# MipsIt—A Simulation and Development Environment Using Animation for Computer Architecture Education

Mats Brorsson

Department of Microelectronics and Information Technology,
KTH, Royal Institute of Technology
Electrum 229, SE-164 40 Kista, Sweden
email: Mats.Brorsson@imit.kth.se

**Abstract**

*Computer animation is a tool which nowadays is used in more and more fields. In this paper we describe the use of computer animation to support the learning of computer organization itself.* MipsIt *is a system consisting of a software development environment, a system and cache simulator and a highly flexible microarchitecture simulator used for pipeline studies. It has been in use for several years now and constitutes an important tool in the education at Lund University and KTH, Royal Institute of Technology in Sweden.*

## 1. Introduction

In order to learn computer architecture and systems you need to learn how to master abstractions. A computer system is one layer of abstraction on top of the other. At one end you have digital electronics, which in itself can be seen as several layers, and at the other you have complex applications using perhaps techniques such as polymorphic inheritance which needs to be resolved in run-time.

For students studying computer organization and architecture, these abstractions are often confusing as they are not always that distinct. Furthermore, given the high level of integration in modern computers, it is also quite difficult for students to get a good understanding of what is happening deep down in the black centipedes on the motherboard.

At Lund University, and now also at KTH, Royal Institute of Technology, both in Sweden, I have taken part in the development of a set of courses in computer systems, organization and architecture in which laboratory exercises and simulation tool support are used extensively to support learning. In this paper I describe some of the simulation tools that were developed during this process.

The MipsIt set of tools is part of a bigger laboratory exercise environment with a hardware platform, software development tools and a number of simulators. Many of the simulators support *animation* as a support for the students to understand the works of a relatively complex structure.

I first describe some of the trade-offs between hardware platforms and simulators that we considered in developing our exercise material. Next I present an overview of the software system of MipsIt followed by a more detailed description of the animated simulators in sections 4 and 5.

## 2. Hardware vs. Simulation

I think that most instructors would agree with me that exercises where the students get real hands-on experience with digital electronics, assembly level programming, data representation etc. are crucial for the students' learning. Furthermore, it is my firm belief that students must touch, feel and smell[1] the real hardware in a computer organization course. Some universities let the students study computers using only simulated hardware. In my experience, this might lead to a confusion as to what is really happening. Is there really another machine inside this PC, workstation or whatever is used as simulation host?

Therefore, we use real, naked hardware–naked in the sense that there is no operating system on the hardware–to aid the students in understanding computer systems. The current system consists of a development board with a MIPS processor, some memory and a few simple I/O devices [2]. Unfortunately, this does not entirely solve the problem of abstraction. When you connect a development board to a host computer through a terminal program this might be a problem as well. I have had students that answer the question on where the program is executed by pointing to the window on the host computer screen instead of on the processor chip on the board on the desk beside the computer. It is anyway less abstract than if the program executes on a simulator and it is possible to remove the cable between the development board and the host computer and verify that the program still executes with simple I/O devices on the board.

This works well for students to learn about data representation, assembly level programming, simple I/O structures (polling and interrupts) and general low-level computer system design. It is, however, not well suited to study cache memories or processor hardware design as these structures are buried deep inside the processor.

1. Hopefully they smell burned electronics before it breaks!

We have previously let the students build simple micro-programmed processors using discrete components during laboratory exercises. Even though this has been very effective for the understanding of how simple hardware can be organized to execute instructions, we have abandoned it for the first compulsory course.[1] The simplifications that we needed to make in the instruction set that we could implement were to big in order to relate to modern computer instruction set architectures and it was not possible to do off-line development with the hardware used.

So how do we then support the study of hardware structures such as cache memories and pipeline design when we cannot build hardware for it. This is where animated simulation fits in. We have also resorted to simulation of the development board to let the students work at home preparing for lab. exercises and to support distance courses.

