The KScalar Simulator

J. C. MOURE, DOLORES I. REXACHS, AND EMILIO LUQUE
Universidad Autónoma de Barcelona

Modern processors increase their performance with complex microarchitectural mechanisms, which makes them more and more difficult to understand and evaluate. KScalar is a graphical simulation tool that facilitates the study of such processors. It allows students to analyze the performance behavior of a wide range of processor microarchitectures: from a very simple in-order, scalar pipeline, to a detailed out-of-order, superscalar pipeline with non-blocking caches, speculative execution, and complex branch prediction. The simulator interprets executables for the Alpha AXP instruction set: from very short program fragments to large applications. The object's program execution may be simulated in varying levels of detail: either cycle-by-cycle, observing all the pipeline events that determine processor performance, or million cycles at once, taking statistics of the main performance issues.
Instructors may use KScalar in several ways. First, it may be used to provide demonstrations in lectures or online learning environments. Second, it allows students to investigate the characteristics of specific processor microarchitectures as practical short assignments associated to a lecture course. Third, students may undertake major projects involving the optimization of real programs at the software-hardware interface, or involving the optimization of a processor microarchitecture for a given application workload.
A preliminary version of KScalar has been successfully used in several lecture courses during the last two years in the University Autónoma of Barcelona. It runs on a x86/Linux/KDE system. The graphical interface has been developed using the KDE and QT libraries. The simulator engine running behind the graphical interface is a heavily-modified version of SimpleScalar. KScalar code is available under the terms of the GNU and SimpleScalar General Public License

Categories and Subject Descriptors: C.0 [**Computer Systems Organization - General**]: Modeling of Computer Architecture; C.4 [**Computer Systems Organization – Performance of Systems**]: Design Studies; I.6.5 [**Simulation and Modeling**]: Model Development; K.3.1 [**Computers and Education**]: Computer Uses in Education
General Terms: Design, Performance
Additional Key Words and Phrases: Education, pipelined processor simulator

1. INTRODUCTION

The increasing complexity of processor microarchitectures (pipelining, superscalar, or out-of-order execution) makes them very difficult to understand and evaluate. Processor performance is determined by many interacting events, so showing their occurrence in a complete and easy-to-understand way is a difficult challenge. Students may be helped by visualizations that represent how the computer system works, showing only the important operations and eliminating product-unique details [Yehezkel et al. 2001]. Books on computer architecture education often use two different schemes to visualize processor

This research was supported by the CICYT under contract number TIC 98-0433 and partially supported by the “Comisionat per a Universitats i Recerca” (Grups de Recerca Consolidats SGR 1997).
Authors' addresses: Computer Architecture and Operating Systems Group, Universidad Autónoma de Barcelona, 08193, Barcelona, SPAIN; e-mail: {juancarlos.moure; dolores.rexachs; emilio.luque}@uab.es.

behavior: cycle diagrams and event frequency tables or charts [Hennessy and Patterson 1996; Tanembaum 1999; Stallings 2000].

Cycle diagrams represent the contents of the pipeline stages along time, indicating the cycles and stages where pipeline hazards occur. Cycle diagrams are very complete, because they allow the display of all types of pipeline events, like RAW stalls or operand forwarding. Book space restrictions, though, often limit the size of cycle diagrams to a few cycles.

Also, in order to maintain the diagram's readability, information describing pipeline hazards must be limited. For example, it is not suitable to provide full information about all data cache misses (addresses conflicting in the same way, matching entries in the Miss Address File, and so on). The use of cycle diagrams in a textbook always involves a tradeoff between readability and completeness.

Event frequency tables or charts show the frequency of certain events, like RAW stalls, cache misses, or forward taken branches, in the execution of a relatively large program segment. In this scheme there are no restrictions on the execution run length, which allows us to focus on events that are significant for a given educational purpose. For example, to compare the relative performance penalties of some hazard types, we may display the percentage of cycles that each hazard type has caused in a pipeline stall.

The main difficulty in teaching processor architecture from a textbook is the lack of interactivity inherent in its static representation. Understanding how a processor works and the relations among the different processor elements is easier when we can interact with the running system and modify either the processor microarchitecture or the executed program. Reproducing the program execution using static tables is limited to a small number of short examples. Generating tables by hand is very error-prone, since there are many details to consider and it is easy to forget one of them. For this task, processor simulators may represent a very useful learning tool.

KScalar is a user-friendly processor simulation tool that facilitates the study of complex processor microarchitectures. It is a graphical environment that simulates the execution of a program binary on a parameterized processor configuration, and provides information about the processor and program states in several visualization modes. Displayed data is hierarchically organized to make it more readable and easier to select.

1. A pipeline view allows observing, cycle-by-cycle, all the microarchitecture events that occur during the execution of a program fragment. It shows a snapshot of the processor's state to allow exploring the state of each single instruction or pipeline stage.
2. A cycle diagram view displays the contents of the pipeline stages along several cycles, using a very simple notation to represent pipeline stages and instruction stalls. It provides the whole picture of what has happened on a short period of time (around 15 cycles). Additionally, we may focus selectively on particular details of an instruction or a pipeline stage, requesting help about almost any data that is displayed on the screen, or switching to the pipeline view mode.
3. A statistics table shows the number of occurrences for certain (configurable) pipeline events, for example RAW stalls or cache misses. These statistics may provide an accurate picture of the bottlenecks that limit the performance of the processor for a given program.

KScalar models a wide range of processor microarchitectures, from a very simple in-order scalar pipeline to a detailed out-of-order superscalar pipeline with nonblocking caches, speculative execution, and complex branch prediction. It interprets executables for the Alpha AXP instruction set. They may be short program fragments or very large, complete applications. Finally, the object's program execution may be simulated either cycle-by-cycle, observing all the pipeline events that determine processor performance, or in a million cycles at once and extracting statistics about the main performance issues. In the cycle-by-cycle mode, with the pipeline and cycle diagram views, we can navigate forward and backward through the simulation. This permits advancing the simulation to obtain performance statistics, and then revising the execution to find the reasons for possible bottlenecks.

In summary, the simulator helps us to understanding the performance impact of the various microarchitectural techniques used on current processors, and permits us to find the performance bottlenecks on real program fragments or whole applications.

Instructors may use KScalar in several ways. First, it may be used to provide demonstrations in lectures or online learning environments. Second, it allows students to investigate the characteristics of specific processor microarchitectures as practical short assignments associated to a lecture course. Third, students may undertake major projects involving the optimization of real programs at the software-hardware interface, or involving the optimization of a processor microarchitecture for a given application workload.

A preliminary version of Kscalar was used successfully in several lecture courses during the last two years at our University. KScalar runs on x86/Linux systems under a KDE environment. It is composed of two parts: a graphical interface and a text-based simulator engine. The graphical interface was developed using the KDE and QT graphic libraries. The simulator engine is a heavily modified version of SimpleScalar 3.0 [Burger and Austin 1997]. KScalar code is available under the terms of the GNU and SimpleScalar General Public License. Updated information and updated new versions of the simulator (including extended tutorials and additional examples) can be found on the KScalar Web page <*http://www.caos.uab.es/kscalar*>.

The remainder of this article is organized as follows. Section 2 reviews some introductory concepts on simulation and processor architectures, presents and discusses existing processor simulators, and highlights the similarities and differences with KScalar. Section 3 describes all the simulator capabilities from the user's point of view. The next section presents a tutorial to show how these capabilities may be used in a learning demonstration. We then present the simulator as a teaching tool, providing some advice to instructors who want to use it in their courses. Section 6 lists a large set of future development lines for KScalar, some of them are ongoing projects. Four extra appendices provide complete information about KScalar installation and technical information about the functionality of the simulator engine alone, about the license conditions, and a brief guide to the Alpha AXP ISA.

2. BASIC CONCEPTS AND RELATED WORK

In this section we present the basic concepts regarding the simulation of processor architectures and present and discuss existing processor simulators, highlighting the similarities and differences with KScalar.

2.1 Functional or Instruction-Level Simulators

The instruction set architecture (ISA) of a processor specifies its programmer-visible state and the operations performed by each instruction on this state. It describes all the details required to write correct programs. Functional or instruction-level simulators can interpret programs written or compiled for processors that may be unavailable, providing detailed information about program behavior. Classical address tracing gathers a list of instruction and/or data memory references performed by a system. A generalization is to trace, count, or categorize execution events, such as common values of variables, register usage patterns, and so on.

A functional simulator may be a very good debugging tool, since it runs deterministically and makes it possible to query the program's state without disturbing it. The simulator can also be backed up to an earlier checkpoint in order to implement reverse execution, and can perform consistency checks that cannot be done on real hardware. However, the internal complexity of current processors, not visible at the user-level, makes functional simulators unable to determine execution performance.

There are many functional simulators, many of them designed for educational purposes and others for research purposes. Some simulators are targeted at students who have no background in computer architecture and need a simple introduction. The simulators try to show the basic ideas of computer organization with relatively few details and complexity. For example, they illustrate the von Neumann architecture, the stored program concept, the principle of sequential execution, how data is represented, the process of instruction translation, the fetch-execute cycle, and the use of registers. All these simulation environments allow us to execute code either step-by-step or continuously. All of them present the architecture’s state (registers and memory) in a graphical form. Finally, some of them include visualization of the core components with their interconnections and interactions. Examples of these simulators are EasyCPU and Little Man Computer.

The EasyCPU environment [EasyCpu 2001] is a PC/windows simulator of a reduced subset of Intel 80x86. It provides students with friendly tools for developing small programs. It enables students to learn the syntax and semantics of individual instructions, and to follow on the screen the data transfer between different computer units during instruction execution. One interesting issue is that EasyCPU allows easy switching between simulated I/O and real I/O (handling external hardware or the speaker port).

The Little Man Computer paradigm consists of a walled mailroom, 100 mailboxes, a calculator, a location counter, and input/output baskets [LMC 2001; Yurcik et al. 2000]. A Web-based simulator allows us to examine the concepts underlying the von Neumann architecture, in particular the concept of memory and the relationship between operating systems and computer architecture.

Intermediate simulators attempt to illustrate realistic instruction set architectures and realistic microarchitectures. They also show the architectural state (registers and memory) in a graphical form, and allow execution of the code either step-by-step or continuously. They also visualize core components with their interconnections and interactions, trying to teach how the microarchitecture implements the execution of each instruction. RTLsim, LC2, SPIM, and Mic-1 are examples of these simulators.

RTLsim is a UNIX program that simulates the datapath of a simple nonpipelined MIPs like processor [RTLsim 2001]. It allows the user to select the control signals on the

datapath in order to execute an instruction. Students are given immediate feedback at several levels: when setting the control signals, when observing the outcome in the registers and memory, and in understanding the comments that an automatic system generates when the exercise is not completed.

The other three simulators, LC2, SPIM, and Mic-1, are associated with a textbook. The LC2 environment [LC2 2001; Patt and Patel 2001] simulates a simple computer architecture, and there are Windows and UNIX versions. The SPIM program (UNIX, Windows) simulates the MIPS (R2000/R3000) processor, and is explained in Patterson and Hennessy [1998]. It provides a debugger and a set of simple operating system services. Source code and documentation is available [SPIM 2001]. The Mic-1 is a Java-based simulator that implements the microarchitecture described in the book by Tanenbaum [1999].

KScalar is not intended to teach issues related to the Alpha ISA or to develop correct assembler programs. Since we are only interested in performance issues, providing the actual contents of memory and registers to show the precise semantics of each instruction is out of our scope. However, we have an ongoing project to include support at the ISA-level for assembling and reordering instructions on the fly and for providing enhanced graphical help on the ISA. The goal is to provide a simple interface to permit students to create or modify simple programs that would evaluate the effects of different code optimizations, such as instruction reordering or loop unrolling, on performance.

2.2 Performance Simulators

A performance simulator models the timing behavior of some computer components when executing a program, especially the processor pipeline and the memory system. It may be trace-driven, when data from the execution is obtained from a trace file, or execution-driven, when it performs its own instruction-level simulation. The microarchitecture of a processor specifies its internal organization (caches, instruction queues, arbitration policy of storage ports) and includes states not specified at the ISA level. It determines the final performance of the processor. A performance simulator provides access to the processor microarchitecture, and may be used to analyze its behavior and, in turn, to improve the design and implementation of everything from microarchitectures to architectures to compilers to applications.

A cycle-by-cycle simulator models all simulated events in terms of processor clock cycles. To get the ultimate measure of a processor's performance, it is necessary to estimate its cycle time on a real implementation. This detailed model must specify several details of the specific implementation technology in order to determine the longest critical path through any of the pipeline stages.

Many existing processor performance simulators may be of interest. We provide references and comments about some existing processor simulators, and will compare them with KScalar. These simulators incorporate the complexity of today’s high-performance processors. In general, all of them simulate complex pipelined microarchitectures and permit visualizing the pipeline state and obtaining statistics about the execution performance. They can be used to investigate the effects of performance-enhancing techniques such as instruction-level parallelism, caches, and branch prediction.

