Using ESCAPE: Environment for the Simulation of Computer Architectures for the Purpose of Education

FREDERIK HABILS PETER VERPLAETSE JAN VAN CAMPENHOUT
Department of Electronics and Information Systems
University of Ghent
Sint-Pietersnieuwstraat 41, B-9000 Ghent
BELGIUM

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, like 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.

In this paper we present the simulator as well as a practical exercise that we prepared for the students of an undergraduate level course on computer architecture at the University of Ghent. Based upon our limited experience with the simulator so far, we can state that the results are excellent. Students 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. *IMACS/IEEE CSCC'99 Proceedings*, Pages:3691-3697

Key-Words: - computer architecture, education, visual simulation, microprogramming, pipelining

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 branch prediction and speculative, out-of-order 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 in fact designed with accurate modeling as a main feature; as a result, these simulators have a complexity similar to today's processors.

At the University of Ghent, we are extensively using *ESCAPE*[1] in an undergraduate level course on computer architecture [1]. *ESCAPE* is an interactive computer architecture simulation environment that 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. We will present an actual assignment on microprogramming that the students have to solve.

The paper is organized as follows. We first describe *ESCAPE*, starting from a situation of 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 the simulated processor models. We then describe an actual assignment to be solved in *ESCAPE*, while also showing some more details of the simulation software and possible uses for the environment in general. We then briefly evaluate the preliminary results obtained, and conclude with an outlook on future work.

2 Computer architecture education

2.1 Situating the course

The course in which the work presented here is situated, is a second course on computer architecture in the computer science and computer engineering

[1]Environment for the Simulation of Computer Architectures for the Purpose of Education

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 main topics of the course are the internal control of the instruction execution process (discussing microprogramming and pipelined execution) and the many issues of the memory hierarchy (caches, virtual memory, 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 [2,3,4]. 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.

2.2 Towards a more thorough understanding

To make such understanding easier to achieve by the students, 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.

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

- *Support for both microprogrammed and pipelined architectures.*
  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.
- *Custom-made yet configurable architectures.* Register file size, the number of temporary registers (for microprogrammed control), memory access time, ALU functionality, forwarding, delay slot usage, etc. should not be fixed.
- *Easy-to-use, highly interactive interface.* Simulation of the architecture should be possible on a cycle-per-cycle basis, with an option for rewinding the clock. The state of the architecture (registers, memory, multiplexers, bus behavior etc.) must be visualized. On-the-fly trace generation and pipeline activity diagrams will further clarify the way the architecture operates.
- *Supporting the collection of quantitative data,* by enabling single click multiple cycle simulations with breakpoints.

In the following sections, we shall present *ESCAPE*. A working version of this tool is already available, though we hope to add some functionality over time.

3 Introducing *ESCAPE*.

The *ESCAPE* environment consists of two simulators. The instruction set architecture (ISA) is essentially identical for both machines, even though the micro-architectural aspects are very different (a microprogrammed processor versus a pipelined processor with simple pipeline).

The (extendible) ISA is inspired by Hennessy and Patterson's DLX [2,3]. 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 formals (n_o and n_r). 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.