I have during the course of being a university teacher found that many students have difficulties of really understanding how cache memories work. They can easily understand the concept, but when it comes to how you should build the hardware to actually implement the concept, it becomes difficult. The concept of pipelining–which is the dominant implementation method of processors today–has similar characteristics. It is easy to understand in principle, but when it comes to the actual hardware design, it becomes tricky. This was the motivation to why we developed the animated simulators described in this paper.

## 3. The MipsIt system

The MipsIt system consists of a development environment, a hardware platform and a series of simulators. The topic of this paper is mainly the animation support in the simulators for cache memory and pipeline simulation, but for completeness I also describe the other parts. All software developed for the MipsIt system are targeted for the Windows (95-XP) platform as host machine.

### 3.1 Development environment

The MipsIt development environment is used to develop software for the hardware platform as well as for the various simulators. It targets the development board shown in section 3.2 but the same binary can be used to execute on the various simulators as well. Figure 1 shows an example of how the development environment might look for a software project with mixed C and assembler files.

The compiler, linker and other tools are standard tools in the gcc tool chain configured for cross-compilation to a MIPS target platform. What we developed was the graphical user interface mimicking the MS Visual DevStudio as a front-end to gcc and which also replaces Makefile by handling projects of source codes and their dependences.

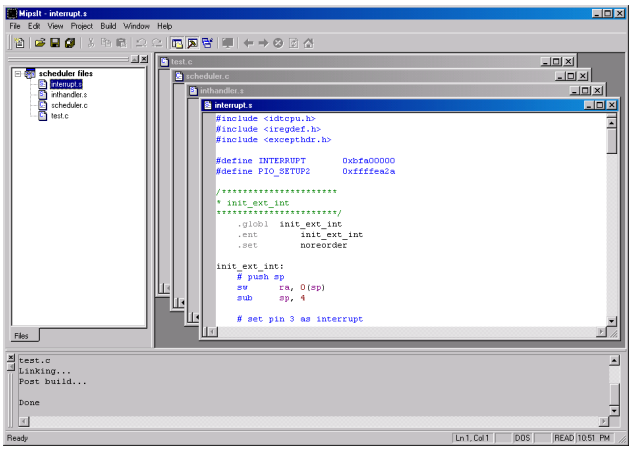


**Figure 1. The development environment is inspired by Visual DevStudio.**

Although not tested, the same front-end should be possible to use with any gcc cross-compiler. The system is highly configurable and there is also an option to run an arbitrary command before or after linking. We use this feature to create an S-record file that can be used for downloading to the development board. We later modified the on-board monitor to use the ecoff-file produced by the compiler/linker.

### 3.2 Hardware

Figure 2 shows a photograph of the development board from IDT that is in use at Lund University [2]. It contains an IDT 36100 micro controller with a MIPS32 ISA processor core [3]. We deliberately chose the MIPS architecture as our instruction ISA because of its simplicity.

Another advantage of the MIPS ISA at the time was also that it is used in the textbooks of Hennessy and Patterson [1, 4]. These textbooks are used at Lund University as well as in many other universities and it makes it easier for the students if they can relate the laboratory exercise material to the textbook directly. The abstractions needed are difficult enough anyway.

The evaluation board itself, as shown in figure 2, contains the processor (chip-select logic, timers and two serial port UARTs are also integrated on-chip), some SRAM, slots for DRAM (not used in our configuration) and EEPROM which contains a simple monitor. The monitor contains some routines from libc which can be taken advantage of for small footprint C programs. The routines include partial functionality of printf. There are also routines to install normal C functions as interrupt routines.

All micro controller signals also appear on the edge connectors of the development board. We developed a simple daughter board containing one eight-bit and one 16-bit parallel bi-directional I/O port. It also contains a simple interrupt unit with three interrupt sources (two push-buttons and

1. The course is given in a 4.5 year programme leading to an M.Sc. in computer science, electrical engineering or information technology engineering.

