiou and Christos P. Sotiriou

Department of Computer Science,
University of Crete,
P.O. Box 1385, Heraklion, Crete, GR 711 10, Greece.
{*ygp,sotiriou*}*@ics.forth.gr*

Abstract

In this paper we present an integrated approach for teaching undergraduates Computer Architecture. Our approach consists of five steps: "read", which corresponds to studying the textbook theory, "use", which corresponds to using a simulator with appropriate graphical features to visualise the application of the theory, "simulate", which corresponds to developing an architectural simulation, "experiment", which corresponds to modifying the architectural simulation and observing the impact that changes make to performance, and finally "build", which corresponds to developing a low-level hardware model in a standard Hardware Description Language. In our experience, going down to the gate-level is of great importance, as students often find difficult to visualise how different architectural approaches affect the actual hardware (both datapath and control). By following this five-step approach in our teaching we observed a significant increase in both student performance and interest in Computer Architecture and hardware design.

1 Introduction

The subject of Computer Architecture is widely recognised as a significant and essential part of the undergraduate syllabus of university degrees related to computer or hardware design. One of the main problems with teaching Computer Architecture, is that students should not only understand the textbook theory, but more importantly its application in real systems and the impact that different architectural approaches have on the complexity and the performance of a system.

Thus, to make the teaching process more effective we have chosen to use an educational approach which we based on five steps: Read, Use, Simulate, Experiment and Build. In this paper we describe these five teaching steps and focus on the ones we believe are yet uncommon, however have been very effective in our experience.

2 "Read": Textbook Theory

Our Computer Architecture teaching is based on the Hennessy and Patterson Computer Architecture textbook, "Computer Architecture: A Quantitative Approach" [1], currently recognised as the most extensive and complete reference on the subject. Our course is tought in the last year of the Computer Science undergraduate degree, *i.e.* year 4, and runs for a duration of 14 weeks. As our teaching philosophy relies on combining theory with practice, we prefer to give students practical experience than a vast amount of theory. Thus, in 14 weeks we cover the first five chapters of the book, both in terms of theory and practice.

3 "Use": HASE Simulator

After the "Read" stage, students are given simple exercises on a graphical simulator. Our simulator of choice is the HASE [2] environment. HASE (Hierarchical computer Architecture design and Simulation Environment) is a graphical design, simulation and visualisation environment that can be used for both teaching and research. We use the DLX HASE model developed at the University of Edinburgh. HASE allows students to visualise both the overall structure of the DLX architecture and the execution of instructions by observing the step-by-step progress of individual events. HASE also allows for students to explore the impact of architectural parameters to the performace of the architecture, as students can change these using only the GUI environment (Graphical User Interface) and then re-run the simulation.

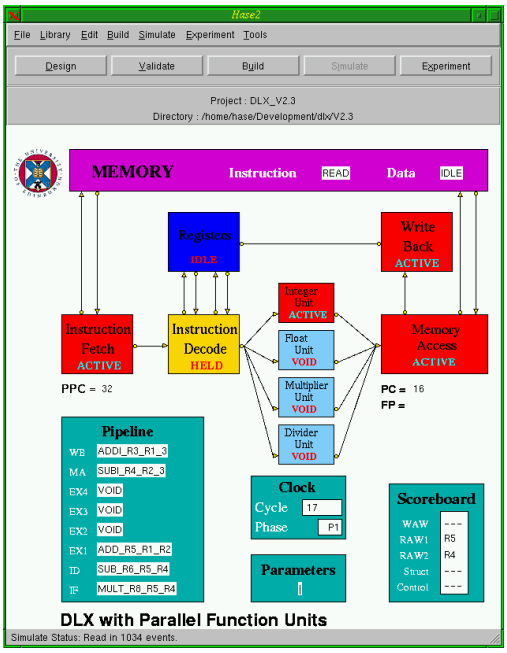

Figure 1: The HASE DLX Model

The DLX HASE exercises require students to write DLX assembly code and execute it on the HASE environment. With the help of the simulation environment students can measure the execution time, study the execution of each instruction in detail (passing through each pipeline stage) and the impact of architectural parameters. Students are asked to reason about the execution time of their program and to optimise their code based on their reasoning. They can experiment with different code schedules and diffrerent parameters and evaluate the execution time with the aim of finding the best possible cases.

Since using HASE as part of our teaching, rather than the standard pen-and-paper ones, we observed a significant increase in the students understanding and performance in the written examinations. This is probably due to the fact that by getting hands-on experience of the theory covered, students gain deeper and more thorough understanding.