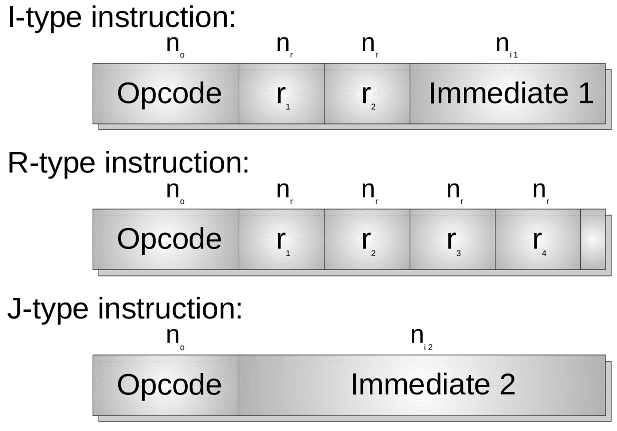


Figure 1: instruction encoding

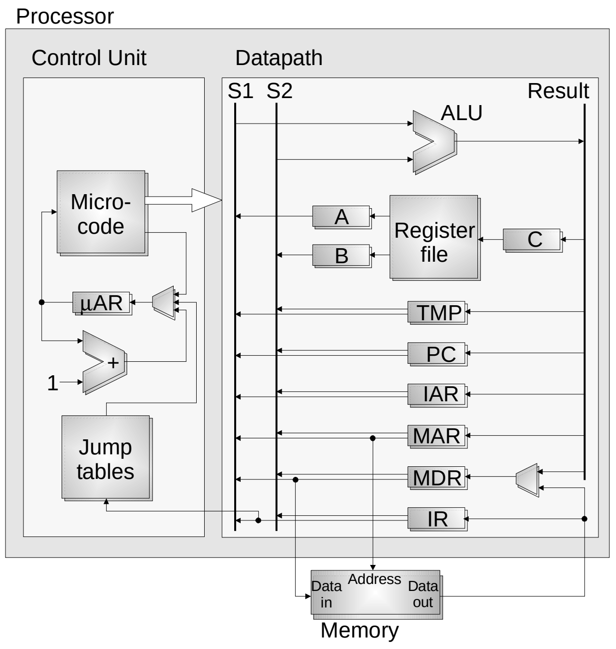


Figure 2: microcoded architecture

3.1 Microprogrammed Architecture

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

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 xor
<<	S1 << S2	shift left
>>	S1 >> S2	shift right logical
$>>_a$	S1 $>>_a$ S2	shift right arithmetic
S1	S1	pass S1
S2	S2	pass S2

Table 1: basic ALU operations

The control unit is microcoded. The microcode address is kept in a special register (`µAR`). During each cycle `µAR` 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. It is easy to add more specific instructions to the ISA by providing the necessary microprograms for the instructions.

The microprogrammed architecture (both control unit and datapath) has 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 [5]. 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.

3.2 Pipelined Architecture

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

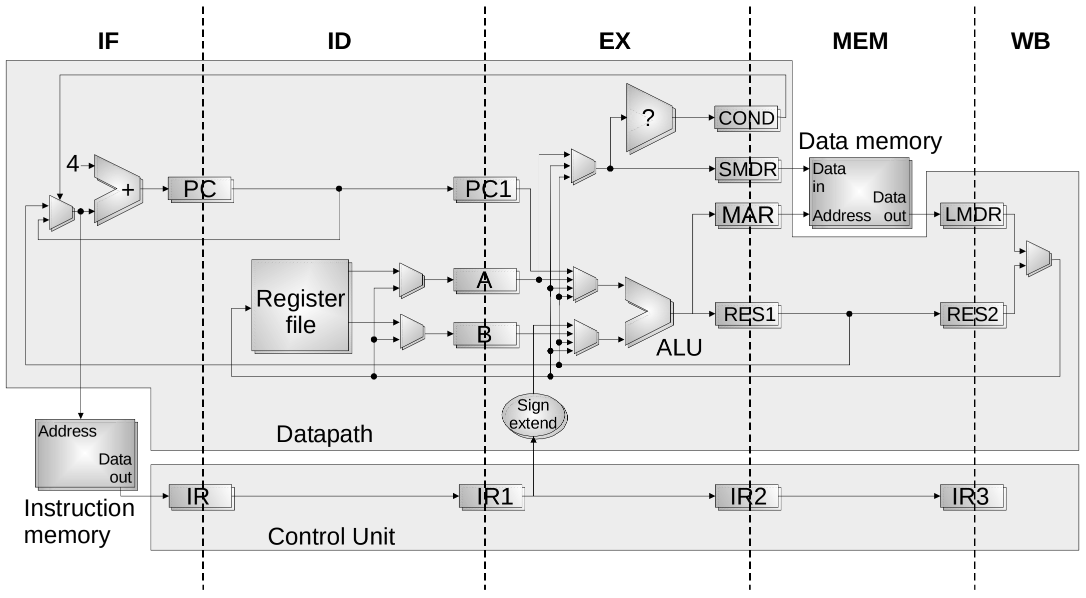


Figure 3: pipelined architecture

3.3 Implementation details

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/escape`.

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.

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

4 A microprogramming assignment

We have prepared a number of assignments that can be used to familiarize the students with the different aspects of microprogrammed and pipelined execution. We are presenting here one such exercise on microprogramming issues.

The basis of the exercise is a high-level algorithm, in this case a heap sort implementation by Williams [6], also documented by H. Stone [7]. The students are given the Pascal-like pseudo-code shown in figure 8.