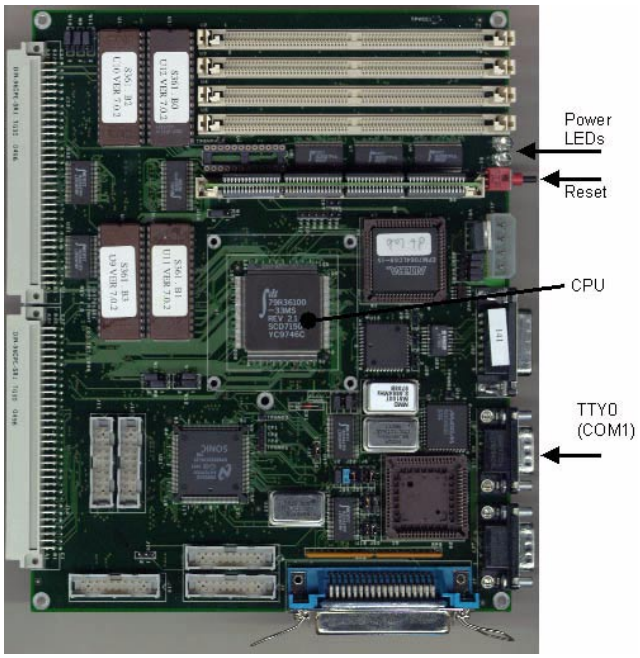


**Figure 2. The IDT development board used in the exercises.**

one adjustable pulse source) that also could be read as a six-bit parallel input port. Three of the bits contain the current status of the three input sources and the other three are latched versions of the same input. These bits retain their value until reset by writing (any value) to the port.

### 3.3 The simulators

The third part of the MipsIt environment is a set of simulators. There is one system simulator which mimics the evaluation board as faithfully as possible. While developing this simulator it was our goal to be able to execute any binary code developed for the target hardware platform. We therefore have to simulate the on-board monitor and all I/O devices, including on-chip timers.

The full code compatibility has been achieved in the system simulator which is described in section 4. This simulator also contains an animated cache simulator. We also wanted to use simulation for microprocessor implementation studies. This resulted in a general micro architecture simulator which is controlled by a simple hardware description language and animation control so that many different microprocessor implementations can be illustrated. This simulator is described in section 5.

## 4. The system simulator

### 4.1 Overview

Figure 3 shows the system simulator with a few of its windows open. The top left window shows a simplified view of the entire system: CPU, instruction and data caches, some RAM, a console window and some I/O devices. The window at bottom left shows the register contents of the CPU. The top right window shows the eight-bit parallel I/O device which consists of eight LEDs and eight binary switches. Just as what we have in hardware. The 16-bit parallel I/O-port is the only one from the hardware that is not implemented in the simulator. The bottom right window shows the simple interrupt sources. Two push-buttons and an adjustable pulse timer.

The main reason for developing this simulator is because it simplifies for the students to study computer organization on their own, at home. Most students have access to PCs with Windows and it is therefore easy for them to download the simulators and development environment to start work on their own.

However, as we designed the laboratory exercises, we found that the simulator could actually be used also in the class-room. Figure 4 shows the memory view in the simulator. The memory addresses are shown to the left and the contents is shown to the right as hexadecimal numbers and an interpretation. In the current view the interpretation is the disassembler view but other possible views are interpretations as unsigned integers, signed integers, single precision floating point numbers and ASCII characters.

The dot to the left in the memory view shows a break point and the line, also to the left, signifies that these instructions are currently in the instruction cache. The darker line shows the current instruction being simulated.

With this view, the simulator became a powerful tool to study instruction execution and the effect each instruction had on the registers etc. Since the MIPS architecture does not have vectored interrupts, it became cumbersome to single-step interrupt routines on the hardware and we therefore used the simulator to study interrupt routines in this detail. The students could also experiment with instruction coding, making hexadecimal instruction codes by hand, entering them in the memory and immediately see if they had coded the instruction correctly. Floating point number coding could be studied in the same way.