WinDLX [2001] is a MS-Windows simulator for the DLX architecture described in Hennessy and Patterson [1996]. The code window is automatically colored to indicate the

status of instructions in the pipeline. Also, a pipeline diagram allows inquiry about the progress of an instruction at a given stage. Finally, a pipeline clock cycle diagram is colored according to the pipeline stage. The simulator models an in-order execution processor, without superscalar execution. It does not model cache behavior and models very simple branch prediction.

SimOS is a very complete environment for studying the hardware and software of computer systems [Rosenblum et al. 1997]. It simulates the CPU, caches, multiprocessor buses, disk drives, ethernet, consoles, and other devices. It can even boot a commercial operating system. Using multiple interchangeable simulation modes, the users can control the tradeoff between simulation speed and simulation detail. It runs on a UNIX machine and in text-mode.

SimpleScalar [Burger and Austin 1997; SimpleScalar 2001] is a tool set containing compiler, assembler, linker, simulation, and visualization tools for the SimpleScalar architecture and the Alpha AXP architecture. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation. The tool set provides simulators ranging from a fast functional simulator to a detailed, out-of-order issue processor that supports nonblocking caches, speculative execution, and state-of-the-art branch prediction. The tool set provides researchers with an easily extensible, portable, high-performance test bed for systems design. The SimpleScalar tools are used widely for research and instruction, for example, in 2000 more than one third of all papers published in top computer architecture conferences used SimpleScalar tools to evaluate their designs. All simulators and the pipeline tracing facility run on a text-based environment.

KScalar provides a graphical interface to a SimpleScalar-based simulator running in text mode. The graphical interface makes it more appropriate for student use to evaluate complex processor microarchitectures. KScalar does not try to show the intricacies of the microarchitecture implementation, but only show the pipeline events that affect performance (cache stalls, speculation squashes, etc.). To our knowledge, this is the only free tool available that permits evaluation from a user-friendly graphical interface of very complex mechanisms (realistic caches, out-of-order, speculative, and superscalar execution). Although there are lots of parameters to configure the microarchitecture, modifications to the text-based simulator in order to include additional features does not require modifying the graphical front-end.

3. KSCALAR USER'S GUIDE

KScalar is a cycle-by-cycle, execution-driven, performance simulator, with a graphical interface that makes it suitable for studying and evaluating complex processor microarchitectures. Its main characteristics are as follows:

1. It runs on x86/Linux systems under a KDE graphical environment.
2. It interprets executables for the Alpha AXP instruction set, ranging from short program fragments to very large, complete applications. It does not support system-level simulation; system calls made by the simulated binary are intercepted and executed by the Linux operating system, with the results copied into the simulated program's memory.

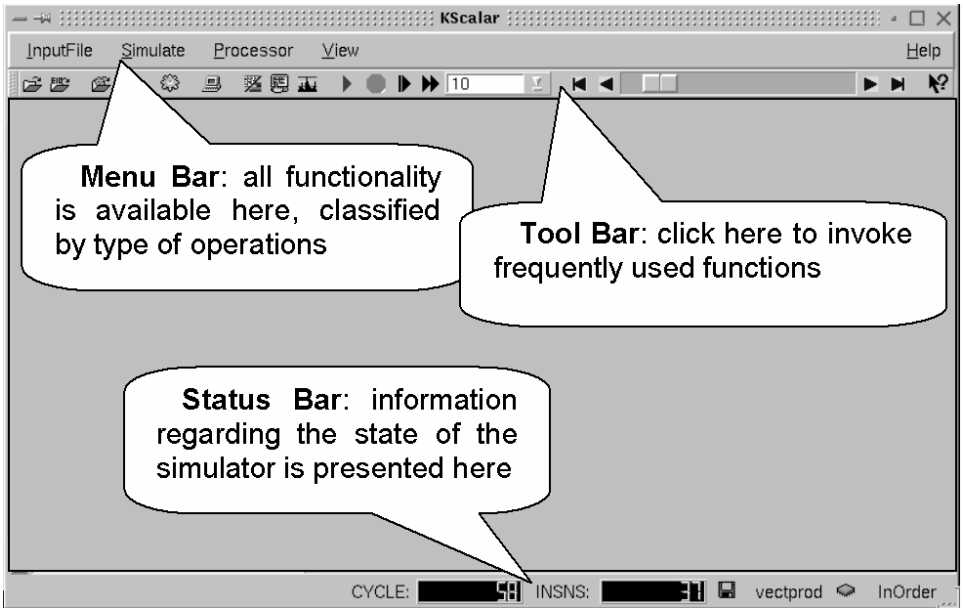


Fig. 1. Main elements of the KScalar main window.

3. It models a wide range of processor microarchitectures: from a very simple in-order scalar pipeline, to a detailed out-of-order superscalar processor with nonblocking caches, speculative execution, and complex branch prediction.
4. It provides information about the processor state and the program state in a hierarchical way, using several visualization modes: pipeline view, cycle diagram view, and statistical table. We can navigate forward and backward through the previous simulated processor states.
5. Execution may be simulated either cycle-by-cycle, observing all the pipeline events taking place, or by a million cycles at once, and obtaining statistics on the main performance issues.

3.1 Overview of the Main KScalar Window

The KScalar graphical interface has the look and feel of the KDE/Linux environment. Basic knowledge of a windows system is assumed along the lines of this description. Figure 1 shows the basic components of the simulator window, i.e., the classical menu, tool, and status bars.

These are present in any windows-based application, and should be very intuitive to understand and use. As a general guideline, we may refer intensively to the context-sensitive help provided in almost any widget in the window. Pointing to an object for a few seconds will cause tip descriptions to pop up. Clicking on the help button (an arrow with a question mark on it) and then on the object of interest will provide more detailed information.

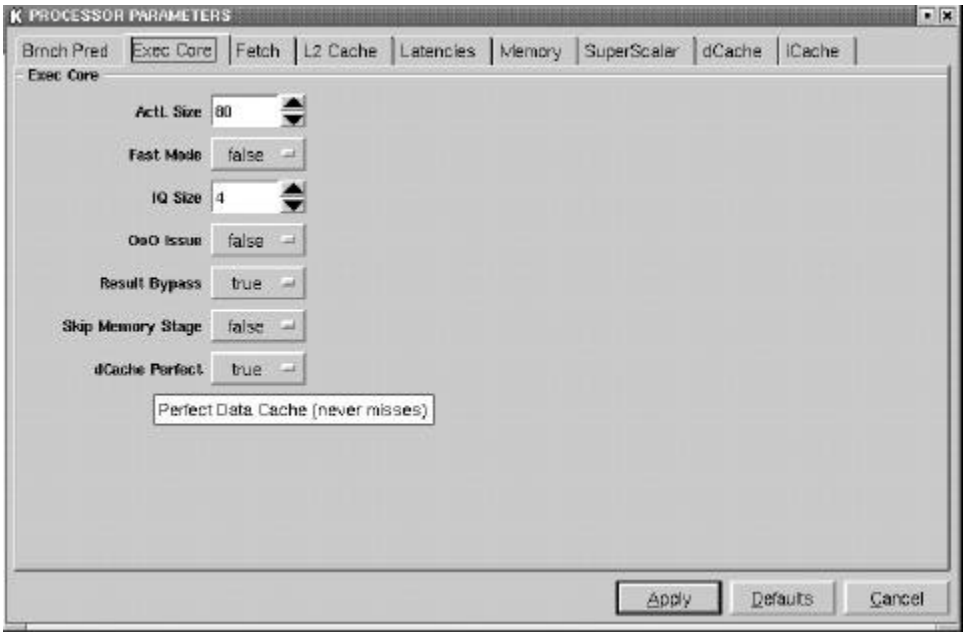


Fig. 2. Processor configuration dialog, which shows the particular options for the execution core. All options are modifiable and pop-up a tip description about their function and format.

The particular elements inside each window bar are described in the order they are likely be used in a simulation session:

1. configuring the simulated processor's microarchitecture;
2. loading the object program;
3. advancing simulation;
4. visualizing information in two basic modes: pipeline view and cycle diagram view;
5. showing simulation statistics.

3.2 Configuring the Processor's Microarchitecture

Configuration may be done interactively, selecting options from a dialog window (see Figure 2), or using a configuration file. A configuration file may be created easily by first selecting options interactively and then using the save facility to store the configuration for later use. Configuration files are expected to have the extension *.cnf* and tbe found in the directory *./cnf*. Loading a new configuration file or modifying the current configuration reinitializes the simulation. The bottom-right corner of the window shows the name of the configuration file used to specify the current options, if any.

Configuration commands are grouped into the Processor menu, and may be accessed using specific buttons on the toolbar or pressing dedicated hotkeys. Individual configuration parameters are classified into several groups:

1. *Superscalar*. It is possible to select, individually, the width of the following pipeline stages: fetch, decode, issue, and retire. The maximum branch predictor bandwidth is also selectable.
2. *Fetch.* Allows modifying the size of the fetch queue and the misprediction recovery latency. It is possible to indicate the use of a perfect branch predictor or a perfect instruction cache. Perfect means that fetch unit performance is never reduced due to branches (either taken or not taken) or instruction cache misses. The size of the Branch Queue is user defined, and determines the maximum number of predicted and unretired branches.
3. *Execution core*. Permits selecting between in-order and out-of-order instruction dispatching. It is possible to activate or deactivate operand bypass and the use of a perfect data cache. For in-order dispatching, we may force all instructions to pass through the pipeline memory stage, or allow nonmemory instructions to skip the stage. For out-of-order processor models, the size of the instruction queue and the active list may be specified. Figure 2 shows a snapshot of the execution core dialog, which also exemplifies the use of the context-sensitive help to get tip information.
4. *Branch prediction*. Static and dynamic branch prediction algorithms may be used. Dynamic branch prediction may be local, global, or a hybrid model that combines both methods using a selection table (McFarling algorithm [McFarling 1993]). We may choose the size of the local and global history tables and the size of the prediction counters. The simulator also models a parameterized branch target buffer and a return address stack.
5. *Data cache, instruction cache, and an L2 unified cache*. The organization and topological parameters of all the caches are user-defined (line size, number of ways and sets, and replacement policy). Access latencies are defined for the L1 data cache and the L2 cache. Additionally, the number of data cache ports is user-defined.
6. *Execution latencies*. The latency of each operation type may be configured individually.
7. *Memory*. The L2 Bus and System bus latencies and bandwidths may be specified. It is possible to indicate the use of a perfect L2 cache. Finally, the size and the number of ports of the Miss Address File (MAF) is user-defined, which determines the maximum number of pending miss requests and the number of miss requests that may be serviced per cycle.

Some option values may be invalid or some combination of options may be incompatible. In this case, an error message appears when we press the ***apply*** button.

3.3 Loading an Object Program: Binary fFles and EIO Trace Files

The simulator is able to execute Alpha AXP EV6 binary code (see Appendix D for a short description of the Alpha AXP ISA). Object programs may be loaded in a typical executable format, produced by a compiler and linker (statically linked), or may be specified as external input/output (EIO) trace files.

EIO files are generated from the original program binary using the internal text-based simulator provided in KScalar. Appendix B describes the procedure to create an EIO file.

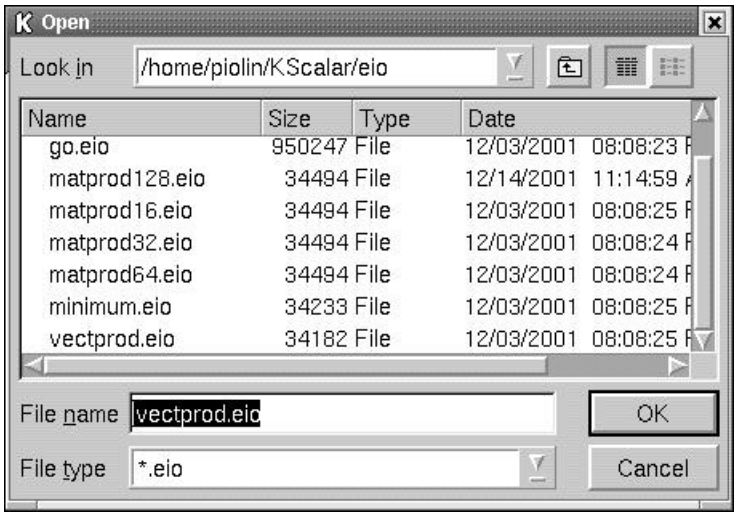


Fig. 3. File dialog used to load a program EIO trace file.

It contains the initial architectural state (registers and memory), including the code text and the command-line arguments. It also contains the trace of the external input/output operations done during the program's execution (i.e., the system calls made by the simulated program to the operating system, with all its inputs and outputs). Using the EIO trace, the simulator can repeat the same execution several times. EIO traces are 100% reproducible, since the sources of irreproducibility are captured in the trace file. Also, EIO files provide a convenient method to execute interactive programs in batch mode, and allows packaging an experiment into a single file (including options, user environment, file accesses, network I/O, etc). Finally, since EIO traces capture the output of a program (writes, network output, etc.), the simulator may check any output attempted against that recorded in the EIO trace file, making EIO trace files self-validating.

EIO files are expected to have the extension *.eio* and to be found in the directory *./eio*. Figure 3 illustrates how to use a file dialog to load an EIO file.

