# *ESCAPE*: Environment for the Simulation of Computer Architectures for the Purpose of Education

Jan Van Campenhout  Peter Verplaetse *  Henk Neefs
Department of Electronics and Information Systems
University of Ghent, Belgium
{jvc|pvrplaet|neefs}@elis.rug.ac.be

## Abstract

*We have developed ESCAPE, an easy-to-use, highly interactive portable PC-based simulation environment aimed at the support of computer architecture education. The environment can simulate both a microprogrammed architecture and a pipelined architecture with single pipeline. Both architectures are custom-made, with a certain amount of configurability. Other tools, such as a memory monitor, assembler/disassembler and analysis tools, such as on-the-fly generation of pipeline activity and usage diagrams, are integrated with the environment.*
*Based upon our limited experience with the material so far, we can state that the results are excellent. Students invariantly respond very positively, and the evaluations indicate a far deeper understanding than was previously attainable by using only the traditional textbook-and-paper-problems approach.*

## 1 Introduction

The complexity of computer architectures has increased significantly over the past decades. It is our experience that many students fail to understand even the basic concepts, such as microprogrammed architectures or pipelined execution with simple pipeline, making it impossible to fully understand the operation of a contemporary processor—typically superscalar with out-of-order execution, branch prediction and possible speculative execution.

One way to clarify these simple concepts is by the use of simulation tools. There are many simulators for computer architectures available, but most of them are unsuitable for inexperienced users. Most simulators are designed with accurate modeling as a main feature, as a result these simulators have a complexity similar to today's processors.

This paper describes *ESCAPE*, a computer architecture simulation environment used extensively in an undergraduate level course on computer architecture at the University of Ghent. This environment was created to increase the effectiveness of the course, i.e. to increase the level of insight in and understanding of computer architectures achieved by the students.

The paper is organized as follows. We first situate our work in the context of computer architecture education. After identifying problems and defining a solution in terms of requirements for the simulation environment, we briefly describe the architectural aspects of both machines. Details of the simulation software and possible uses for the environment are described in the next sections. We then briefly evaluate the preliminary results obtained, and conclude with an outlook on future work.

## 2 Computer architecture education: goals and means

The course in which the work presented here is situated, is a second course on computer architecture in the computer science and computer engineering undergraduate curriculum of the University of Ghent. The preceding course covers instruction-level aspects, such as addressing modes, assembly language, etc. The second course focuses on the micro-architectural aspects of contemporary architectures, with significant emphasis on their evolution. The course is a one-semester course featuring 12 weekly 75-minute lectures, and three lab sessions each initiating an independent homework assignment to be completed by the students.

The course starts with a discussion of the internal control of the instruction interpretation or execution pro-

*Research Assistant of the Fund for Scientific Research – Flanders (Belgium)(F.W.O.)

cess; it covers both microprogramming and pipelined execution. This is an ideal basis for more advanced topics such as multiple and superscalar pipelines, also treated in the course. Next, we address the issues of the memory hierarchy, covering topics such as caches, virtual memory, cache coherency, register optimization, etc.

Although the initial approach taken in the course is rather qualitative, we clearly aim at a more quantitative treatment of the material, along the style advocated by some recent excellent textbooks [1][2][3]. Obviously, hands-on experience is crucial to achieve the latter.

Based on the experience of many years of teaching computer architecture and related courses, we feel that the lack of a thorough understanding of basic concepts makes it hard for students to fully grasp more complex topics, let alone to achieve quantitative insights.

# 3 Achieving a more thorough understanding

To make such understanding easier to achieve by the student, we decided to develop a simulation environment addressing some of the architectural issues treated in the course. The main goal of this environment is to present the students with an easy-to-understand custom-made architecture that allows them to become familiar with the basic concepts of computer architecture, without being overwhelmed by the complexity of realistic microprocessor architectures.