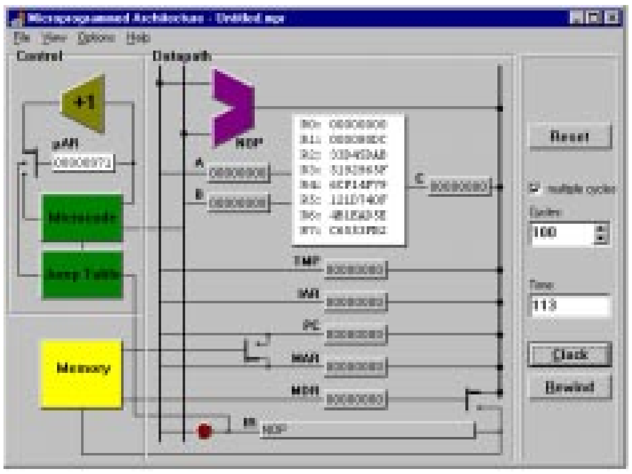


Figure 4: screenshot of the microprogrammed architecture form

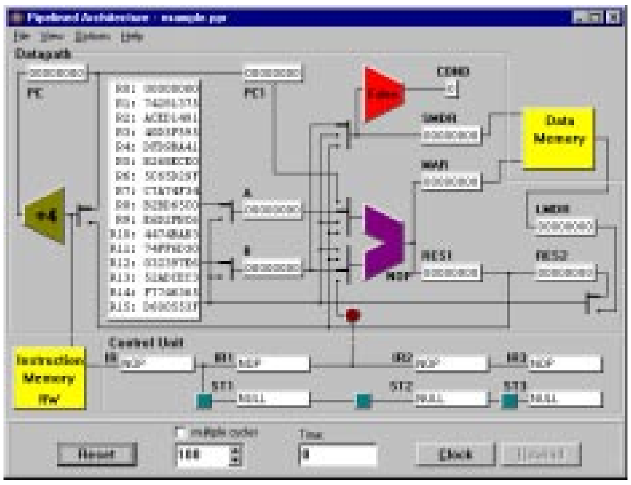
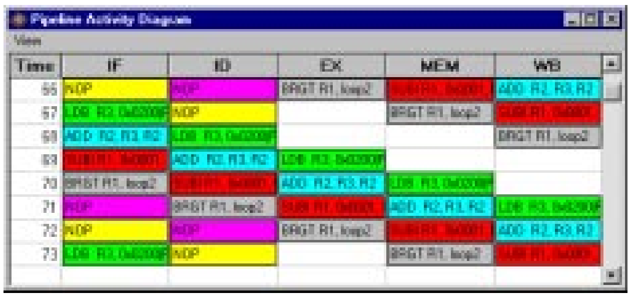
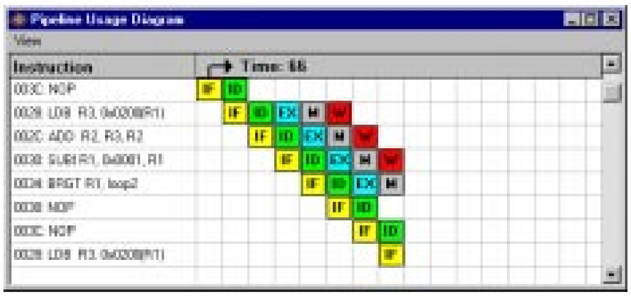


Figure 5: screenshot of the pipelined architecture form



Time	IF	ID	EX	MEM	WB
66	NOP	NOP	BRGT R1, loop2	SUBI R1, 0x0001	ADD R2, R3, R2
67	LDB R3, 0x0200(R	NOP		BRGT R1, loop2	SUBI R1, 0x0001
68	ADD R2, R3, R2	LDB R3, 0x0200(R			BRGT R1, loop2
69	SUBI R1, 0x0001	ADD R2, R3, R2	LDB R3, 0x0200(R		
70	BRGT R1, loop2	SUBI R1, 0x0001	ADD R2, R3, R2	LDB R3, 0x0200(R	
71	NOP	BRGT R1, loop2	SUBI R1, 0x0001	ADD R2, R3, R2	LDB R3, 0x0200(R
72	NOP	NOP	BRGT R1, loop2	SUBI R1, 0x0001	ADD R2, R3, R2
73	LDB R3, 0x0200(R	NOP		BRGT R1, loop2	SUBI R1, 0x0001