3.4 Advancing Execution: Step, Run, Stop, and Fast Forward

There are three modes to advance the simulation: stepping a single cycle, simulating cycle-by-cycle in continuous mode, and executing a given number of cycles at maximum speed. Simulation history is normally maintained as the simulation advances. This allows us to navigate forward and backward through the previous simulated processor states, and observe certain pipeline events in detail, or to take a broader look at several contiguous executed cycles. Storing simulation history, though, consumes memory and slows down simulation. For this reason, the third simulation model does not store history, making its operation two or three orders of magnitude faster. This last mode is designed to obtain execution statistics for thousands or millions of simulated cycles.

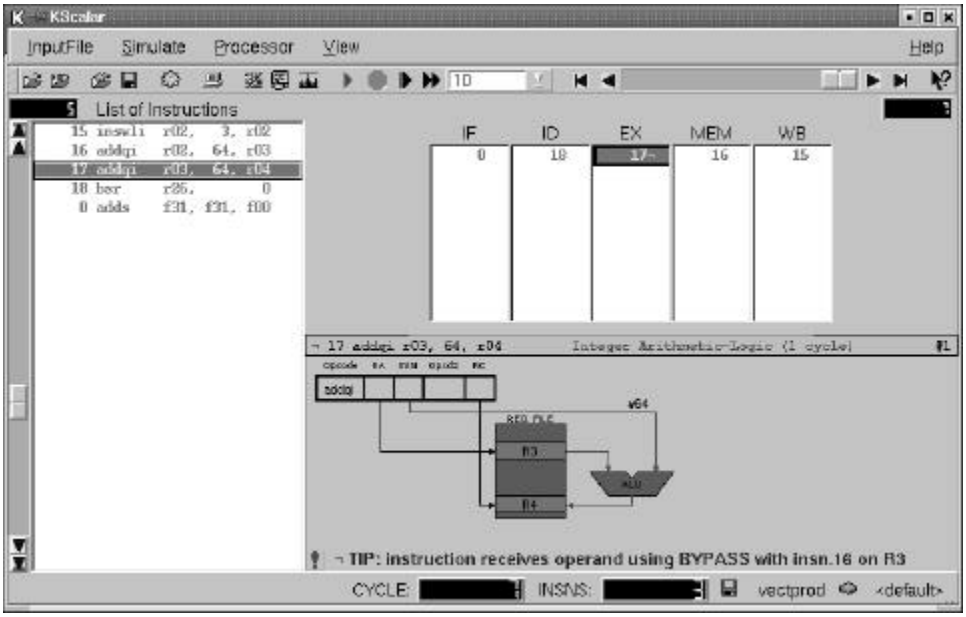


Fig. 4. Pipeline view. The active list shows instructions in order. The pipe stage lists show the instruction Pctag, while the instruction view window describes the semantics of the selected instruction.

The *step* command (available as a button, as a menu option, and with the hotkey F7) advances the simulation a single cycle and permits us to observe in detail all the events during the execution of this cycle. The *run* command advances the simulation cycle-by-cycle, storing simulation history, but continues the simulation until the *stop* command is invoked (generally by clicking on the stop button or hotkey F9). It is possible to configure a simulation delay, in order to avoid the simulation advancing too fast.

Finally, the *fast forward* command advances the simulation a selectable number of cycles without updating simulation history. This behavior introduces holes in the pipeline and cycle diagram views, and may lead to some confusion.

3.5 Pipeline View

The pipeline view allows us to observe, cycle-by-cycle, all the microarchitecture events that occur during the execution of a program fragment. It shows a snapshot of the processor state, allowing the exploration of the state of each single instruction or pipeline stage (see Figure 4). There are three visualization areas: the active list, the pipeline stages, and the instruction information window.

Instructions are identified using a combination of two tags, which provides a unique identification method: PC tags and counter tags. *PC tags* classify instructions using the sequential order of their memory addresses. Instructions stored in contiguous memory addresses have contiguous PC tags. This kind of tag is the most intuitive way to identify instructions, and it is often found in textbooks about computer architecture. The first

Fig. 5. Detail of the state slide. It permits navigating through the simulated cycles. The associated counter indicates the cycle being visualized. There is a similar bar for navigating through instructions.

instruction in the text segment is assigned PC tag 0, the following instruction is assigned tag 1, and so on.

PC tags, however, do not uniquely identify the instruction instances in the processor pipeline. For example, when a loop is executed, it may frequently happen that two instances of the same instruction, each belonging to different loop iterations, are concurrently present in the pipeline. One way to differentiate this instruction instances is by using an additional counter tag. We count all the instances of each static instruction with a unique PC tag and use a new *counter tag* for each executed instance. Therefore, counter tags are built using the dynamic program execution order (which matches the decode and retire order even in an out-of-order processor). Combining a PC-tag and a Counter-tag always allows us to differentiate the instructions in the processor pipeline. All the instances of the same static instructions have the same PC tags, maintaining intuitiveness, but have different counter tags that establish an order among them. In the graphical simulator, counter tags are not shown. Instead, a different color is used for different counter tags (from a palette of eight different colors).

The active list (on the left) contains, in instruction-fetch order, all the instructions into the pipeline. It shows the PC-tag identifier of the instruction and its mnemonic description.

Each pipeline stage (upper-right) shows the PC-tag of the instructions currently in that stage (color may be used to differentiate instances of the same static instruction), sometimes followed by a character symbol indicating that a significant event has occurred with the instruction. A significant event does not necessarily mean that a pipeline hazard has occurred. It may indicate, for example, the use of a bypass path to forward a result to the executing instruction (as is the case in the example of Figure 4).

By clicking on an instruction, some semantic information is displayed in the instruction information window (bottom-right), including the type of operation and its execution latency, the operand registers, the constants used, and the accessed memory address. A short explanation of the event is also shown at the bottom line. Additional help may be obtained by clicking on the help button and then on the "!" symbol.

At each simulation step, the view is updated with the new pipeline state, and the last instruction being entered in the pipeline is automatically selected and its semantics are visualized. We can navigate forward and backward through the previous simulated processor states using the horizontal and vertical sliders (Figure 5). This permits detailed revision of the execution, analyzing events that were previously ignored. Remember that the status bar always shows the counters for the last simulated cycle and the last retired instruction, while the counters on the sliders show the cycle and instruction being visualized at this moment.

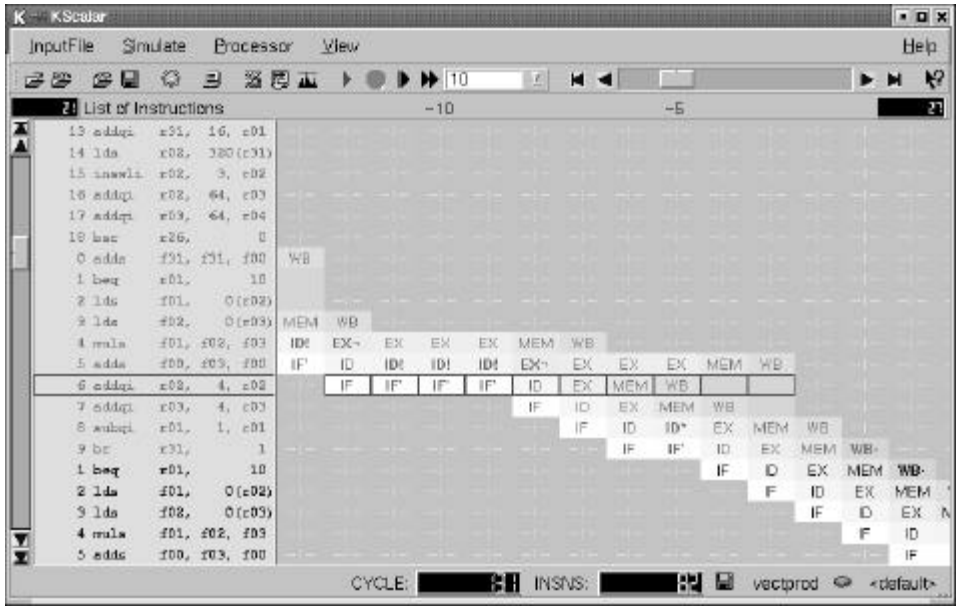


Fig. 6. Cycle diagram view. Colors indicate different iterations of a loop.

3.6 Cycle Diagram View

A cycle diagram view displays the contents of the pipeline stages along several cycles, using a very simple notation to represent pipeline stages and instruction stalls. It provides the whole picture of what has happened in a short period of time (14 cycles). We may select a specific instruction and pipeline stage, and then switch to the pipeline view mode in order to get additional help regarding the instruction. There is a button for switching between the pipeline and the cycle diagram views (an also the hotkey CTRL+T).

The cycle diagram representation is very similar to the representation in the Patterson and Hennessy [1996] textbook. It includes the same symbols used in the pipeline view to indicate the occurrence of a significant pipeline event. Figure 6 visualizes some instructions inside a loop. Note how instructions 1-4 appear two times, but with a different color. This indicates that they have a different counter tag and, then, belong to different loop iterations. Visualization counters point to the bottom instruction and to the leftmost simulation state. The horizontal and vertical sliders also allow navigating forward and backward through the previous simulated processor states.

It is easy to find the dependency chains that cause the main problems in the execution. They are visualized as large rows in the cycle diagram and make the bottom part of the diagram grow to the right. If an instruction is delayed (large row), we must look for the stage where the instruction is delayed (a stage that is replicated several times in the row) and look for the reason it is delayed (waiting for a dependence with a previous instruction). The process is repeated with the previous instruction until the full dependence chain is found (often, the same problem is repeated many times in a loop).

Fig. 7. Predefined statistics for in-order execution (loaded when executing the file *inOrderStats.cmd*).

3.7 Showing Statistics

A statistics table shows the number of occurrences for certain (configurable) pipeline events, for example RAW stalls or cache misses. These statistics may provide an accurate picture of the bottlenecks that limit the performance of the processor for a given program.

KScalar does not provide a fixed set of statistics. Instead, statistics have to be generated (defined) dynamically for some of the following reasons:

1. Obtaining statistics wastes considerable simulation time, and we often want to obtain statistics from a large program fragment; allowing "activating" and "deactivating" statistics affords simulation time.
2. Statistics strongly depend on the model being simulated. For example, the way to measure pipeline stalls is very different if we use an in-order processor model or an out-of-order processor model. Selecting statistics dynamically permits us to adapt the measures to the processor model.
3. It is impossible to provide all the statistics that a user should try to obtain. For example, the number of times conditional branch instructions jump forward is mispredicted when using a local predictor. Facilitating a mechanism to create statistics allows us to adapt better to user requirements.

All statistics are constructed from built-in state variables. State variables can be inspected using a menu option or a button. For example, there is a state variable indicating whether

Fig. 8. Communicating with the underlying simulator core using the expert mode window.

a data cache miss has occurred in the current simulation cycle. State variables represent events occurring on the processor microarchitecture and describe the state of the simulated processor. Some of them are present in any processor configuration, while other variables are particular to a given processor model. For more details about how to create statistics, refer to Section 5 and Appendix B.

Although creating statistics is a complex task, the method for "installing" predefined statistics is very simple. A command file, which contains commands understood by the underlying simulator core, can be executed to install the statistics. For example, Figure 7 shows a list of statistics created when invoking the (provided) command file *inOrderStats.cmd*. Command files are expected to have the extension *.cmd* and to be found in the directory *./cmd*. Loading a new configuration file or modifying the current configuration reinitializes the simulation and loses the predefined statistics. Help information about the meaning of each statistic may be obtained by pointing to the statistic name for a few seconds.

The underlying text-mode simulator provides functionality that is not fully integrated in the graphical interface. However, there is a special expert mode window that permits communicating directly with the simulator engine. Figure 8 gives an example that also presents some of the available text-mode commands.

4. TUTORIAL: A FIRST EXAMPLE

This tutorial describes a session using KScalar that shows how a simple processor pipeline works. The tutorial's basic educational goal is to show and quantify the problem of true data dependencies (RAW) and contention on a single register write port. We start

with a very simple 5-stage pipeline configuration (similar to the Patterson and Hennessy [1996] pipeline model) with all execution latencies equal to one cycle.

1. We analyze the performance advantage of implementing the bypassing of results (or forwarding the results).
2. We study the problems introduced when multicycle operations are considered.

This simple example is not meant to show all aspects of KScalar, and only acts as an initial introduction. We may use context-sensitive help by pressing the help button and then pressing anywhere. The software distribution provides many configuration files and some additional object binaries that will be helpful to use.

At this point it is assumed that we have an installed KScalar distribution (see Appendix A), and have basic knowledge both in the use of the Linux-KDE environment and about the Alpha AXP achitecture (see Appendix D).