The application area is twofold:

1. to allow its interactive use during lectures for the demonstration of basic concepts;
2. to form a foundation for homework assignments and projects that will allow students to obtain valuable insight in more complex matters, without first having to study many manual pages.

To be successful in meeting these goals, a number of requirements should be met.

First, the environment should support architectures for both microprogrammed and pipelined execution. Even though many recent textbooks mainly focus on pipelined execution, we feel that in-depth coverage of microprogrammed control is essential because it can provide valuable insight on the internal operation of the processor. Furthermore, we deem the capability to put contemporary evolutions into their proper historical setting essential in a university-level course. And, lastly, by having identical instruction sets for both architectures, the sometimes subtle distinction between instruction level and micro-architectural aspects is ideally exemplified.

Second, the architectures provided should be custom-made with a certain amount of configurability. The size of the register file, the number of temporary registers (for microprogrammed control), memory access time and ALU functionality should not be fixed. This may require a flexible instruction encoding. Other aspects that can influence the behavior are cache emulation and pipeline specific features such as enabling or disabling the delay slot or register forwarding.

Third, an easy-to-use, highly interactive interface is essential. Simulation of the architecture should be possible on a cycle-per-cycle basis, with an option for rewinding the clock. The content of the registers should be visible while simulating, and the memory content and cache behavior should be represented in a clear and concise way. Visual representation of multiplexers and bus behavior is necessary to expose the forwarding and stalling mechanism. On-the-fly trace generation and pipeline activity diagrams will further clarify the way the architecture operates.

Fourth, it should also serve to collect quantitative data, in addition to provide valuable qualitative insight. This requires single click multiple cycle simulations with breakpoints. Simulation speed is an important factor here.

In the following sections, we shall present *ESCAPE*. A preliminary version of this tool is already available; the full-featured version is still under development at the time of writing, but should be completed shortly.

# 4 Architectural details

The *ESCAPE* environment consists of two simulators. The instruction set architectures of both machines are essentially identical, even though the microarchitectural aspects are very different (a microprogrammed processor versus a pipelined processor with simple pipeline).

The instruction set architecture is inspired by Hennessy and Patterson's DLX [1]. The three distinguished types of instructions (I-type, R-type and J-type) are shown in figure 1. Contrary to the DLX architecture the size of the bitfields is not fixed, but depends on the maximum number of instructions and the size of the register file. All instructions have a 32-bit encoding, hence the length of the immediate fields ($n_{i1}$ and $n_{i2}$) can be derived from the bitfield sizes of the opcode and

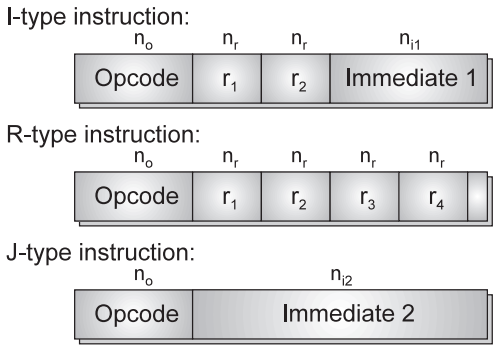


Figure 1: instruction encoding.

| Operation | Result | Note |
|---|---|---|
| $+$ | S1 + S2 | add |
| $-$ | S1 − S2 | subtract |
| $-_r$ | S2 − S1 | reverse subtract |
| & | S1 & S2 | bitwise and |
| \| | S1 \| S2 | bitwise or |
| ^ | S1 ^ S2 | bitwise exclusive or |
| << | S1 << S2 | shift left |
| >> | S1 >> S2 | shift right |
| $>>_a$ | S1 $>>_a$ S2 | shift right arithmetic |
| S1 | S1 | pass S1 |
| S2 | S2 | pass S2 |

Table 1: basic ALU operations.

formals ($n_o$ and $n_r$):

$$n_{i1} = 32 - n_o - 2n_r$$
$$n_{i2} = 32 - n_o$$

R-type instructions can have up to 6 formals (assuming $n_r$ is sufficiently small). This can be useful for implementing more advanced operations in the microprogrammed architecture, a popular homework assignment.