4 "Simulate and Experiment": Develop a Simulator

The next stage of the course requires for the students to implement their own architectural simulation using a standard Hardware Description language (HDL), *i.e.* Verilog in our case. In this stage the implementation of the architecture is to be at the behavioural level. The students are asked to implement a RISC CPU called ARCP. The reason we chose an alternative to the DLX architecture was to give students something more challenging than simply re-implementing the DLX, which they already are familiar with at this stage from the HASE simulations.

4.1 ARCP - A 2-way Issue Architecture for Teaching

The ARCP architecture is based on the DLX, and has a very similar instruction set, however it is slightly more complicated, being 2-way superscalar. ARCP fetches two instructions at the same time from its instruction memory, which should be aligned and independent of each other for reasons of simplicity (students are given only 6 weeks of term for completing the whole project).

The main characteristics of the ARC architecture are :

- 64 General Purpose Registers.
- 32-bit address and word lengths.
- byte addressable, big-endian architecture.
- support for two data types: words (32-bits) and bytes (8-bits).
- 2-way fetch and execution of *independent* instructions; the independence of instructions must be ensured by the compiler/assembly programmer.
- only one control instruction (branch or call instruction) is allowed in an instruction pair and it must be placed in the first of the two instructions.
- only one memory reference instruction is allows in an instruction pair and it must be placed in the second of the two instructions.
- any number of arithmetic/logical operations are allowed.
- same memory used for instructions and data and self-modifying code is not allowed.
- memory can only be accessed using load or store instructions.
- branches are not delayed.
- register 0 is hardwired to 0.
- there as no condition codes; comparison instructions write a 1 (for true) or a 0 (for false) at a desstination register.
- conditional branches are PC-relative while unconditionals (call instructions) may be PC-relative or register-indirect; unconditionals store their current address in their destination register.

4.1.1 ARCP Instruction formats

The three different instruction formats and the format of an instruction pair are shown in Figure 2.

4.1.2 ARCP Instructions

All supported instructions along with their opcodes and formats are shown in Figure 3.

Most of these instructions are straightforward and found in the majority of RISC style architectures. The

4 MS bits	3 LS opcode bits							
opcode	000	001	010	011	100	101	110	111
0000	add R	addi I	sub R	subii I	mul R	muli I		cmgti I
0001	cmeq R	cmeqi I	cmne R	cmnei I	cmge R	cmgei I	cmlt R	cmlti I
0010	and R	andi I	or R	ori I	xor R	xori I	gcp R	cmlei I
0011	shru R	shrui I	shrs R	shrsi I	shl R	shli I		sethi L
0100	ldbu I	ldbs I	ldw I		stb I		stw I	
0101	breq L	brne L	brge L	brlt L			callr R	call L

Figure 3: ARCP Instructions and Opcodes

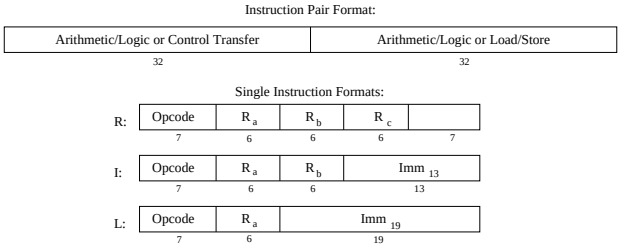

Figure 2: ARCP Instruction Formats

only unusual ones are the `sybii` and `gcp` instructions. The `sybii` instruction corresponds to a subtract immediate inverse operation, *i.e.* subtracts the register operand from the immediate, thus inverting the order of the subtraction. The `gcp` instruction corresponds to a guarded copy operation. A guarded copy operates using three registers and copies the source register into the destination if the third register, the guard, is not equal to zero. Guarded copy instructions can be used for implementing if-then-else blocks without branches and therefore can improve the efficiency and performance of the pipelining.

4.2 ARCP Simulation and Evaluation

In the "Simulate and Experiment" phase of the project the students are asked to build a behavioral simulation of this CPU and collect a set of measurements based on a number of small benchmark programs. Some of these benchmarks are provided by the lecturers, whereas the rest are to be developed by the students and are to be representative of typical applications. In our view, letting the student deal with the problem of finding the best benchmarks for evaluating the performance of the processor is really important, as it makes them really think hard of all the underlying issues involved. To help students achieve this, our research group has developed simple compilers and assemblers which students can use to produce their benchmarks.

The measurements that we are asking the students to provide (and we believe they are the most important for such a simulation) are the following:

- number of useful instructions executed (non NOOP).
- number of instruction pairs executed.
- average number of useful instructions.
- average number of memory reads per pair.
- average number of memory writes per pair; the last two are important for understanding the use of the memory hierarchy and the impact of having different data and instruction memories.
- number of taken and not-taken branches.
- percentage of useful instructions for each of the following groups: add/sub/mul, compare, and/or/xor, shift, gcp, load/store, branch, subroutine-call and jump.