4.1 The Simulated Object Program

This chapter uses a synthetic code fragment that performs the scalar product of two vectors. The basic subroutine is called from a main program and then the execution ends. A C implementation for that code is something like this:

```
float V1[16], V2[16], Res;
void main()
{
  VectProd(16, V1, V2, &Res);
  exit(0);
}
void VectProd(int length, float *A, float *B, float *C)
{
  float tmp=0.0;
  for (;length; length--, A++, B++)
    tmp= tmp + (*A) * (*B);
  *C = tmp;
}
```

Next, we list the assembler code with register-transfer and C-like comments. Instructions are numbered from 0 to 20 (execution starts at instruction 13) and instruction numbers are used as substitutes for code addresses (this is the way KScalar displays instructions).

```
VectProd:
 0> adds    f31, f31, f0      ; f0 ← 0.0     tmp=0.0
 1> beq     r1,  10           ; if (r1==0) goto 10 for (;length; )
 2> lds     f1,  0(r2) ; f1 ← MEMORY(r2)   access to *A
 3> lds     f2,  0(r3) ; f2 ← MEMORY(r3)     access to *B
 4> muls    f1, f2, f3 ; f3 ← f1 · f2 *A · *B
 5> adds    f0, f3, f0 ; f0 ← f0 + f3 tmp += ...
 6> addqi   r2,  4, r2 ; r2 ← r2 + 4  A++
 7> addqi   r3,  4, r3 ; r3 ← r3 + 4  B++
 8> subqi   r1,  1, r1 ; r1 ← r1 - 1  length--
 9> br      r31, 1            ; goto 1               repeat for
10> sts     f0,  0(r4)        ; MEMORY(r4) ← f0      *C = tmp
11> ret     r31,(r26)  ; jump to r26          return
```

```
main:
13> addqi   r31,16, r1 ; r1 ← 16               length
14> lda     r2, 320(r31)   ; r2 ←140h (16bits)  build &V1
15> inswli  r2,  3, r2        ; r2 ← 140h             build &V1
16> addqi   r2, 64, r3 ; r3 ← r2+64      build &V2 as &V1+16*4
17> subqi   r3, 64, r4 ; r4 ← r3+64      build &Res as &V2+16*4
18> bsr     r26, 0            ; call 0 (r26←nextPC) call VectProd()
19> addqi   r31, 1, r0        ; r0 ← 1            (exit system call)
20> syscall                   ; system call    exit()
```

The execution of function *VectProd* consists of a loop that is executed 16 times (16 is the size of each vector and is the initial value of the variable *length*, allocated to register r1). At each iteration, two 32-bit memory loads, two floating-point operations, three integer operations, and two branch operations (one conditional and one unconditional) are executed. Memory loads start on addresses 0x140000000 and 0x140000040 (variables *A* and *B*, which are allocated to registers r2 and r3). The final result is written to memory address 0x140000080 (variable *Res*, allocated to register r4). Instructions 0, 10-11, and 13-20 are executed only once, while instructions 1t-9 are executed 16 times. Instruction 1, the conditional branch, is executed 17 times, and only the last time is taken. In total, 156 instructions are executed.

4.2 The Simulated Processor Microarchitecture

As we said before, a default processor configuration is initiated. We will explain later how to change it, but for now, we will use the default configuration. The main characteristics of the default 5-stage processor pipeline are listed bellow. Detailed characteristics will be introduced during the simulation tutorial.

1. **IF**: One instruction fetched per cycle, ideal instruction cache (always hit), and ideal branch prediction (always fetches the correct instruction).
2. **ID**: One instruction decoded per cycle. RAW or resource hazards stall the instruction here. Up to two operands are read from the register file. Results from an older instruction may be forwarded to the decoded instruction.
3. **EX**: One instruction issued per cycle (always in order). Execution latency is always 1 clock cycle. Taken branches (both conditional and unconditional) modify the PC at the end of this stage (perfect branch prediction, however, allows ignoring branches).
4. **MEM**: All instructions traverse this stage, but only loads/stores access the ideal data cache (always hit).
5. **WB**: At most one result written per cycle.

4.3 Starting KScalar and Loading an Object Program

When KScalar is started, a clean main window appears with several menu options. A default processor configuration is initiated but no object program is loaded. You can check the current configuration and program in the status bar, at the bottom-right of the main window.

To begin the simulation, a program code must be loaded. To load our example, click on the ***InputFile*** menu and click ***open EIO*** (you can either push the corresponding tool button or press the hotkey CTRL+O. Hotkeys are shown in the menus). We do not

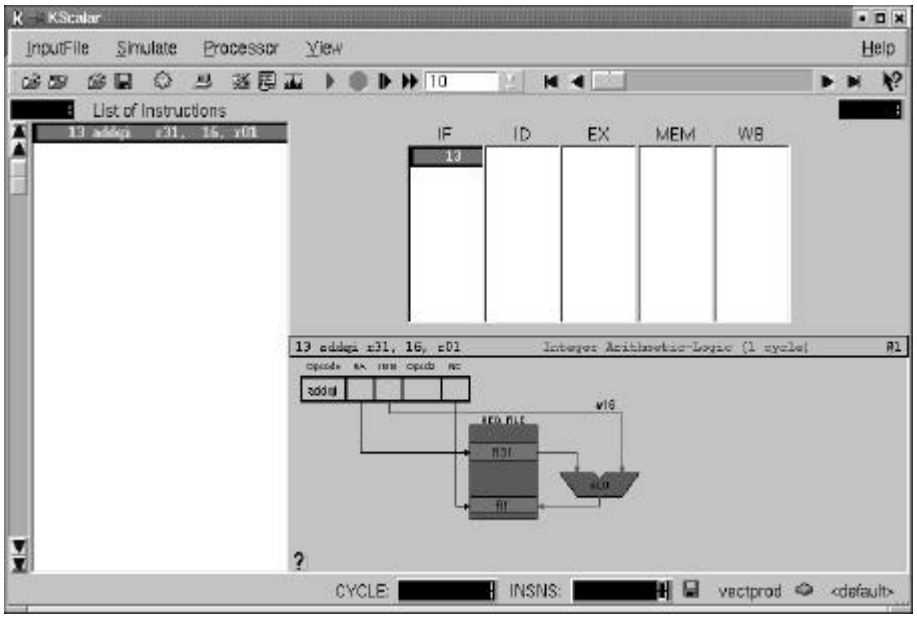


Fig. 9. Pipeline view. An instruction has entered the pipeline and is shown in the IF stage. Semantic information is displayed in the bottom-right window, specifying the instruction's input and output registers, and its operation's type and latency.

specify where the command buttons are. We just point to the buttons in the tool bar, wait a few seconds and a tip description of the button function will pop up.

After invoking the ***open EIO*** command, a file window is opened and you have to type or select file name "*vectprod.eio*" and then click the the ***OK*** button. Confirm the loading by looking at the status bar.

It is now time to start the simulation, so click ***Simulate*** in the main window. Click ***step one Cycle*** in the pull-down menu which now appears Pressing F7 or clicking the ***step*** button has the same effect. Don't worry about configuring the simulated processor for now. The default configuration is just the simple one described in the previous section.

At this moment, the simulation has advanced one cycle (see the instruction counter in the status bar), an instruction has entered the pipeline, and some information regarding the instruction is displayed on the screen.

4.4 Pipeline View

Let us take a look at how KScalar represents the processor pipeline. The simulator starts with the pipeline view as a default view, but we may toggle between the pipeline and the cycle diagram view using the option ***Toggle Pipeline View*** (CTRL+T or toggle button). We must not toggle the view, for the moment.

Figure 9 shows the pipeline view after executing one instruction. There is one large and wide column, called *Active List*, which displays a list of currently active instructions. There are five shorter and narrower columns representing the processor's five-stage

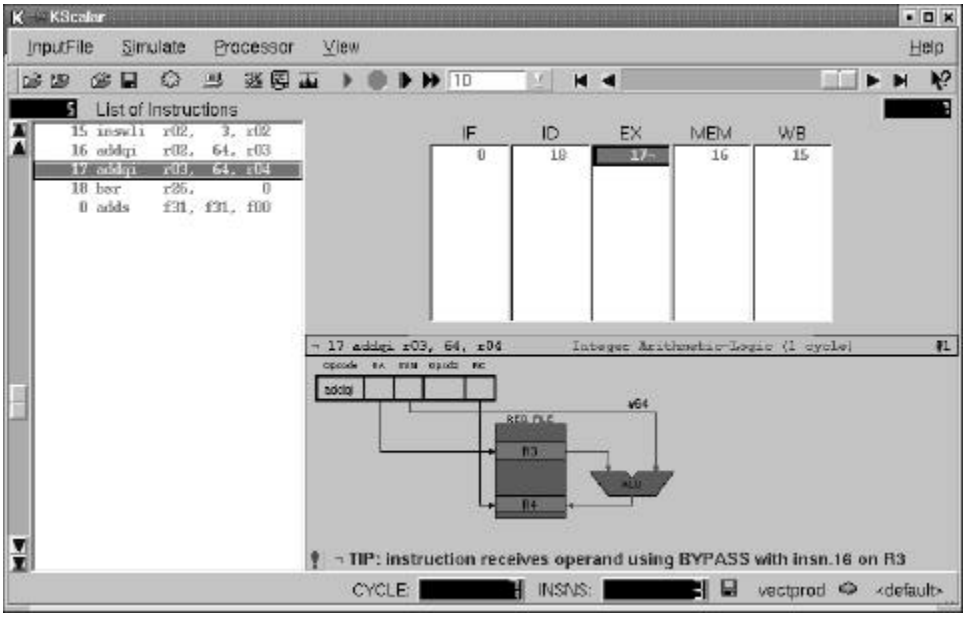


Fig. 10. Pipeline view. An instruction has received an input operand from a bypass path. See the ¬ symbol in theEX pipeline stage and the help explanation at the bottom of the instruction view window.

Pipeline. Instruction number 13 has entered the pipeline and is in the **IF** column, i.e., it is in the instruction fetch stage. Clicking on the help button and then on the column title, provides detailed information about the pipeline stage function. This information is context-sensitive, i.e., it changes when a different processor model is simulated. So make extensive use of the help facility each time a different configuration is loaded.

Semantic information is displayed in the Instruction View window, specifying the input (R31) and output (R1) registers; it also specifies the immediate value used (16) and the type (integer arithmetic) and latency (1 cycle) of the operation. The number at the top-right of the instruction view window represents the number of execution instances of this static instruction. In this case, since the instruction is executed for the first time, its value is 1.

4.5 Advancing the Simulation

Press the ***step*** button four times to reach the 5th cycle. Instruction 15 has a ¬ symbol in the **EX** pipeline stage. Click on the **EX** pipeline stage on top of instruction 15 and observe the help explanation in the instruction view window. Figure 10 shows that situation. Note that instruction 15 has a RAW dependence with instruction 14 on register R2. However, since instruction 14 is in the **MEM** stage and has completed its operation (it is an integer operation, not a load instruction), the result that must be written to register r2 in the next cycle may be forwarded to instruction 15 using a bypass path. From now on, associate the symbol ¬ to a bypass. We can always click on the instruction to get help in the instruction

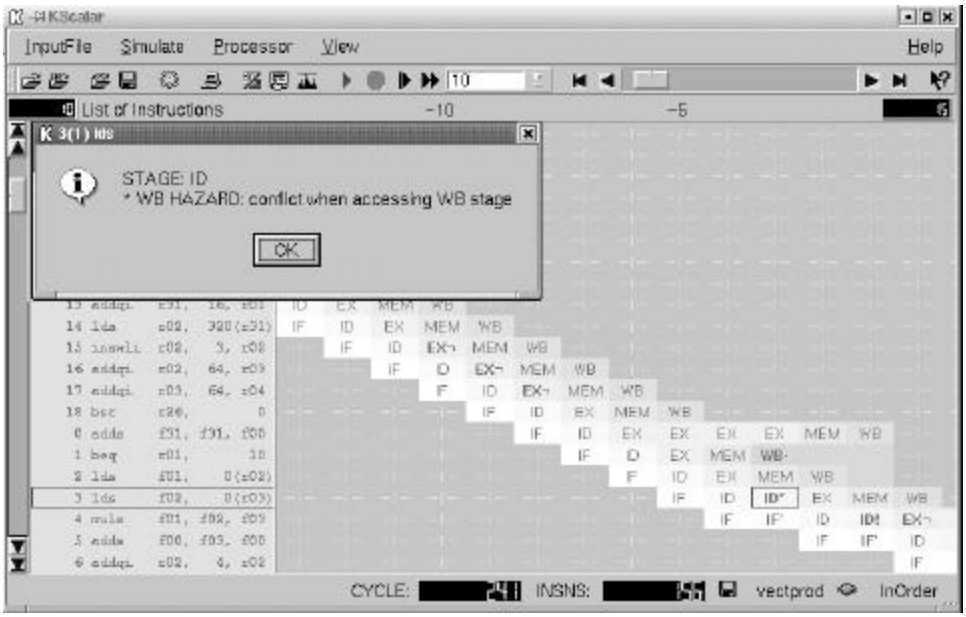


Fig. 11. Cycle diagram view. Instruction 3 (selected) is stalled due to a WB conflict. Issuing instruction 3 to EX on cycle 12 would have created a conflict with instruction 0 at the WB stage, since there is only one register file write port.

view window. We may try getting help by clicking on the help button and then on symbol !, in the instruction view window (bottom-left of the window).