## 4.1 Microprogrammed architecture

The architecture consists of a control unit and a datapath (figure 2). The datapath consists of a register

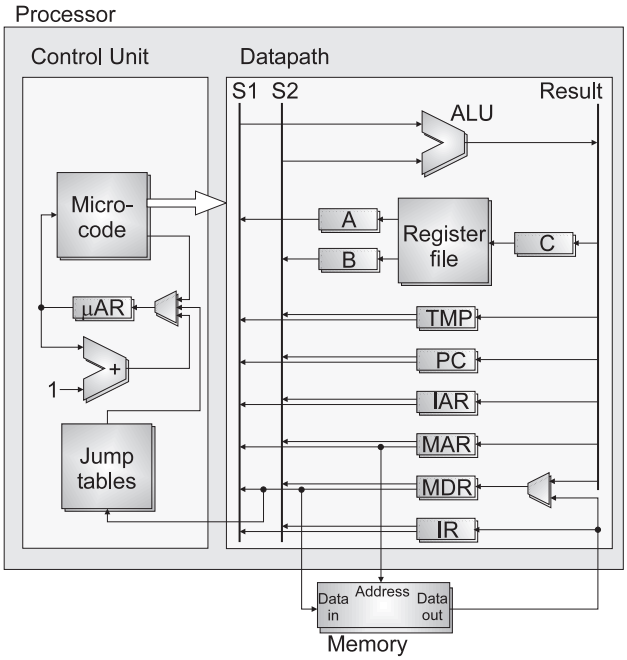


Figure 2: microcoded architecture.

file, two read registers (`A`, `B`) and a write register (`C`), a memory interface with address (`MAR`), data (`MDR`) and instruction (`IR`) registers, a number of extra registers (typically `IAR`, `PC` and a few temporary registers) and an ALU. The different parts are connected by two input buses (`S1` and `S2`) and a result bus. The ALU can perform a number of basic operations in a single cycle (table 1). A built-in comparator does zero and sign detection on the result.

The memory interface can load and store bytes, halfwords (16 bit) or words (32 bit), with adjustable access time. Both instructions and data are stored in the same memory (von Neumann architecture).

The control unit is microcoded. The microcode address is kept in a special register (`uAR`). During each cycle `uAR` is either incremented or replaced with a new value (i.e. a jump to a new microinstruction). Typical jump conditions are: memory busy, ALU output zero, ALU output negative and interrupt pending. The jump address is either in the microcode, or read from a jump table (indexed by the opcode field in `IR`). The latter is useful for instruction decoding. The number of jump tables is adjustable from 1 to 4.

The standard register file functionality ((`A`,`B`)←($r_1$,$r_2$), $r_1$←`C` and $r_3$←`C`) can be extended with `A`←$r_i$, `B`←$r_j$ and $r_k$←`C` operations.

The microprogrammed architecture (both control unit and datapath) have deliberately been kept simple. There is no microcode pipelining register, it only has basic single-cycled operations, and virtually no microcoding tricks have been used [4]. The datapath is very lean, and could be improved on several counts. This deliberate simplicity leaves ample room for the students to suggest improvements for the architecture.

## 4.2 Pipelined architecture

Both the control unit and the datapath are pipelined into the five traditional stages (figure 3): IF (instruc-