### 4.2 Cache simulator

Even with all the benefits as described above, these were not the only purposes for developing the simulator. The major driving force was to introduce animation to aid the students to really understand the inner workings of cache memories. We used the figures of a textbook as an inspiration as how to present the caches graphically [4].

Figure 5 shows the cache view in the simulator. It shows the current configuration of the data cache. The data and the instruction caches can be configured independently. It shows the entire cache contents with both tag and data store. It also shows how the address presented by the processor is divided into different fields for indexing, word select and tag

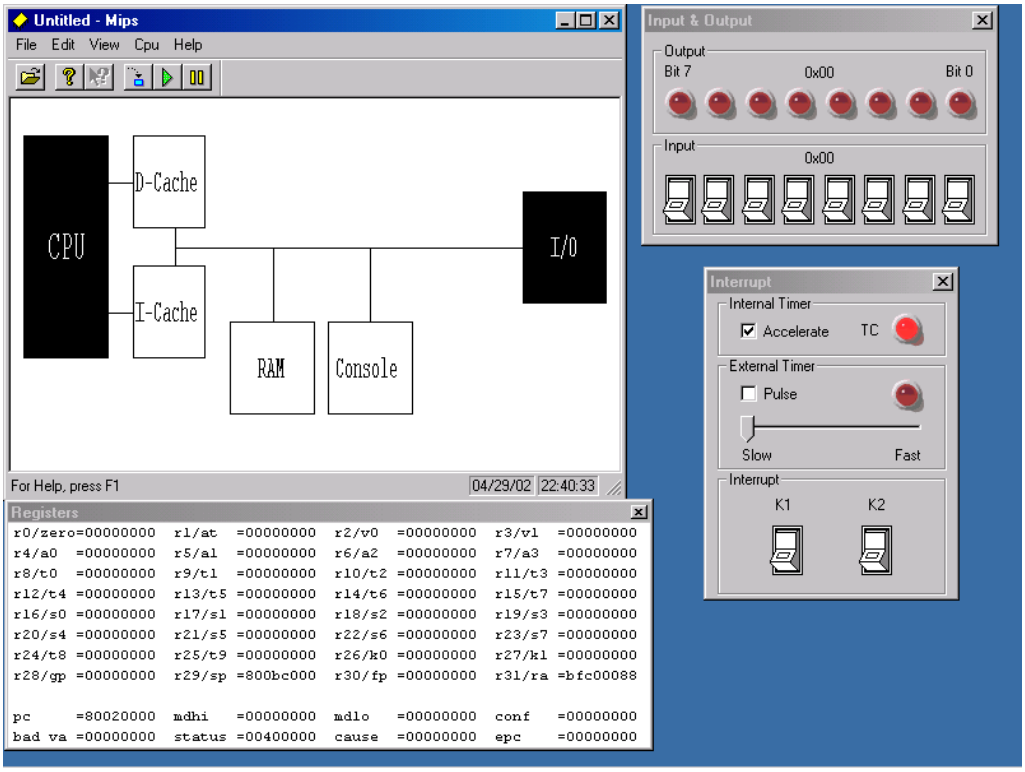


**Figure 3. The system simulator with CPU register and I/O-device windows open.**

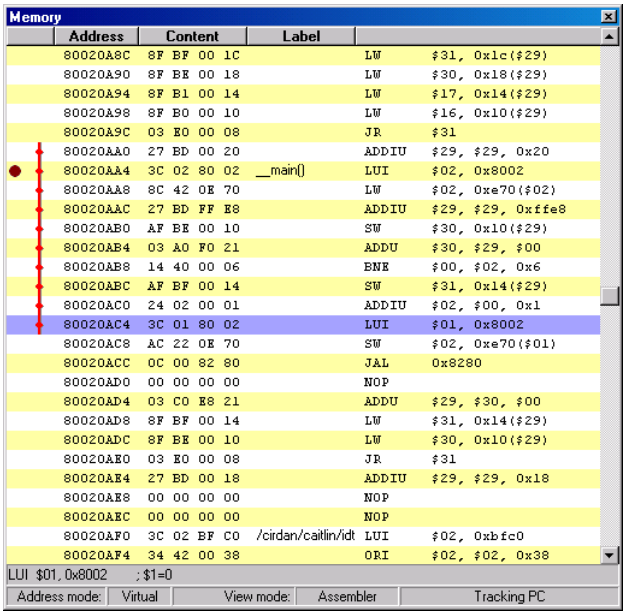


**Figure 4. The memory view in the simulator.**

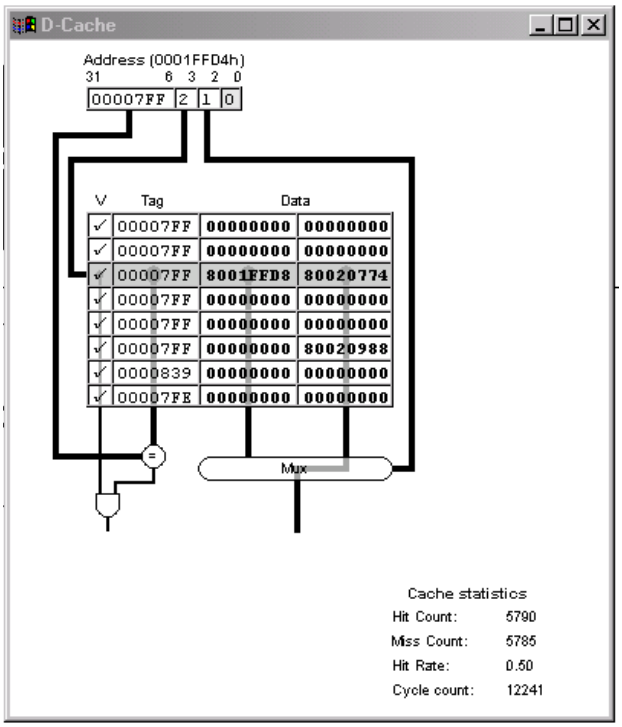


**Figure 5. The animated cache view.**

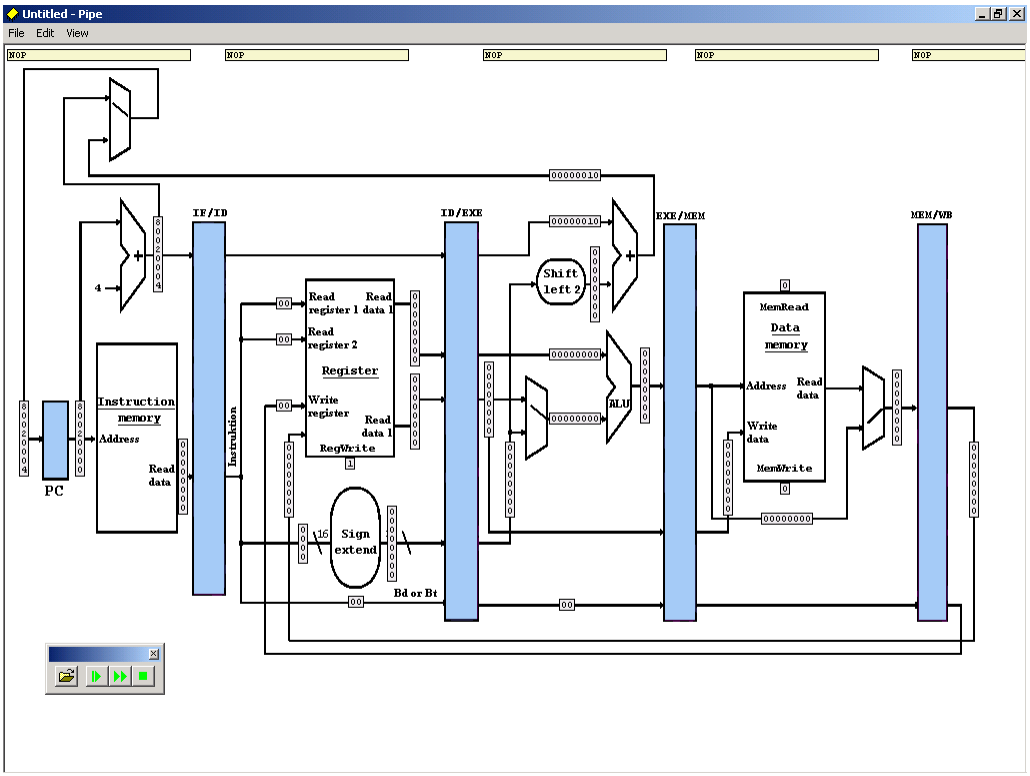


**Figure 6. An example of a simple pipeline simulator view. The simulator is only a shell which can be loaded with arbitrary pipeline structures.**

check. The students can single-step their programs and follow the cache access for every memory reference and therefore gain a deeper understanding of how the cache works.