Before advancing simulation, let's try to navigate backward through the previous simulated processor states. We may use the horizontal sliders (in the tool bar), the buttons next to the slider, or hotkeys F5 and F6. First, we must say a word to explain how the counters on the screen must be interpreted. The status bar always shows the counters for the last simulated cycle and for the last retired instruction (*retired* means that the instruction has left the pipeline). On the other side, the counters on the sliders show the cycle and instruction being visualized at this moment.

Notice that the active list contains all the instructions currently in the pipeline in instruction-fetch order. Moving through the active list or selecting instructions, modify the counter on the top-left of the active list, but do not modify the top-right counter (on top of the pipeline view) nor modify the pipeline contents. The counter on top of the active list is a global instruction counter, i.e., count instructions as they enter the pipeline.

Pressing on the top-right slider, though, modifies the visualized cycle, so the pipeline contents change. It permits repeating the sequence of execution backwards and forwards.

Advance to the 12th cycle and then advance to the 14th cycle. We can find another two symbols, * and !. Try getting help on instructions 3 and 4 to understand the reasons for the pipeline stalls.

Fig. 12. Statistics dialog. The "installed" statistics count the total penalties for each type of pipeline stall.

4.6 Cycle Diagram View

On the *View* menu, deselecting the option ***Toggle Pipeline View*** (or clicking the toggle button or pressing CTRL-T), will switch us to the cycle diagram view. It will show execution information along time. Figure 11 shows all the information from cycle 1 to cycle 14. In this mode we cannot get visual information about the instructions (we must remember the meaning of symbols). However, we may select an instruction (as in Figure 11), and then switch to the pipeline view to get information.

We press F8 to execute the program continuously. This hotkey invokes the ***run*** command. We can stop a simulation run by pressing F9 -or clicking on the ***stop*** button.

In this case, however, we should let the simulation run until the instruction count reaches 156 and the simulation is stopped automatically. At this stage, the pipeline contains only a *syscall* instruction. It has generated a system call with parameter 0, which indicates an exit program system call. We should now navigate backward through the cycle diagram to analyze the execution of the last instructions. We should go at our own speed, but be sure that we understand all the information shown in both views.

4.7 Statistics Wndow

We now examine the statistics window. We try to open the statistics window by invoking the ***Execution Statistics*** command in the *View* menu, and find that it doesn't

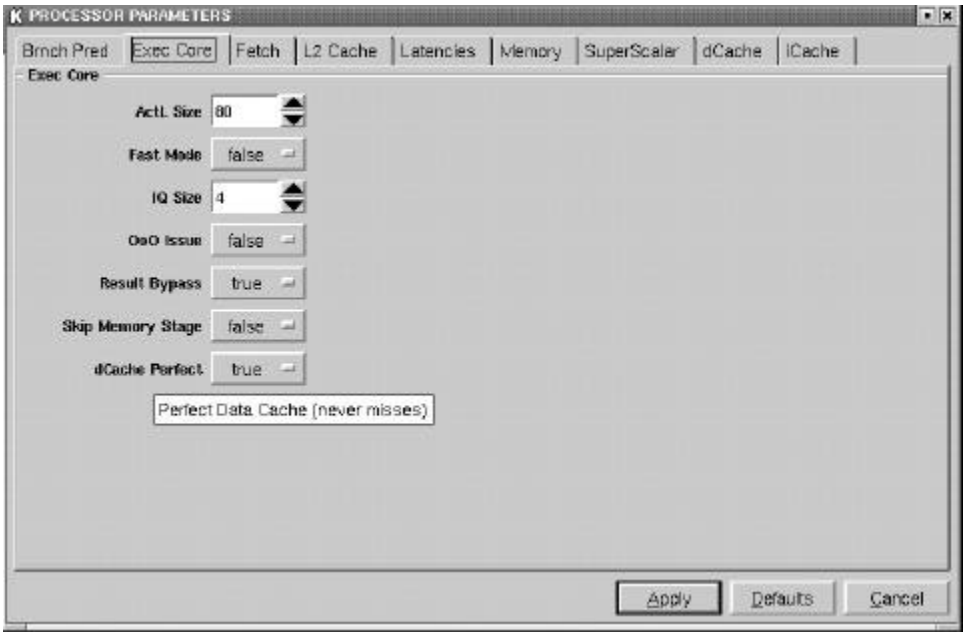


Fig. 13. Options dialog. The result bypass option is enabled on the execution core.

work! In fact, statistics must be installed each time we want to start taking measures on a program execution. We may also select which statistics are installed (those we are in at interested that moment). The procedure to install statistics is very simple.

First, we must reset simulation (CTRL+R), and then invoke the ***Execute Command File*** on the *Simulate* menu and select the file *inOrderStats.cmd*. If we press *OK*, the command file will be executed. If we invoke the *Execution* statistics command again, we will see the statistics window with all of them set to zero.

We now run the program in fast mode. We must modify the fast-forward step from 10 to 1000 and press the fast-forward button (or SHIFT+F7). The simulation stops when the instruction reaches 156. If we toggle to the cycle diagram view, we find a strange diagram (fast-forward simulation does not store simulation history!)

If we invoke the *Statistics* command now, we will see a list of stall counters with updated information (a snapshot of the statistics screen is shown in Figure 12). The "installed" statistics count the total penalties for each type of pipeline stall. The penalties only occur in the ID stage, creating a bubble in the pipeline.

Statistics are extremely useful in comparing the effects of change on the microarchitecture configuration. Let us try this now: examine the effects of operand bypassing in the example. We have used this feature until now; but what would execution time be without bypassing?

Let's take a look at the configuration dialog. It describes all the processor options and allows modifying the configuration. Figure 13 shows the *Execution Core* options. To disable bypassing, we click on the *Result Bypass* checkbox. When we select apply, the simulation is restarted. We must not forget to install the in-order statistics again (by

loading and executing the *inOrderStats.cmd* file) and run the whole simulation again with SHIFT+F7.

By re-examining the statistics window, we learn that the number of WB stalls and Trap stalls remained the same, but the number of RAW stalls was now 134 instead of 64, thus increasing the total number of simulation cycles from 241 to 311. With this information we can calculate, for example, the speedup gained by forwarding (311/ 241 = 1.290), which indicates that the processor with forwarding is 29 % faster than the processor without forwarding with program *vectprod*.

4.8 Further Experiments

This tutorial hurried through the example somewhat, due to the necessity of showing all the important features of the simulator. But understanding pipelining in general and a processor microarchitecture's mode of operation in particular can only come if we work through this and other examples in greater detail and at a suitable speed.

KScalar is able to simulate a large range of processor configurations. We may explore the configuration options and refer intensively to the help facilities, and explore which processor parameters have more influence in the performance of this and other examples. Furthermore, we can simulate the effects of an optimizing compiler by rearranging lines in the source codes, thus avoiding RAW stalls and WB stalls. Many details, which could not be answered in this tutorial, can be found by using the configuration and statistics files. In general, we should "play" with the simulator to get a "feeling" for the function of pipelining, and hope that KScalar will be a means to accomplish this.

5. COMMENTS TO THE INSTRUCTORS

KScalar is not intended as a tool to teach processor architecture autonomously; but it is necessary in order to explain the basic concepts and details of the microarchitecture being simulated. Instructors may use KScalar as a tool to demonstrate examples of execution stalls, students may then analyze alternative program fragments.

The goal of the simulator is neither to understand how Alpha AXP instructions behave nor to help develop applications in the ISA. The tool only provides the semantic information needed to explain the causes of some pipeline stalls, for instance, the type of operation, the input and output registers, and the accessed memory address.

There are several uses for the simulator:

1. It may be used to provide demonstrations in lectures or online learning environments. The examples may be enough to show the main characteristics of superscalar, pipelined processors, including caches, and branch prediction.
2. Students may investigate the characteristics of specific processor microarchitectures as practical short assignments associated with a lecture course. This may involve analyzing the performance of several program benchmarks for particular processor parameters.
3. Students may undertake major projects involving the optimization of real programs at the software-hardware interface, or involving the optimization of a processor microarchitecture for a given application workload.

Apart from the program fragments, the SPEC2000 Alpha binaries are available at <http://www.simplescalar.org> if a SPEC license is obtained to get the program inputs. Short programs can be created by assembling an EIO trace (the example was done using this strategy). However, we have an ongoing project to support on-the-fly assembling and reordering of instructions. We plan to provide this facility very soon (refer to *http://www.caos.uab.es/kscalar*).

Appendix B explains in detail the mechanisms to create new statistics and simulation experiments. Interested users should use the examples provided as a guide, and prepare simulation sessions with care. We are preparing a larger set of statistics for superscalar and out-of-order execution, which will also be available at the KScalar Web page.

6. FUTURE DEVELOPMENT

A preliminary version of KScalar has been used successfully in several lecture courses during the last two years at our University, and it is being used now at two other universities. During this time, apart from detecting many bugs, students have suggested many interesting improvements; some of which are being developed currently and some will be in the near future.

There are three ongoing projects:

1. ISA-level simulator: support for assembling and reordering instructions on the fly; enhanced graphical help on the ISA; existing support for creating EIO traces from binaries will be integrated in the graphical interface; static analysis of program dependencies.
2. Enhanced trace-support for the ISA-level simulator and the processor-level simulator; breakpoints and watch-points in both program-related events and in processor-related events. For example, analyzing why a given conditional branch is frequently mispredicted may require tracing particular entries on the branch prediction tables and identifying conflicting conditional branches. Detailed cache and branch prediction visualization modules will allow the selective tracing of table contents (watches on specific cache lines or branch prediction entries).
3. Graphical statistics package: Support to create new statistics integrated on the graphical interface; configurable chart diagrams; generating graphical data while running the program; generation of result tables to be processed with a spreadsheet.

Additionally, there are future development lines that are under consideration. First, we plan to integrate the new ISAs supported in the new SimpleScalar 4.0 release on our graphical tool (i386, ARM). Second, we plan to extend our simulation tool to allow configuring the processor microarchitecture using a real graphical interface, including dynamic control of restrictions for the configuration parameters. Finally, we want to enhance the simulated processor's model with register renaming, restrictions on the execution bandwidth for each operation type, trace-only and warm-up simulation mode, and so on.

APPENDIX A: INSTALLATION GUIDE AND TECHNICAL INFORMATION

A.1.General Software Description

KScalar runs on x86/Linux systems under a KDE environment. It is composed of two parts: a graphical interface and a text-based simulator engine.

The graphical interface was developed using KDevelop 1.2, a C++ visual language that uses the KDE (version 1.1) and QT (version 1.4) graphical libraries. Unfortunately, KDE 2.X and QT 2.X introduce substantial changes and are slightly backward incompatible. We are currently working on a new version that uses the new libraries.

The simulator engine was developed using prerelease 3.0 (May 2000) of the SimpleScalar tool set (*www.simplescalar.org*). The SimpleScalar tools are used to develop execution-driven simulators. They avoid generating, storing, and reading instruction trace files, and permit modeling control and data mis-speculation in a performance simulator. Our implementation has introduced substantial changes, both in the simulation engine and the simulator environment. Appendix B describes the added functionality in depth.

The graphical interface communicates with the text-based simulator by capturing the standard input and the standard output streams. It converts commands generated in graphical mode to text commands that are sent to the simulator engine. The simulator answer is parsed and information is translated to a graphical view.

Most of the microarchitectural-related issues are obtained from the simulation engine on the fly, and the graphical interface adapts to that information. This issue permits making substantial modifications to the text-based (SimpleScalar-like) simulator without needing to recompile the graphical interface. For example, the number of pipeline stages, the different type of stalls with their tip descriptions and help messages, all the simulation options and their default values, etc. are all obtained from the text-based core dynamically. In fact, options and statistics may be deleted and created during the simulation run.

KScalar was tested by running the SPEC2000 benchmarks to completion. We have used the binaries provided by Christopher T. Weaver in the SimpleScalar Web (*http://www.simplescalar.org*/). The test, though, only assures functional correctness (the benchmarks do what they have to) but not performance correctness (if the execution time is correct under the simulation parameters). Performance tests are always difficult to do. Only an accurate analysis by several people could guarantee correctness to some degree. We have validated our program with our students, but surely some little bugs are still hidden.

We tested KScalar on several Linux distributions: Red Hat versions 6.1, 7.1, and 7.2, Mandrake 7.0 and Suse 6.X and 7.0. For very new distributions (Red Hat 7.1/7.2), we have had to relink some shared libraries to make the program work (see the installation problems section below).