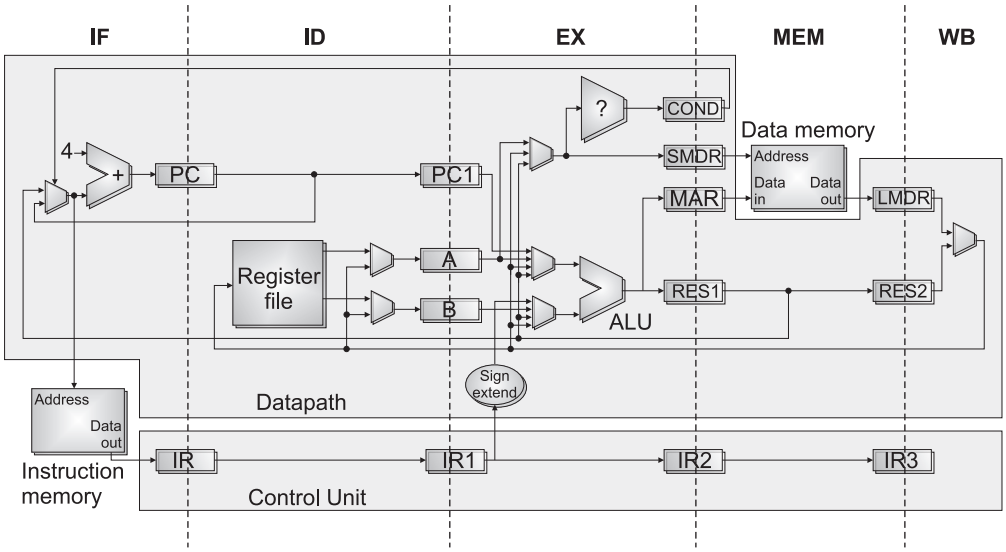


Figure 3: pipelined architecture.

tion fetch), ID (instruction decode), EX (execute and effective address calculation), MEM (memory) and WB (write back). Because there are at least three cycles between reading the register file and write back, a forwarding mechanism is implemented to prevent the pipe from unnecessary stalling. The register file is read in the ID stage, but written during the WB stage. Write through is explicit by the use of multiplexers.

The EX stage consists of an ALU and a comparator. The ALU can perform the same operations as the one for the microprogrammed architecture, and additional multiply and divide operations. These two operations can have a latency of more than one cycle. During the execution of a branch the comparator evaluates the branch condition while the ALU calculates the effective address. Depending on the settings of the simulator the two instructions following the branch can be executed (i.e., a *double* delay slot), nullified (no delay slot), or only the instruction in the IF stage is nullified (single delay slot).

There are two separate memory interfaces: one for instructions and one for data (Harvard architecture). Access to the data memory occurs during the MEM stage. The data memory access time is adjustable. The `SMDR` and `MAR` registers can be frozen to mitigate stalling.

# 5 Features of the simulation environment

The *ESCAPE* environment has been implemented in Borland®'s Delphi®. Because the code is compatible with Delphi 1.0, we have both 16- and 32-bit versions, which makes the application run on every Windows®based operating system. A copy of the latest version, further documentation as well as sample exercises can be downloaded from the web:

*http://www.elis.rug.ac.be/ pvrplaet/escape.html*

After starting the simulator an architecture specific form appears. The layout of this form is based on the structural representation of the architecture (figures 2 and 3). Having all the key elements of the architecture on one single form makes it possible to understand the processor operation without having to swap back and forth between windows. Screenshots of these forms are shown in figures 4 and 5. The forms have been designed to be displayable with a 640 × 480 resolution, for classroom use.

A few other forms exist. The memory can be viewed and/or edited in two ways. The data form acts as a memory monitor/editor that allows you to examine or edit the memory content in groups of bytes, halfwords or words, and different number bases (unsigned hexadecimal and unsigned or signed decimal). The code form behaves as an assembler/disassembler that allows easy writing of assembly code.

For the microprogrammed architecture a form similar to the code form exists to edit the microinstructions and jump tables. This is the so-called microcode form. Another important form is the configuration form, that allows one to configure the two architectures.