The simulator also keeps some simple statistics, as shown at the bottom right in the figure. This can be used to compare different cache settings for more longer-running programs.

Memory access penalty can also be configured and it is thereby possible to perform direct comparison with the hardware which contain small instruction and data caches.

## 5. The pipeline simulator

At Lund University, we had a long experience of using animation and graphical representation of pipeline execution [5]. We wanted to make use of this experience, but retain the compatibility with the hardware that we developed for the system simulator as described previously. Another design goal was to make a flexible design that could be run by students at home on their PCs. The existing software was for Sun/Solaris and neither very portable nor flexible.

### 5.1 PipeS and PipeXL

Instead of having a hardwired pipeline design in the simulator software, we developed a flexible simulation shell which could be loaded with different micro architecture implementations. The simulator shell can be used to load programs into memory, to drive the simulated clock signal and to monitor register and memory contents, as in the previously described simulator. However, when the program starts, it contains no description of the simulator hardware. This has to be loaded from a file which describes the micro architecture in a hardware description language (see next section). Figure 6 shows an example in which a simple five stage pipeline without forwarding is shown.

The students can load the memory with a program, just as before, and starts to single step the execution. As the program advances, the pipeline graphics is changed. Muxes are set in the positions needed for execution, data values on buses and inputs are shown and the instruction currently executing in each pipeline stage is shown at the top.

Our experience is that this tool has been tremendously powerful to convey the concept of pipelining to the students. The way that the pipeline is graphically represented is an invention of a student at Lund University in the late 80s but was independently later discovered for use in major textbooks [4, 6].

Figure 7 shows another example of a micro architecture implementation. This is much more complex and complete. In addition to what is present in figure 6, it also contains the control signals, data forwarding and hazard control. We will now see how we can represent different pipeline structures to be used in the simulator.