A.2.Installing the Software

There are three distribution files:

kscalar.tgz	– simulator suite (binaries and examples)
ksource.tgz	– sources for the graphical interface
kcore.tgz	– sources for the simulator engine

Copy the *kscalar.tgz* file into the root directory. To decompress the file, we need to run the ***gunzip*** (GNU) decompress utility and then the ***tar*** command. We may do both in one step with the command: *ROOT/> tar -xzvf kscalar.tgz.* We should now have the following files in our target directory:

KScalar	*– Directory containing all the binaries and examples*
KScalar/Kscalar	*– Graphical simulator interface binary (you may execute this file)*
KScalar/Kcore	*– Text-Based simulator core executable*
KScalar/bin	*– Should contain sample Alpha executables (it is empty)*
KScalar/eio	*– Contains sample EIO traces of Alpha binaries*
.../vectprod.eio	*– Tiny object program example (used in the tutorial)*
.../go.eio	*– SPEC95 go program (EIO trace of 100 million instructions)*
KScalar/cnf	*– Contains sample processor configuration files*
.../inOrder.cnf	*– In-order processor, perfect mode, issues one instruction per cycle*
.../inOrderLat1.cnf	*– In-order processor, perfect mode, all execution latencies set to 1*
.../inOrderBp.cnf	*– In-order processor, perfect mode but models branch prediction*
.../inOrderDC.cnf	*– In-order processor, perfect mode but models Data Cache*
.../inOrder2.cnf	*– In-order processor, perfect mode, two instructions per cycle*
.../inOrder4.cnf	*– In-order processor, perfect mode, four instructions per cycle*
.../outOrder.cnf	*– Out-of-order processor, perfect mode, one instruction per cycle*
.../outOrder2.cnf	*– Out-of-order processor, perfect mode, two instructions per cycle*
.../outOrder4.cnf	*– Out-of-order processor, perfect mode, four instructions per cycle*
KScalar/cmd	*– Contains command files used to create execution statistics*
.../inOrderStats.cmd	*– Creates statistics for accounting stalls in an in-order processor*
.../dCacheStats.cmd	*– Creates statistics for accounting data cache misses*
.../iCacheStats.cmd	*– Creates statistics for accounting instruction cache misses*
.../brpredStats.cmd	*– Creates statistics for accounting branch mispredictions*

A.3. Problems in Running the Program

To run the program we must be on the KDE graphical environment, and must invoke the graphical interface from its directory: *./Kscalar* (we use the ./ before the name of the executable to override default paths). Unfortunately, some problems may occur. We will comment the most common ones and provide solutions.

1. ***The graphical interface opens but the simulator core is not found.*** The *Kcore* binary must be found in the directory where the *Kscalar* program is invoked. If necessary, we set the appropriate path. We also check that there is permission to execute both programs (set permission with *chmod +x Kcore Kscalar*).

2. ***The graphical interface does not open and complains that a shared library cannot be found.*** The Kscalar binary needs the KDE libraries, version 1.1, the QT libraries, version 1.44, and the *libstdc++* libraries. They have to be installed in the system. For very new Linux distribution, the old QT and KDE libraries are not installed by default.

First, we must check that the libraries are not really present in the system. To do this, we may use the command "ldconfig –v". It shows a list of all the system libraries. If a newer version of the requested library exists, a link may be created to connect the name of the older library to the new library. However, since the QT and KDE versions 2.X are

not compatible with version 1.X, this solution does not work. In this case we must install the older libraries from the installation media. We will describe the steps required to install the libraries for the Red Hat distribution of Linux, version 7.1/7.2. For other Linux distributions, the steps should be very similar. Older Linux distributions should work with no problems.

We first mount the Red Hat install CD number 2. We then enter the package manager in the KDE graphical environment (it is necessary that it be installed, and is found on the system menu). We then select the *open a package file* menu option and navigate through the directory *mnt/cdrom/redhat/RPMS/* to find the packages **qt1X** and **kde1-compat**. We select those files and the option install. We must check the options *upgrade + replace files + check dependencies*. An error message window appears, but the installation is done. We then check that all is well by navigating through the package manager. The **qt1X** and **kde1-compat** packages should be listed bellow the options **system Environment/libraries**.

There is still one step left to do. It is possible that some shared libraries have still not been found They should be found in the directory </usr/lib>, but they may be newer versions. All we need to do is to create links to the newer versions. The following command will work for Linux 7.1/7.2:

/usr/lib/> ln -s libstdc++-libc6.2-2.so.3 libstdc++-libc6.1-2.so.3

Since the simulator binary provided looks for version 1.2 of the *libstdc++* library, this command creates a link to version 2.2.

A.4.Compiling the Software

In many cases, recompiling the graphical interface will be a cleaner solution than creating links to newer libraries. However, recompiling in a system that contains the newer, backward incompatible QT 2.X or KDE 2.X libraries could be difficult, since the autoconfigure utility cannot find the older versions of QT and KDE. In this case it is necessary to change the QT_PATH variable to point to the correct (older) path.

In order to recompile the graphical interface, we must first copy the *ksource.tgz* file to the root directory and then execute the following commands:

ROOT/> tar -xzvf ksource.tgz
ROOT/> cd sscalar-0.1
ROOT/sscalar-0.1> ./configure
ROOT/sscalar-0.1> cd sscalar
ROOT/sscalar-0.1/sscalar> make

We should now have generated a file named *sscalar*, which is the graphical simulator interface binary. We copy the *kcore.tgz* file to a directory */Kcore*, created on the root directory, and then decompress the file and compile it with the following commands:

ROOT/> cd Kcore
ROOT/Kcore/> tar -xzvf kcore.tgz
ROOT/Kcore/> make

We should now have generated a file named *Kcore*, which is the text-based simulator binary. We can test it with the following:

```
ROOT/Kcore/> ./Kcore
Dlite! > help
Command: help {name}
.....
```

A.5.More Information

KScalar is maintained by Juan C. Moure López.
Questions and comments can be sent to <*juancarlos.moure@uab.es*>.
Updated information can be obtained at <*http://www.caos.uab.es/kscalar*>.
For more information about SimpleScalar, visit the SimpleScalar Web page at *http://www.simplescalar.org*.

APPENDIX B: KCORE

The Kcore text-mode simulator is based on the SimpleScalar 3.0 tools. It contains an embedded lightweight symbolic interpreter (DLite!) that parses and executes simulation commands. Since Kcore has been adapted to communicate directly with a graphical front-end, all the options that could be specified in the command line are also accessible from the command interpreter. We have modified Kcore so that DLite! is invoked automatically and permits the execution of simulation commands from a file.

Conversely, the facilities supporting access to the processor architectural state (registers and memory) have been cleared. The goal of our simulator is not to teach issues relative to the Alpha ISA or develop correct assembler programs, nor to prove how the speedup techniques used in modern processors, like out-of-order execution, register renaming and speculation, are implemented to preserve correct processor operation. Since we are only interested in performance issues, providing the actual contents of memory and registers to show the precise semantics of each instruction is out of our scope.

B.1 State Variables

There are several built-in state variables that can be inspected using DLite! commands. State variables may be combined with a wide range of operators to form expressions. These expressions can be used to compute complex execution statistics and generate pretty-printed lists of results.

State variables describe the state of the simulated processor. Some of them are present in all simulated processor models while other are particular to a given processor model. Additionally, it is possible to create new integer, float or string variables that can be used to compute intermediate values or hold temporal information. A formula variable is assigned a string that defines an expression including other variables or constants. A formula variable is evaluated dynamically each time its value has to be computed and allows replacing long formulas by a simple name.

The next formulas show the syntax of valid expressions. It is very similar to the C language syntax, including array indexing ***[]***, a ternary conditional operator ***?:***, and an

special ***def*** operator that evaluates as TRUE (non-zero) if the following identifier is a defined variable.

```
<const>       := <dec_const> | <hex_const> | <oct_const> | <float_const>
<dec_const>   :=  (1-9)[0-9]*
<hex_const>   :=  0x[0-9,a-f]+
<oct_const>   :=  0[0-7]*
<float_const> :=  <dec_const>.[0-9]*  |  [0].[0-9]*
<ident>       :=  <variable-name>  |  <option-name>
<term>  := (<expr>)| (- | ~)<term>  |  <const>  | <ident>['[''<expr>']']
<factor>:= <term>  | <factor>(* | / | % | & | '|')<term>
<sum>   := <factor>  |  <sum>( + | - )<factor>
<cond>  := <sum>|!<cond>|def<term>|<sum>( = | < | > | <= | >= | != )<sum>
<expr>  := <cond> | <expr>(&& | "||")<cond> | <expr>?<sum>:<sum>
```

B.2 Information and File Commands

The following commands describe how to get help, how to load an object program, and how to create an EIO trace at any point of the simulation. An EIO file created before the simulation has started may be used as a checkpoint to rapidly start the execution of the program at a particular point, instead of restarting the simulation from the beginning. The creation of EIO and checkpoint files and the use of checkpoint files are done using special simulator commands.

help [<name>] print a description of a simulator command, state variable, or simulator option. By default lists all commands

quit exit simulator

version print simulator version

print [/<modifiers>] <expr> print the value of the expression using optional modifiers

prints <string> print a string without evaluation. If the string has to include blanks, then it must be double quoted ("…."). Special characters can be specified inside the string using '\'

load <string>|<file> load a program with its arguments specified by *string* (double quoted) or an EIO file. The command reconfigures and restarts the simulated processor

loadchk <file> load a checkpoint file and forward execution of current program to the checkpoint. The command reconfigures and restarts the simulated processor

makeeio <file> start the generation of a program trace from the current execution point. The trace ends when the program execution is completed, the simulator ends, or another program is loaded

makechk <file> create a checkpoint file at the current execution point. It is assumed that a trace is being used for execution or a trace is being generated.

B.3 Execute Commands

Execute commands are used for controlling the execution of a program. An execution interval is specified using simulated cycles or executed instructions. It is important to note that in a processor using out-of-order execution, an instruction is not considered complete until it is retired, i.e., until all previous instructions have been executed and retired. Also, several instructions may be retired in the same cycle.

step [<expr>] simulate the execution for an interval of *expr* cycles (1 cycle by default)

stepi <expr1> [<expr2>] simulate the execution for an interval of *expr1* instructions. Optionally, *expr2* sets a maximum on the number of simulated cycles for the execution interval

cont continue execution until completion of the program or until a specific processor breakpoint event is found

B.4 Option Commands

All the simulation options described so far are general options that control the operation of the simulator. There are however, options that are used to specify a particular configuration of the simulated processor, for example, to indicate the size of the internal cache, or the number of execution units. These kind of options are called *configuration options* and, as any other options, may be included into a configuration file. Of course, configuration options are specific to a given processor model. Since the name of the options and their description are accessible from the graphical interface we only describe here the commands used to handle options.

Configuration options, like state variables or defined variables, can be inspected and modified interactively. This issue is extremely useful when designing a large simulation experiment, since it permits to redefine the simulated processor on the fly and take performance measures over a range of processor parameters.

OPTS print the name of all simulation options

dumpopt [<file>] print the name and value of all simulation options in a form that can be used as a configuration file (optionally dump information to a *file*)

config [<file>] reconfigure the processor microarchitecture using the currently specified options. The processor memory and the internal registers are initialized. An optional configuration file may be used to modify a sequence of options It is necessary to use the *config* command for these changes to take effect in the simulator.

B.5 Control Commands

There are three simple control commands (*if*, *while*, and *for*) to conditionally invoke another built-in command or a sequence of commands stored in a command file. Command files contain simulation commands and are expected to have the extension *.cmd* and to be found in the directory *cmd*. The recursive invocation of command files can be nested up to a certain depth (implementation dependent). Command files can include comments by starting a line with the character #.

The provision of simple control commands, powerful expression syntax, and operations to create new variables provides a very simple script language for designing simulation experiments and visualizing simulation statistics. This issue is very useful for simulating large programs and/or obtaining results for varying processor parameters.

def_int <*name*> <*expr*>*[*<*nelem*>*]* create an integer variable (or array of *nelem* elements) called *name*, initialized with value *expr*

def_float <*name*> <*expr*>*[*<*index*>*]* create a float variable (or array of *nelem* elements) called *name*, initialized with value *expr*

def_str <*name*> <*string*> create a string variable called *name* initialized with *string*

do <*file*> execute command file unconditionally

exit exit current command file

if <*expr*> <*command*> evaluate the integer expression *expr* and, if it is TRUE (non-zero), then execute *command* (it may be a command file)

while <*expr*> <*command*> evaluate the integer expression *expr* and, if it is TRUE (non-zero), then execute *command* (it may be a command file) and repeat the execution of the *while* command

for <*expr*> <*command*> execute *command* (it may be a command file) as many times as the value provided by the integer expression *expr*

When specifying a command as an argument to another command, it may be necessary to double quote ("…") the command. Special characters can be specified inside the string using the character '\'. Example: `if var "prints \"not found\""`. If the value of variable *var* is different than zero, then the command `prints "not found"` is executed. Note how the use of the character `'\'` has allowed the inclusion of the character `"` into the specification of the second argument of *if*.

B.6 Generating Statistics

The simulator provides extensive support for taking statistics of selected variables. It is possible to dynamically select one or several variables (either simple state variables or

complex formula variables) to be sampled cycle-by-cycle during the simulation of a program in order to obtain a distribution of values, or Little's law queue statistics.