Key features of the simulation environment are:

- easy-to-use interface;

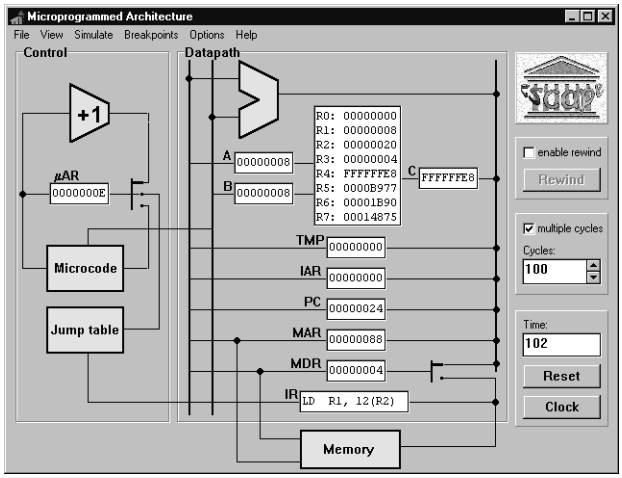


Figure 4: screenshot of the microprogrammed architecture.

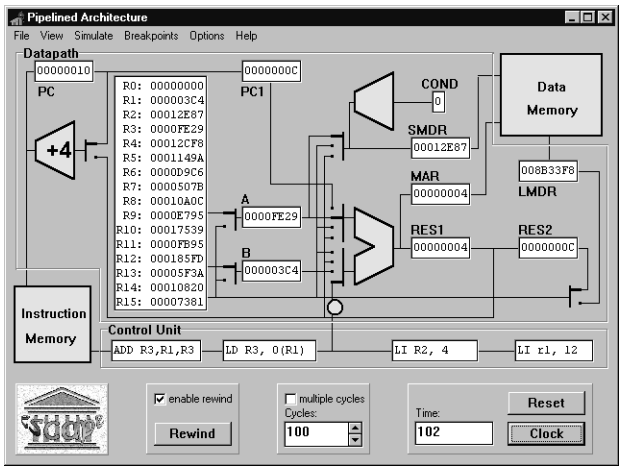


Figure 5: screenshot of the pipelined architecture.

- partially configurable, easy-to-understand custom-made architectures;
- cycle-per-cycle simulation, or multi-cycle simulation with breakpoints;
- clock rewind, can be disabled to increase simulation speed;
- memory monitor and assembler/disassembler;
- microcode editor;
- on-the-fly trace generation;
- on-the-fly generation of pipeline activity and pipeline usage diagrams;
- all files are in ASCII format, which allows them to be altered with external editors.

A pipeline *activity* diagram plots for each instruction the current pipeline stage versus time, as shown in figure 6.

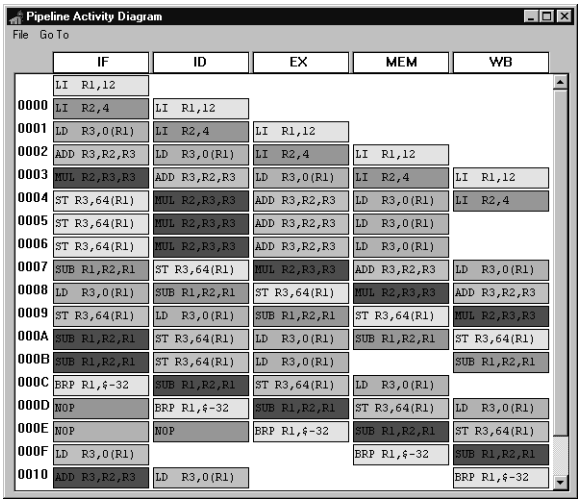


Figure 6: pipeline activity diagram.

A pipeline *usage* diagram plots for each pipeline stage the current instruction (if any) versus time, as shown in figure 7.