Towards the end of the course students are asked to write a report which describes possible optimisations on the above architecture based on their simulation results. They are also asked to run new experiments on their architecture so as to support their claims for the possible optimisations. We believe that this idea of students proposing possible optimisations given an initial architecture is a crucial skill that a Computer Architecture student should acquire.

5 “Build”: Implementing the ARCP CPU in an HDL

The last stage of the course involves the development of the ARCP CPU, using synthesisable and structural HDL code based on a set of pre-implemented “library” components which we have developed for this exercise. The ARCP instruction set has been designed with emphasis on straightforward mapping to a gate-level circuit description. The students are asked to implement the ARCP CPU using a five stage pipeline, similar to the DLX pipeline of the textbook. This is shown in Figure 4.

Figure 4: ARCP CPU Pipeline

We provide students with the following pre-implemented library of components to use in their ARCP CPU:

- a 6-port Register File.
- separate Data and Instruction Memories with a bandwidth of 64 bits/clock cycle.
- two 32-bits ALUs.
- any number of multiplexers, flip-flops and decoders.

The ARCP CPU control logic is to be implemented in synthesisable or at least “almost” synthesizable HDL; for this purpose students are provided with guidelines on producing synthesizable Verilog Code. We ask students to identify all possible data and control hazards and to try and reduce them using data forwarding. Whenever forwarding cannot eliminate a hazard, their control logic should insert wait states, *i.e.* “bubbles” in the pipeline. As students have only 3 weeks to implement this stage of the course, to save time they are provided with a schematic of a reference datapath. Figures 5, 7, 6 show the schematics for stages 1, 2 and stages 3,4 and 5 of the ARCP pipeline respectively.

The ARCP datapath schematics shown include some of the required control signals to give students a hint of how to implement the control logic for the pipeline stages and for forwarding data. During the past few years of running this course we have experimented with these schematics, in some years showing some of the control signals in these schematics, whereas in other years we did not. We found that students took about 50% more time to complete the implementation when they were not given any of the control signals in these schematics.

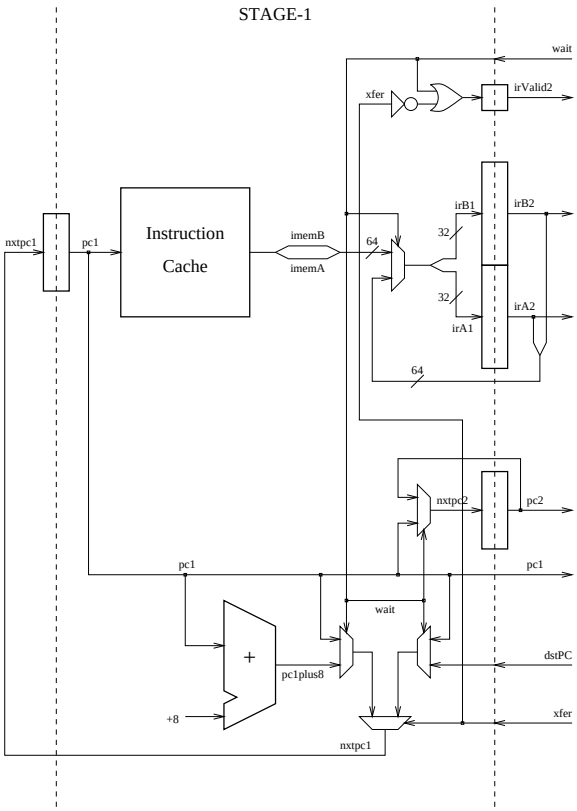

Figure 5: Stage 1 of the pipeline

After completing their implementation students must verify the correctness of their low-level implementation by using their architectural simulator developed in the “Simulate and Experiment” stages as a “Golden Model” and comparing the operation of the two on the same program code. In this way, students acquire another necessary skill for hardware design, verification against high-level models.

To make good use of their implementation and to make them realize that detailed hardware models can be used whenever detailed results are required, we ask them to calculate the speedup of this architecture compared to a reference non-pipelined architecture we provide them.

Finally, the students are to provide a report on how different architectural approaches affect the hardware implementation. To help them realise the complexity of the task we suggest that they alter their implemented datapath so as to implement their proposed optimisations, which they already implemented in their architectural simulation. By doing this, it is easily made obvious how complex it can be to implement a new optimisation which might take almost no time to incorporated in the architectural simulation.