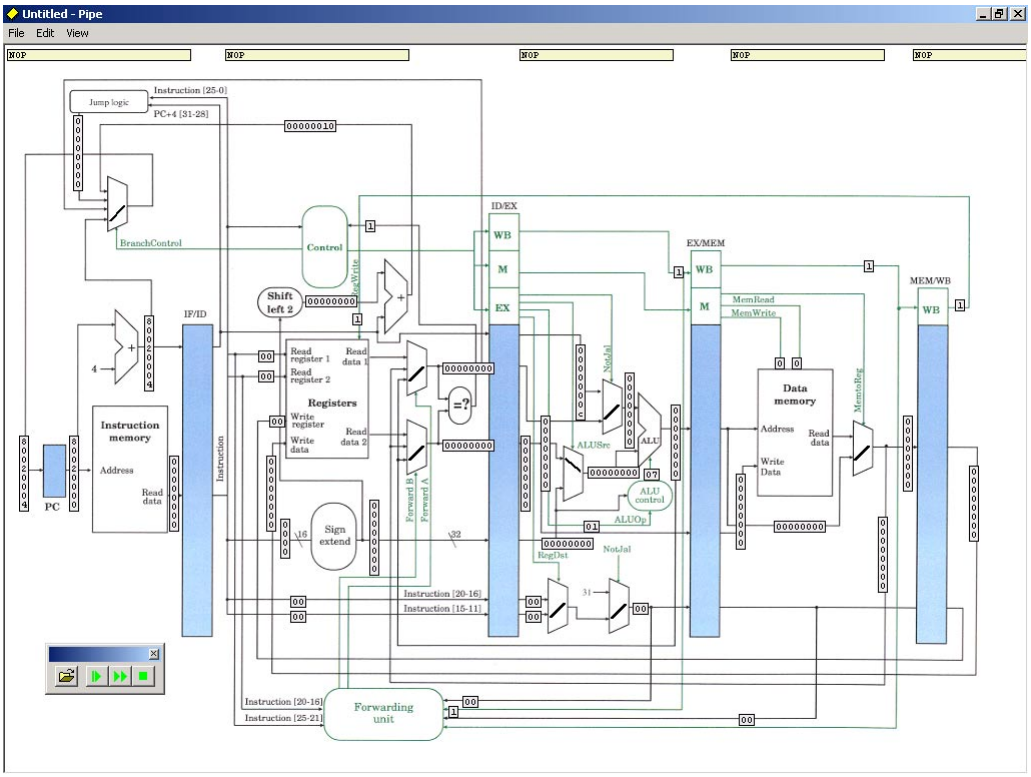


**Figure 7. A more complex pipeline structured using the same simulator shell.**

## 5.2 Some Hardware Description Language

The micro architectural structure of the processor is described in a simple hardware description language which is described here shortly. The pictures shown in figures 6 and 7 are not derived from this language but simple bitmap files that are shown in the window.

The original aim was to use a subset of VHDL as description language to be able to leverage on the body of text written about this language. However, it turned to be cumbersome to parse and instead we developed a simple object oriented HDL where each component is a class which can be instantiated to an object and the inputs and outputs of an object are connected to the inputs and outputs of other objects. Almost any synchronous hardware structure can be expressed in this language. There are also hooks to the simulation framework in the language. Most notably for the clock signal, memory accesses and to the register and memory views of the simulator.

## 5.3 Components

Below is the code for a simple component; the two-input mux:

```
class CMux2
{
  in In0
  in In1
  in Control:1
  out Out

  script
    function OnChange()
    {
      if ( Control.Get()==0 )
        Out.Set( In0.Get() );
      else
        Out.Set( In1.Get() );
    }

    Out.Set(0);
  end_script

  event ALL OnChange()
}
```

At first a class description is used to give the component a name. Next the interface of the component is specified. The default width of inputs and outputs is 32 bits. In this case only the control signal deviates in that it is specified as one bit only.

Then follows a script which describes the behavior of the component. The function OnChange is executed whenever

any of the input signals changes state as described by the last statement in the component description. Input signal values are retrieved by accessing a member function `Get()` and similarly output signal values are set by the member function `Set(x)`. The last lines of the script can be used to set the initial state of the output signals.

A mux is a combinational component and does not contain any state. If any of the input changes, the output is immediately changed also. In contrast, the program counter is a simple component which is clocked. As shown by the example text below, the PC is clocked by a simulated two-phase clock.

```
// this is the Program Counter
class CPC {
  in Ph0:1
  in Ph1:1
  in In
  out Out

  script
    var rPC = 0;

    function OnPh0() {
      rPC = In.Get();
    }

    function OnPh1() {
      Out.Set(rPC);
    }
  end_script

  event Ph0 OnPh0()
  event Ph1 OnPh1()
}
```

The signals `Ph0` and `Ph1` are driven by a clock object and are used to derive the two-phase clock. The value of the PC is stored in an internal variable (`rPC`) which is read from the input on one clock phase and output on the second clock phase.