# 6 Possible uses of *ESCAPE*

Studying the microprogrammed architecture will allow the student to

- become acquainted with the basic synchronous operation at the register transfer level of a datapath and its microprogrammed control;

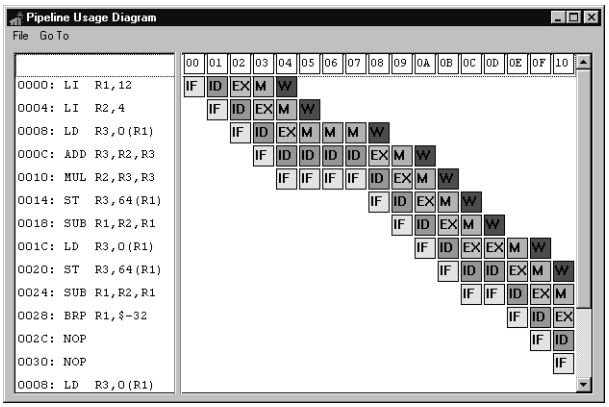


Figure 7: pipeline usage diagram.

- learn the basics of microprogramming;
- learn about microprogram optimization techniques, which is quite relevant with respect to contemporary VLIW architectures;
- obtain quantitative data on speed and code size, the influence of memory speed on microcode efficiency, etc.

By using *ESCAPE* for simulating the pipelined architecture, the student can

- become acquainted with pipelined operation, hazards and their solutions;
- experiment with traditional code optimization techniques such as code motion, register renaming, loop unfolding, software pipelining, etc;
- perform small-scale quantitative measurements on benchmark programs, which will result in better insight on the power and main limitations of pipelined execution.

# 7 Results

As stated before, the simulation environment is still under development, therefore it is premature to draw final conclusions as to its effectiveness. However, a provisional version of the *ESCAPE* environment has been used for two different offerings of the course: once in a post-graduate version, addressed to students that graduated 5–10 years ago (class size: 15), and once in the regular engineering curriculum of the University of Ghent (class size: 120).

At the time of writing, the final exams had not yet taken place, but from the experience gathered from homework assignments, significant improvement has been observed both in the understanding of the architectural issues, as in the ability to effectively deploy such architectures. The comparison is based on the performance on similar assignments during previous years.

# 8 Conclusions and future work

In this paper, an interactive graphical simulation environment has been presented, aimed at the support of computer architecture education. The environment allows simulation of simple custom-made microprogrammed or pipelined architectures.

First experiments have revealed significant improvements of the teaching effectiveness. Students invariantly respond very positively, and the evaluations indicate a far deeper understanding than was previously attainable by using only the traditional textbook-and-paper-problems approach.

At this point the environment simulates either a microprogrammed or a pipelined machine with limited configurability. Several extensions and additions are being formulated. We plan to extend the simulation model with caches (which will result in variable instruction memory access), out-of-order write back, multiple execution units, and possibly superscalar pipelines with scoreboarding and branch prediction. This will allow the use of a single environment for teaching a wide range, from basic concepts to more advanced topics in contemporary computer architecture.

# References

[1] Hennessy, J.L, Patterson, D.A. (1990, 1996), *"Computer Architecture A Quantitative Approach"*, Morgan Kaufmann Publishers: San Mateo, CA, USA.

[2] Patterson, D.A, Hennesy, J.L. (1998), *"Computer Organization & Design. The Hardware/Software Interface"*, Morgan Kaufmann Publishers: San Francisco, CA, USA.

[3] Flynn, M.J. (1995), *"Computer Architecture Pipelined and parallel processor design"*, Jones and Barlett Publishers: Boston, MA, USA.

[4] Rauscher, T.G., Adams, P.M., *"Microprogramming: A Tutorial and Survey of Recent Developments"*, IEEE transactions on Computers, Vol. C-29, No. 1, pp 2–20.

[5] Sima, D., Fountain, T., and Kacsuk, P. (1997), *"Advanced Computer Architectures A Design Space Approach"*, Allison Wesley Longman: Harlow, England.

[6] Heuring, V.P., and Jordan, H.F. (1997), *"Computer Systems Design and Architecture"*, Allison Wesley Longman: Menlo Park, CA, USA.