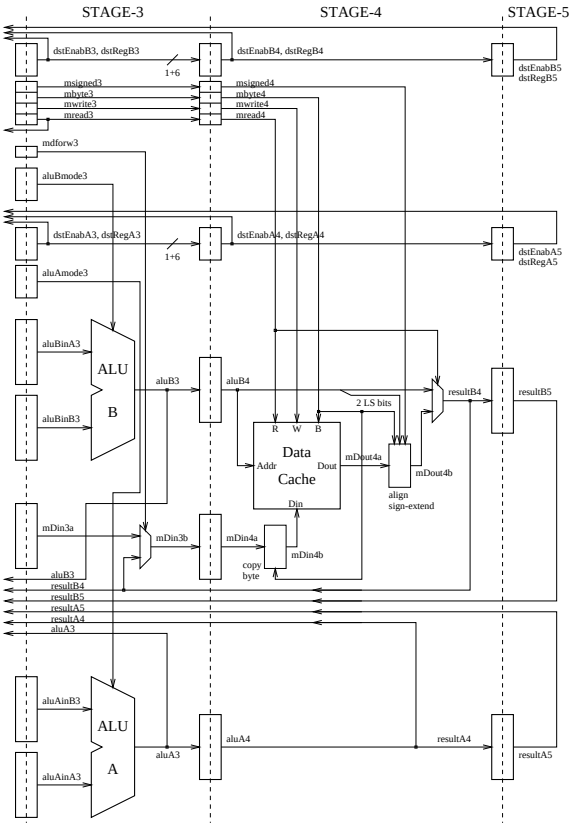

Figure 6: Stages 3, 4 and 5 of the pipeline

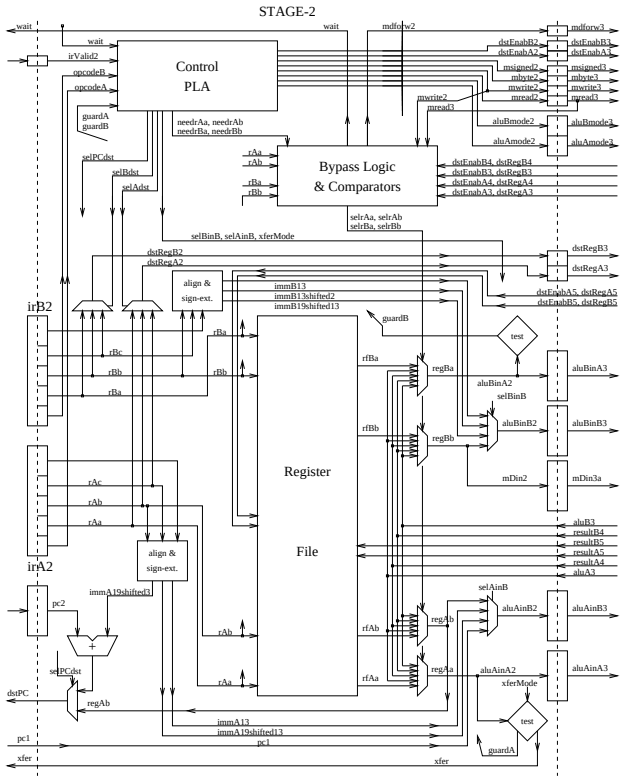

Figure 7: Stage 2 of the pipeline

6 Conclusion

In this paper an integrated approach for teaching Computer Architecture was presented, which is currently used at our University, and has been found to be very effective. Its main advantages are the following:

1. It increases the interest of the students in Computer Architecture and hardware in general. There was a significant increase in the number of students concentrating on Hardware after we have adopted this approach either by taken their undergraduate thesis on a hardware subject or enroll on a hardware or semi-hardware oriented postgraduate program.

2. It gives the student a thoroughly comprehension of the main subjects of Computer Architecture

3. It enhances their performance in the exams which is probably due to the fact that they get a lot of hands-on experience on every aspect of Computer Architecture.

4. It provides them with skills that are very useful when designing hardware and not only when investigating the architecture of a system.

This approach is, we believe, ideal for a course that is taken by students that might want to focus on hardware, or have already made such a decision and they would like to get a first idea of how a system is initially designed, then simulated and finally built and tested. Its main disadvantage is, we believe, that it relatively increases the work needed for the course and might not be that appropriate for cases where just an introduction to Computer Architecture is needed (maybe because there are a great number of more specialized hardware design courses in the syllabus).

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[2] P. S. Coe, F. W. Howell, R. N. Ibbett, and L. M. Williams, “A Hierarchical Computer Architecture Design and Simulation Environment,” *ACM Transactions on Modelling and Computer Simulation*, vol. 8, Oct. 1998.