Figure 6: screenshot of the pipeline activity diagram



Instruction	Time: 66							
003C NOP	IF	ID						
0028 LDB R3, 0x0200(R1)		IF	ID	EX	M	W		
002C ADD R2, R3, R2			IF	ID	EX	M	W	
0030 SUBI R1, 0x0001, R1				IF	ID	EX	M	W
0034 BRGT R1, loop2					IF	ID	EX	M
0038 NOP						IF	ID	
003C NOP							IF	ID
0028 LDB R3, 0x0200(R1)								IF

Figure 7: screenshot of the pipeline usage diagram

4.1 Part One

In the first part of the assignment, the students have to study an incomplete assembler version of the algorithm that we supplied them with. The assembler code uses function inlining for the functions `intree()` and `outtree()`, as they are called only once, and also to simplify the exercise. The students are asked to complete the code, using similar optimization techniques as were already demonstrated in the existing code, i.e. strength reduction, register scheduling, induction variable elimination, etc. For this exercise, they are only supposed to use the standard, simple set of assembler instructions, given in table 2. The missing code represents the call to `outtree()`.

The students are asked to test their code with the *ESCAPE* simulator, on a set of 100 randomly ordered numbers, and measure the execution time for different memory access times (1–9 cycles), using the breakpoint facility of *ESCAPE*.

This part of the assignment, while not delving deeply in the matters of microprogramming, familiarizes the students with the algorithm and the environment of the simulations. Having them create part of the assembler program themselves, should create a greater variation in the rest of the solutions later on as well, thereby reducing the possibility of simply copying other peoples solution.

4.2 Part Two

Secondly, we 'introduce' the programmable microcoded processor, and ask the students to write microcode for the following instructions:

1. **`swapld R1, R2, R3`**
   This instruction swaps `A[1]` and `A[i]` and saves the new `A[1]` in `R3`, while `R1` equals `A` and `R2` equals `4*i`.
2. **`intree R1, R2`**
   This instruction executes subroutine `intree(lim: integer)`, where `R1` equals `A` and `R2` equals `4*lim`.
3. **`outtree R1, R2, R3`**
   This instruction executes subroutine `outtree(lim: integer)`, where `R1` equals `A`, `R2` equals `4*lim`, and `R3` equals `A[1]`.

The students are asked to write microcode that is as fast as possible (first priority), yet also as small as possible (second priority). We have created solutions using 11, 17 and 22 lines of microcode, respectively. We also ask them to use no more temporary registers than strictly necessary (third priority).

```
procedure treesort(A: array of integer,
                   N: integer)
// array A is declared A[1:N]
begin

  // Phase 1: A[lim] added to heap A[1:lim-1]
  procedure intree(lim: integer)
  begin
    var in, i: integer;
    in := A[lim];
    i:= lim;
  scan:
    if (i > 1) then
    begin
      j:= i/2;
      if (in > A[j]) then
      begin
        A[i] := A[j];
        i:=j;
        goto scan;
      end;
    end;
    A[i] := in;
  end procedure intree;

  // Phase 2: A[1] pushed down through 'heap'
  procedure outtree(lim: integer)
  begin
    var i, j, copy: integer;

    i := 1;
    copy := A[1];
  loop:
    j := 2*i;
    if (j <= lim) then
    begin
      if (j+1 <= lim) then
      begin
        if (A[j+1] > A[j]) then j := j+1;
      end;
      if (A[j] > copy) then
      begin
        A[i] := A[j];
        i := j;
        goto loop;
      end;
    end;
    A[i] := copy;
  end procedure outtree;

  // main body
  var i, temp: integer;

  for i:=2 to N do intree(i);
  for i:=N downto 2 do
  begin
    temp := A[1];
    A[1] := A[i];
    A[i] := temp;
    // largest heap element put at end array
    outtree(i-1);
  end;

end procedure treesort;
```

Figure 8: heapsort algorithm

Instruction	Result	
`NOP`	-	
`ADD R1,R2,R3`	`R3 := R1+R2`	(*i*)
`SUB R1,R2,R3`	`R3 := R1-R2`	(*i*)
`MUL R1,R2,R3`	`R3 := R1*R2`	(*i*)
`DIV R1,R2,R3`	`R3 := R1/R2`	(*i*)
`AND R1,R2,R3`	`R3 := R1 AND R2`	(*i*)
`OR  R1,R2,R3`	`R3 := R1 OR  R2`	(*i*)
`XOR R1,R2,R3`	`R3 := R1 XOR R2`	(*i*)
`SLL R1,R2,R3`	`R3 := R1 << R2`	(*i*)
`SRL R1,R2,R3`	`R3 := R1 >> R2`	(*i*)
`SRA R1,R2,R3`	R3 := R1 $>>_a$ R2	(*i*)
`LDW R1,off(R2)`	R1 := mem_{32}[R2+off]	
`STW R1,off(R2)`	mem_{32}[R2+off] := R1	
`BRZ  R1,label`	`if(R1==0) PC:=adr(label)`	
`BRNZ R1,label`	`if(R1!=0) PC:=adr(label)`	
`BRGT R1,label`	`if(R1> 0) PC:=adr(label)`	
`BRGE R1,label`	`if(R1>=0) PC:=adr(label)`	
`BRLT R1,label`	`if(R1< 0) PC:=adr(label)`	
`BRLE R1,label`	`if(R1<=0) PC:=adr(label)`	