The following list presents all the commands related with the creation of statistics. Look at the examples provided with the distribution to better understand its use and feel free to create new statistics.

vars list name of all state variables

stats list name of all statistics

set <*name*> <*expr*> [<*index*>] assign value *expr* to variable *name* (optionally indexed by *index*)

def_state <*name*> <*definition*> <*state-class*> <*description*> create a state variable called *name*, which will be evaluated as the formula *definition*

def_stat <*name*> <*definition*> <*statistic-class*> <*description*> create a statistic variable called *name*, which will be evaluated as the formula *definition*

def_statdis <*name*> <*stat-name*> <*statistic-class*> <*description*> <*number-of-buckets*> <*bucket-size*> <*init-value*> create a statistic called *name*, which stores the distribution of values taken by variable *stat-name*, classifying values into buckets of an specified size and starting from the value *init-value*

def_little <*name*> <*statistic-class*> create a statistic variable that stores the minimum and maximum values of variable *name*, called *name_min* and *name_max*, and the sum of values and the sum of the squares of the values called *name_sum* and *name_sqr*

del <*name*> delete variable *name*

B.7 Program-Related Commands

The syntax of the per-instruction information provided by the *list* command is very complex, since it provides lots of information that the graphical interface will parse to generate a user-friendly view. For example, it provides the complete instruction description, its state, its operands, its dependencies, and the functional unit requirements.

The break command is very useful for skipping the execution of initialization code or large loops. Currently, there is no access to that command from the graphical interface (we must use the expert command dialog). Breakpoints will be included in the next release of KScalar.

list list the state of all the instructions (in program execution order) hold into the processor pipeline

dis <*PCtag*> [<*count*>] disassemble *count* consecutive instructions (in memory order) starting from the specified instruction (default is 20 instructions)

break <*PCtag*> set breakpoint at instruction *id*

B.8 System Calls and Memory layout

System calls are managed by a proxy handler. It intercepts system calls made by the simulated binary (using the CALL_PAL syscall instruction), decodes the system call, copies the system call arguments, makes the corresponding call to the host's operating system (Linux), and then copies the results of the call into the simulated program's memory.

The Alpha architecture uses a 43-bit address space. The text segment starts at the address 0x120000000, the data segment starts at the address 0x140000000, and the stack base address (grows down) is 0x11ff9b000.

Systems calls are initiated with the *CAL_PAL syscall* instruction. Prior to executing the instruction, register **v0** (r1) should be loaded with the system call code. The arguments of the system call interface prototype should be loaded into registers **a0-a6** (r16-r22) in the order specified by the system call interface prototype. For example, the command: `read (int fd, char *buf, int nbyte)` loads the value 0x03 into register **v0**, loads the value *fd* into register **a0**, *buf* into register **a1**, and *nbyte* into register **a2**. The list of all system calls supported by the simulator with their system call code (syscode) and interface specification is the following.

EXIT: Exit process.	(0x01) void exit(int status);
READ: Read from file to buffer.	(0x03) int read(int fd, char *buf, int nbyte);
WRITE: Write from buffer to file.	(0x04) int write(int fd, char *buf, int nbyte);
OPEN: Open a file.	(0x05) int open(char *fname, int flags, int mode);
CLOSE: Close a file.	(0x06) int close(int fd);
CREAT: Create a file.	(0x08) int creat(char *fname, int mode);
UNLINK: Delete a file.	(0x0a) int unlink(char *fname);
CHDIR: Change process directory.	(0x0c) int chdir(char *path);
CHMOD: Change file permissions.	(0x0f) int chmod(int *fname, int mode);
CHOWN: Change owner/group.	(0x10) int chown(char *fname, int owner, int group);
BRK: Change process break addr.	(0x11) int brk(long addr);
LSEEK: Move file pointer.	(0x13) long lseek(int fd, long offset, int whence);
GETPID: Get process identifier.	(0x14) int getpid(void);
GETUID: Get user identifier.	(0x18) int getuid(void);
ACCESS: Accessibility of a file.	(0x21) int access(char *fname, int mode);
STAT: Get file status.	(0x26) int stat(char *fname, struct stat *buf);
LSTAT: Get file status.	(0x28) int lstat(char *fname, struct stat *buf);
DUP: Duplicate a file descriptor.	(0x29) int dup(int fd);
PIPE: Create a comm. channel.	(0x2a) int pipe(int fd[2]);
GETGID: Get group identifier.	(0x2f) int getgid(void);
IOCTL: Device control interface.	(0x36) int ioctl(int fd, int request, char *arg);
FSTAT: Get file descriptor status.	(0x3e) int fstat(int fd, struct stat *buf);
GETPAGESIZE: Get page size.	(0x40) int getpagesize(void);
GETDTABLESIZE: Get table size.	(0x59) int getdtablesize(void);
DUP2: Duplicate a file descriptor.	(0x5a) int dup2(int fd1, int fd2);

FCNTL: File control.	(0x5c) int fcntl(int fd, int cmd, int arg);
SELECT: Synchronous I/O multiplexing.	(0x5d) int select (int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
GETTIMEOFDAY: Get date/time.	(0x74) int gettimeofday(struct timeval *tp, struct timezone *tzp);
WRITEV: Write output, vectored.	(0x79) int writev(int fd, struct iovec *iov, int cnt);
UTIMES: Set file times.	(0x8a) int utimes(char *file, struct timeval *tvp);
GETRLIMIT: Get maximum resource consumption.	(0x90) int getrlimit(int res, struct rlimit *rlp);
SETRLIMIT: Set maximum resource consumption.	(0x91) int setrlimit(int res, struct rlimit *rlp);

APPENDIX C: LICENSING TERMS

The Kscalar graphical interface is free software; we may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the license, or (at one's option) any later version.

The software is distributed in the hope that it will be useful, *but without any warranty*; without even the implied warranty of *merchantability*, or *fitness for a particular purpose.*

License is hereby given to use, modify, and redistribute the software, in whole or in part, for any noncommercial purpose, provided (1) that the user agrees to the terms of this copyright notice, including disclaimer of warranty; (2) that this copyright notice, including disclaimer of warranty, is preserved in the documentation of anything derived from this software; and (3) that the person or group modifying this software notifies us regarding the modifications and makes those modifications available to other non-commercial parties. Any redistributor of this software or anything derived from this software assumes responsibility for ensuring that any parties to whom such a redistribution is made are fully aware of the terms of this license and disclaimer.

The Kcore text-simulator inherits the licensing terms of the SimpleScalar tools, which we replicate here:

> No portion of this work may be used by any commercial entity, or for any commercial purpose, without the prior, written permission of SimpleScalar, LLC (info@simplescalar.com). Nonprofit and noncommercial use is permitted as described below.
>
> 1. SimpleScalar is provided *as is*, with no warranty of any kind, express or implied. The user of the program accepts full responsibility for the application of the program and the use of any results.
> 2. Nonprofit and noncommercial use is encouraged. SimpleScalar may be downloaded, compiled, executed, copied, and modified solely for nonprofit, educational, noncommercial research, and noncommercial scholarship purposes provided that this notice in its entirety accompanies all copies. Copies of the modified software can be delivered to persons who use it solely for nonprofit, educational, noncommercial research, and noncommercial scholarship purposes provided that this notice in its entirety accompanies all copies.

All commercial use and all use by for-profit entities is expressly prohibit-ed without a license from SIMPLESCALAR, LLC info@simplescalar.com.

3. No nonprofit user may place any restrictions on the use of this software, including as modified by the user, by any other authorized user.
4. Noncommercial and nonprofit users may distribute copies of SimpleScalar in compiled or executable form as set forth in Section 2, provided that either: (A) it is accompanied by the corresponding machine-readable source code, or (B) it is accompanied by a written offer, with no time limit, to give anyone a machine-readable copy of the corresponding source code in return for reimbursement of the cost of distribution. This written offer must permit verbatim duplication by anyone, or (C) it is distributed by someone who received only the executable form, and is accompanied by a copy of the written offer of source code.
5. SimpleScalar was developed by Todd M. Austin, Ph.D. The tool suite is currently maintained by SimpleScalar LLC <info@simplescalar.com>. US Mail: 2395 Timbercrest Court, Ann Arbor, MI 48105.

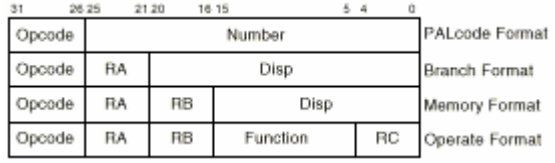
APPENDIX D. ALPHA IS A DESCRIPTION

Alpha is a 64-bit load/store RISC architecture designed to avoid bias toward any particular operating system or programming language. It facilitates pipelining multiple instances of the same operations because there are no special registers and no condition codes. Instructions only interact with each other by one instruction writing a register or memory and another instruction reading from the same place:

1. all operations are done between 64-bit registers;
2. memory is accessed via 64-bit virtual byte addresses using the little-endian byte numbering convention;
3. there are 32 integer registers and 32 floating-point registers;
4. longword (32-bit) and quadword (64-bit) integers are supported;
5. floating-point data types that are supported (among others): IEEE single (32-bit) and IEEE double (64-bit)

D.1 Instruction Format Overview

Alpha instructions are all 32 bits in length. There are four major instruction format classes that contain 0, 1, 2, or 3 register fields. All formats have a 6-bit opcode.



1. In the function code field, PALcode instructions specify one of a few dozen complex operations to be performed.
2. Conditional branch instructions test register Ra and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in register Ra.
3. Memory instructions (load and store) move bytes, words, longwords, or quadwords between register Ra and memory, using Rb plus a signed 16-bit displacement as the memory address.
4. Operate instructions for floating-point and integer operations are both represented by the same operate format and are as follows:
   – word and byte sign-extension operators;
   – integer and floating-point operations use Ra and Rb as source registers, and write the result in register Rc. There is an 11-bit extended opcode in the function field;
   – integer operate instructions can use the Rb field and part of the function field to specify an 8-bit literal. In this case, there is only one source register (specified by Ra), a source literal operand, and a 7-bit extended opcode in the function field.

Branch instructions: There are no branch condition codes. Conditional branch instructions can test a register for positive/negative or for zero/nonzero. They can also test integer registers for even/odd. Unconditional branch instructions can write a return address into a register. There is also a calculated jump instruction that branches to an arbitrary 64-bit address in a register.

Load/store instructions: Load and store instructions move 8-bit, 16-bit, 32-bit, or 64-bit aligned quantities from and to memory. Memory addresses are flat 64-bit virtual addresses, with no segmentation. A 32-bit integer datum is placed in a register in a canonical form that makes 33 copies of the high bit of the datum. A 32-bit floating-point datum is placed in a register in a canonical form that extends the exponent by 3 bits and extends the fraction with 29 low-order zeros. The 32-bit operates preserve these canonical forms.

Integer operate instructions: The integer operate instructions manipulate full 64-bit values, and include the usual assortment of arithmetic, compare, logical, and shift instructions. There are just three 32-bit integer operates: add, subtract, and multiply. They differ from their 64-bit counterparts only in overflow detection and in producing 32-bit canonical results. There is no integer divide instruction.

Floating-point operate instructions: The floating-point operate instructions include IEEE arithmetic instructions, plus instructions for performing conversions between floating-point and integer quantities. In addition to the operations found in conventional RISC architectures, Alpha includes conditional move instructions for avoiding branches and merge sign/exponent instructions for simple field manipulation. The arithmetic trap

enables and rounding mode are encoded in the function field of each instruction, rather then kept in global state bits. That makes it easier to pipeline implementations.
The Alpha architecture also supports the following additional operations:

1. scaled add/subtract instructions for quick subscript calculation;
2. 128-bit multiply for division by a constant, and multiprecision arithmetic;
3. conditional move instructions for avoiding branch instructions; and
4. an extensive set of in-register byte and word manipulation instructions.

The integer overflow trap enable is encoded in the function field of each instruction, rather than being kept in a global state bit. Thus, for example, both ADDQ/V and ADDQ opcodes exist for specifying 64-bit ADD with and without overflow checking. This makes it easier to pipeline implementations.

D.2 Alpha Registers

Program counter: The program counter (PC) is a special register that addresses the instruction stream. As each instruction is decoded, the PC is advanced to the next sequential instruction. This is referred to as the updated PC. Any instruction that uses the value of the PC will use the updated PC. The PC includes only bits <63:2> with bits <1:0> treated as RAZ/IGN. This quantity is a longword-aligned byte address. The PC is an implied operand on conditional branch and subroutine jump instructions. The PC is not accessible as an integer register.
Integer registers: There are 32 integer registers (R0 through R31), each 64 bits wide. Register R31 is assigned special meaning by the Alpha architecture. When R31 is specified as a register source operand, a zero-valued operand is supplied. For all cases except the unconditional branch and jump instructions, results of an instruction that specifies R31 as a destination operand are discarded. An exception is never signaled for a load that specifies R31 as a destination operation.
Floating-point registers: There are 32 floating-point registers (F0 through F31), each 64 bits wide. When F31 is specified as a register source operand, a true zero-valued operand is supplied. Results of an instruction that specifies F31 as a destination operand are discarded, and it is *unpredictable* whether the other destination operands (implicit and explicit) are changed by the instruction. An exception is never signaled for a load that specifies F31 as a destination operation. A floating-point instruction that operates on single-precision data reads all bits <63:0> of the source floating-point register. A floating-point instruction that produces a single-precision result writes all bits <63:0> of the destination floating-point register.