### 5.4 Connecting components together

When the entire micro architecture has been suitable broken down in components, clocked or not, they can be connected together. The following piece of code shows the connections for the ID-stage in the simple pipeline of figure 6.

```
// components:

object CPC PC
object CInstrMem InstrMem
object CMux2 PcMux
object CiAdd4 IfAdd4
object CRegIfId RegIfId

// Net list:
// to PC
connect PcMux.Out PC.In
// to InstrMem
connect PC.Out InstrMem.Address
// To pc+4 thing
connect PC.Out IfAdd4.In
// to the pipeline register
connect InstrMem.ReadData
                RegIfId.in_Instruction
connect IfAdd4.Out RegIfId.in_PC
// to the pc mux
// (only connections from this stage
// are done here)
connect IfAdd4.Out PcMux.In0
// connect the clock
connect clk.Ph0 RegIfId.Ph0
connect clk.Ph1 RegIfId.Ph1
connect clk.Ph0 PC.Ph0
connect clk.Ph1 PC.Ph1

// and now some probes:
probe InstrMem.ReadData 173 393 16 8 1
probe PC.Out 70 355 16 8 1
probe IfAdd4.Out 148 171 16 8 1
probe PcMux.Out 14 355 16 8 1
probe InstrMem.ReadData 2 4 16 30 2
```

Note how the components first are defined and then connected together in simple connect-statements. At the end, graphical probes are defined. It is these that makes up the animation in the final simulation picture. After the key word `probe`, a signal name is given. This is the signal to monitor. The next two arguments are the x- and y-coordinates of where the probe is to be shown in the pipeline picture. The third argument is the number format for the probe data, the fourth argument is the number of digits to use, and the final argument is the direction of the probe: 0 for horizontal and 1 for vertical. The last probe is different. It is used to show the assembler format of the instruction read from memory. The mux direction is also a probe, but since the mux in this pipeline is controlled in the EX-stage, the probe is also defined there.

We have not yet let students define their own pipeline designs. What we have done is to provide them with a skeleton and let them work out the instruction deciding, hazard control, and forwarding unit for themselves. It has been amazingly simple for them to iron out these details, once they got the hang of pipelining in the first place.

## 6. Conclusion

The software described in this paper has been used in computer organization and architecture education at Lund University and at KTH for several years now. It is now mature enough that we feel it is time to share our experiences which have been very good. We have received quite encouraging feedback from students who both find it useful to work with laboratory exercises at home and who appreciate the graphical animation in the user interface.

## Acknowledgements

The work reported here was performed while the author was affiliated with the department of Information Technology at Lund University. The laboratory exercises were made together with Jan Eric Larsson. Coauthors of the software were: Ola Bergqvist, Georg Fischer, Mats Brorsson, Martin Andersson, Joakim Lindfors and Tobias Harms.

A binary version of the MipsIt package for Windows 95-xp with suitable exercises can be retrieved for educational and personal use from the following web site:
http://www.embe.nu/mipsit.

## References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture — A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, 2002.

[2] Integrated Device Technology, Inc., *79S361 Evaluation board: Hardware User's Manual*, ver 2.0, Sept. 1996.

[3] Integrated Device Technology, Inc., *IDT79RC36100, Highly Integrated RISController: Hardware User's Manual*, ver 2.1, Aug. 1998.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd Ed., Morgan Kaufmann Publishers, 1997.

[5] P. Stenstrom, H. Nilsson, and J. Skeppstedt, Using Graphics and Animation to Visualize Instruction Pipelining and its Hazards, in *Proceedings of the 1993 SCS Western Simulation Multiconference on Simulation in Engineering Education*, 1993, pp. 130-135.

[6] B. Werner, K. Ranerup, B. Breidegard, G. Jennings, and L. Philipson, *Werner Diagrams - Visual Aid for Design of Synchronous Systems*, Technical report, Department of Computer Engineering, Lund University, November 1992.