Table 2: simple assembler instructions (the instructions with (*i*) also have a variant where `R2` is an immediate, e.g. `ADDI R1,imm,R3`

The students receive the assembler code that implements the algorithm using these extra instructions. As before, they are asked to test their code in *ESCAPE* and measure the execution time for different memory access times (1–9 cycles) when sorting the same sequence of 100 integers. Another question that needs to be answered is whether they see specific architectural enhancements that would make even faster microprograms possible for the three instructions above. Given the simplicity of the architecture at hand, there are at least some obvious suggestions, like increasing the number of memory address and data registers.

4.3 Part Three

In this last part, we ask the students to compare the two implementations of the heapsort algorithm. Specifically, we make the assumption that the processor of part two ($\mu Proc_2$) can only be driven at half the clock speed of the processor of part one ($\mu Proc_1$), due to its more complex nature (programmable control store, extra temporary registers, etc.). We then ask to compare the execution speed of the algorithm on $\mu Proc_1$ and $\mu Proc_2$ for memory access times of 1, 2 and 3 cycles in $\mu Proc_2$, and assuming that both processors use similar memory chips. The trick here of course, is to realize that, when $\mu Proc_1$ is driven at double clock speeds, the memory access time in cycles should also be double compared to μ$Proc_2$ (taking the simplest approach for lack of exact data on the memory). So, the answer should compare the execution time of μ$Proc_1$ using 2 cycles for memory access with the execution time of μ$Proc_2$ using only 1 cycle for memory access, etc.

A final question is whether the students can deduce from their measurements, how many times μ$Proc_1$ should be clocked faster than μ$Proc_2$ (with memory access time of 1 cycle) to make both implementation run at equal speed. In fact, given our implementations and measurements, this is not possible (we can only indicate it must be more than 9 times faster), but perhaps, given other implementations in part one and part two, some student may have totally different, yet correct answers to this question.

5 Results

This is the second year that we are using *ESCAPE* in this course, though last year we worked with a preliminary version of the environment. Obviously, given the limited experience we had using *ESCAPE* as an educational tool, any conclusions as to its effectiveness remain premature. While we concluded from last year's experience with the homework assignments (in a class of 120 students), that significant improvement was observed in the understanding of architectural issues, it will be hard to quantify this improvement without a larger scaled and controlled test. Probably the best indication of the usefulness of this tool is the gratefulness with which the students make use of it, to better understand the material that they may find dully presented in the textbooks.

6 Conclusion 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. We also presented a specific homework assignment on microprogramming that was presented to the students during this semester.

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.

Acknowledgements

Frederik Habils is supported by a grant from the University of Ghent. Henk Neefs is supported by project GOA 12050895. Jan Van Campenhout is a full professor at the University of Ghent.

References:

[1] Jan Van Campenhout, Peter Verplaetse and Henk Neefs. ESCAPE: Environment for the Simulation of Computer Architectures for the Purpose of Education. *Workshop on Computer Architecture Education*, Barcelona. 1998.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers. 1990 & 1996.

[3] David A. Patterson and John L. Hennessy. *Computer Organisation and Design; The Hardware/Software Interface*. Morgan Kaufmann Publishers. 1994 & 1998.

[4] Michael J. Flynn. *Computer Architecture: Pipelined and parallel processor design*. Jones and Bartlett Publishers. 1995.

[5] T.G. Rauscher and P.M. Adams. Microprogramming: A Tutorial and Survey of Recent Developments. *IEEE Transactions on Computers*. Vol. C-29, No. 1, 1980, pp. 2-20.

[6] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*. Vol. 7, No. 6, June 1964, pp. 347-348.

[7] Harold S. Stone. *Introduction to Computer Organization and Data Structures*. McGraw-Hill. 1972.