D.3 Addressing

The basic addressable unit in the Alpha architecture is an 8-bit byte. Virtual addresses are 64 bits long. An implementation may support a smaller virtual address space (the minimum size is 43 bits).
Byte: A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left, 0 through 7. A byte is an 8-bit value specified by its address. The byte is only supported in Alpha by the load, store, sign-extend, extract, mask, insert, and zap instructions.

Word: A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 15. A word is specified by the address of the byte containing bit 0. A word is a 16-bit value. The word is only supported in Alpha by the load, store, sign-extend, extract, mask, and insert instructions.
Longword: A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 31. A longword is specified by the address of the byte containing bit 0. A longword is a 32-bit value. When interpreted arithmetically, a longword is a 2's-complement integer with bits of increasing significance from 0 through 30. Bit 31 is the sign bit. The longword is only supported in Alpha by sign-extended load and store instructions and by longword arithmetic instructions.
Quadword: A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 63. A quadword is specified by the address of the byte containing bit 0. A quadword is a 64-bit value. When interpreted arithmetically, a quadword is either a 2's-complement integer with bits of increasing significance from 0 through 62 and bit 63 as the sign bit, or an unsigned integer with bits of increasing significance from 0 through 63.

D.4 IEEE Floating-Point Formats

Alpha architecture supports the basic single and double ANSI/IEEE 754-1985 formats.
S_floating: An IEEE single-precision, or S floating datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31. An S floating operand occupies 64 bits in a floating register, left-justified in the 64-bit register.

The S floating load instruction reorders bits on the way in from memory, expanding the exponent from 8 to 11 bits, and sets the low-order fraction bits to zero. In the register this produces an equivalent T floating number, suitable for either S floating or T floating operations. This mapping preserves both normal values and exceptional values. The S floating load instruction does no checking of the input.

The S floating store instruction reorders register bits on the way to memory and does no checking of the low-order fraction bits. Register bits <61:59> and <28:0> are ignored by the store instruction. The S floating store instruction does no checking of the data; the preceding operation should have specified an S floating result.

An S floating datum is specified by the address of the byte containing bit 0. The memory form of an S floating datum is sign magnitude with bit 31 the sign bit, bits <30:23>, an excess-127 binary exponent, and bits <22:0>, a 23-bit fraction.

Floating-point operations on S floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.
T-floating: An IEEE double-precision, or T floating datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63. A T floating operand occupies 64 bits in a floating register, arranged as shown in the figure below.

The T floating load instruction performs no bit reordering on input, nor does it perform checking of the input data. The T floating store instruction performs no bit reordering on output. This instruction does no checking of the data; the preceding operation should have specified a T floating result. A T floating datum is specified by the address of the byte containing bit 0. The form of a T floating datum is sign magnitude with bit 63 the sign bit, bits <62:52>, an excess-1023 binary exponent, and bits <51:0>, a 52-bit fraction.

Floating-point operations on T floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.

D.5 Alpha Instruction Set

This is a summary of all Alpha architecture instructions. Values are in hexadecimal. Table I describes the contents of the Format and Opcode columns in Table II.

Table I. Instruction Format and Opcode Notation

Instruction Format	Format Symbol	Opcode Notation	Meaning	Syntaxis
Branch	Bra	Oo	oo is the 6-bit opcode field	B*cc* Ra,displ
Floating-point	F-P	oo.fff	oo is the 6-bit opcode field fff is the 11-bit function code field	Op Fa,Fb,Fc Fa, Fb: input, Fc: output
Memory	Mem	Oo	oo is the 6-bit opcode field	LD/ST Ra,displ(Rb)
Memory/ function code	Mfc	oo.ffff	oo is the 6-bit opcode field ffff is the 16-bit function code in the displacement field	Special
Memory/ branch	Mbr	oo.h	oo is the 6-bit opcode field h is the high-order 2 bits of the displacement field	BSR/JMP/JSR/RET Ra, Rb, displ
Operate	Opr	oo.ff	oo is the 6-bit opcode field ff is the 7-bit function code field	Op Ra,Rb,Rc Ra,Rb: input, Rc: output
PALcode	Pcd	Oo	oo is the 6-bit opcode field; the PALcode instruction is specified in the 26-bit function code field	CAL_PAL function

Table II. ISA Instructions (part 1 of 5)

Mnemonic	Format	Opcode	Description
ADDF	F-P	15.080	Add F floating
ADDG	F-P	15.0A0	Add G floating
ADDL	Opr	10.00	Add longword
ADDL/V	Opr	10.40	Add longword (check overflow)
ADDQ	Opr	10.20	Add quadword
ADDQ/V	Opr	10.60	Add quadword (check overflow)
ADDS	F-P	16.080	Add S floating
ADDT	F-P	16.0A0	Add T floating
AMASK	Opr	11.61	Determine byte/word instruction implementation
AND	Opr	11.00	Logical product

Table II. ISA Instructions (part 2 of 5)

Mnemonic	Format	Opcode	Description
BEQ	Bra	39	Branch if = zero
BGE	Bra	3E	Branch if ≥ zero
BGT	Bra	3F	Branch if > zero
BIC	Opr	11.0	Bit clear (and ~)
BIS	Opr	11.20	Logical sum (or)
BLBC	Bra	38	Branch if low bit clear
BLBS	Bra	3C	Branch if low bit set
BLE	Bra	3B	Branch if ≤ zero
BLT	Bra	3A	Branch if < zero
BNE	Bra	3D	Branch if ≠ zero
BR	Bra	30	Unconditional branch
BSR	Mbr	34	Branch to subroutine
CALL_PAL	Pcd	00	Trap to PALcode
CMOVEQ	Opr	11.24	Conditional MOVE if = zero
CMOVGE	Opr	11.46	Conditional MOVE if ≥ zero
CMOVGT	Opr	11.66	Conditional MOVE if > zero
CMOVLBC	Opr	11.16	Conditional MOVE if low bit clear
CMOVLBS	Opr	11.14	Conditional MOVE if low bit set
CMOVLE	Opr	11.64	Conditional MOVE if ≤ zero
CMOVLT	Opr	11.44	Conditional MOVE if < zero
CMOVNE	Opr	11.26	Conditional MOVE if ≠ zero
CMPBGE	Opr	10.0F	Compare 8 unsigned bytes in parallel (multimedia)
CMPEQ	Opr	10.2D	Compare signed quadword equal
CMPGEQ	F-P	15.0A5	Compare G floating equal
CMPGLE	F-P	15.0A7	Compare G floating less than or equal
CMPGLT	F-P	15.0A6	Compare G floating less than
CMPLE	Opr	10.6D	Compare signed quadword less than or equal
CMPLT	Opr	10.4D	Compare signed quadword less than
CMPTEQ	F-P	16.0A5	Compare T floating equal
CMPTLE	F-P	16.0A7	Compare T floating less than or equal
CMPTLT	F-P	16.0A6	Compare T floating less than
CMPTUN	F-P	16.0A4	Compare T floating unordered
CMPULE	Opr	10.3D	Compare unsigned quadword less than or equal
CMPULT	Opr	10.1D	Compare unsigned quadword less than
CPYS	F-P	17.020	Copy sign to floating-point value
CPYSE	F-P	17.022	Copy sign and exponent to floating-point value
CPYSN	F-P	17.021	Copy sign negate (complement sign)
CVTDG	F-P	15.09E	Convert D floating to G floating
CVTGD	F-P	15.0AD	Convert G floating to D floating
CVTGF	F-P	15.0AC	Convert G floating to F floating
CVTGQ	F-P	15.0AF	Convert G floating to quadword
CVTLQ	F-P	17.010	Convert longword to quadword
CVTQF	F-P	15.0BC	Convert quadword to F floating
CVTQG	F-P	15.0BE	Convert quadword to G floating
CVTQL	F-P	17.030	Convert quadword to longword
CVTQL/SV	F-P	17.530	Convert quadword to longword (overflow & s/w completion)
CVTQL/V	F-P	17.130	Convert quadword to longword (check overflow)
CVTQS	F-P	16.0BC	Convert quadword to S floating
CVTQT	F-P	16.0BE	Convert quadword to T floating
CVTST	F-P	16.2AC	Convert S_floating to T floating

Table II. ISA Instructions (part 3 of 5)

Mnemonic	Format	Opcode	Description
CVTTQ	F-P	16.0AF	Convert T floating to quadword
CVTTS	F-P	16.0AC	Convert T floating to S floating
DIVF	F-P	15.083	Divide F floating
DIVG	F-P	15.0A3	Divide G floating
DIVS	F-P	16.083	Divide S floating
DIVT	F-P	16.0A3	Divide T floating
EQV	Opr	11.48	Logical equivalence (xor ~)
EXCB	Mfc	18.0400	Exception barrier (wait for exceptions on all previous insts)
EXTBL	Opr	12.06	Extract byte low (shift right)
EXTLH	Opr	12.6A	Extract longword high (shift left)
EXTLL	Opr	12.26	Extract longword low (shift right)
EXTQH	Opr	12.7A	Extract quadword high (shift left)
EXTQL	Opr	12.36	Extract quadword low (shift rigth)
EXTWH	Opr	12.5A	Extract word high (shift left)
EXTWL	Opr	12.16	Extract word low (shift right)
FBEQ	Bra	31	Floating branch if = zero
FBGE	Bra	36	Floating branch if ≥ zero
FBGT	Bra	37	Floating branch if > zero
FBLE	Bra	33	Floating branch if ≤ zero
FBLT	Bra	32	Floating branch if < zero
FBNE	Bra	35	Floating branch if ≠ zero
FCMOVEQ	F-P	17.02A	Floating Conditional MOVE if = zero
FCMOVGE	F-P	17.02D	Floating Conditional MOVE if ≥ zero
FCMOVGT	F-P	17.02F	Floating Conditional MOVE if > zero
FCMOVLE	F-P	17.02E	Floating Conditional MOVE if ≤ zero
FCMOVLT	F-P	17.02C	Floating Conditional MOVE if < zero
FCMOVNE	F-P	17.02B	Floating Conditional MOVE if ≠ zero
FETCH	Mfc	18.80	Prefetch data (aligned 512-byte block surrounding virt.addr.)
FETCH_M	Mfc	18.A0	Prefetch data, modify intent (aligned 512-byte block)
IMPLVER	Opr	11.6C	Determine CPU type
INSBL	Opr	12.0B	Insert byte low
INSLH	Opr	12.67	Insert longword high
INSLL	Opr	12.2B	Insert longword low
INSQH	Opr	12.77	Insert quadword high
INSQL	Opr	12.3B	Insert quadword low
INSWH	Opr	12.57	Insert word high
INSWL	Opr	12.1B	Insert word low
JMP	Mbr	1A.0	Jump
JSR	Mbr	1A.1	Jump to subroutine
JSR_ COROUTINE	Mbr	1A.3	Jump to subroutine return
LDA	Mem	08	Load address
LDAH	Mem	09	Load address high (displacement shifted 16 positions)
LDBU	Mem	0A	Load zero-extended byte
LDF	Mem	20	Load F floating
LDG	Mem	21	Load G floating
LDL	Mem	28	Load sign-extended longword
LDL_L	Mem	2A	Load sign-extended longword locked
LDQ	Mem	29	Load quadword
LDQ_L	Mem	2B	Load quadword locked

Table II. ISA Instructions (part 4 of 5)

Mnemonic	Format	Opcode	Description
LDQ_U	Mem	0B	Load unaligned quadword
LDS	Mem	22	Load S floating
LDT	Mem	23	Load T floating
LDWU	Mem	0C	Load zero-extended word
MB	Mfc	18.4000	Memory barrier (ensures memory access ordering)
MF_FPCR	F-P	17.025	Move from floating-point control register
MSKBL	Opr	12.02	Mask byte low
MSKLH	Opr	12.62	Mask longword high
MSKLL	Opr	12.22	Mask longword low
MSKQH	Opr	12.72	Mask quadword high
MSKQL	Opr	12.32	Mask quadword low
MSKWH	Opr	12.52	Mask word high
MSKWL	Opr	12.12	Mask word low
MT_FPCR	F-P	17.024	Move to floating-point control register
MULF	F-P	15.082	Multiply F floating
MULG	F-P	15.0A2	Multiply G floating
MULL	Opr	13.00	Multiply longword
MULL/V	Opr	13.40	Multiply longword (check overflow)
MULQ	Opr	13.20	Multiply quadword
MULQ/V	Opr	13.60	Multiply quadword (check overflow)
MULS	F-P	16.082	Multiply S floating
MULT	F-P	16.0A2	Multiply T floating
ORNOT	Opr	11.28	Logical sum with complement (or ~)
RC	Mfc	18.E0	Read and clear (VAX compatibility)
RET	Mbr	1A.2	Return from subroutine
RPCC	Mfc	18.C0	Read process cycle counter
RS	Mfc	18.F000	Read and set (VAX compatibility)
S4ADDL	Opr	10.02	Scaled add longword by 4 { (Ra*4 + Rb) → Rc }
S4ADDQ	Opr	10.22	Scaled add quadword by 4 { (Ra*4 + Rb) → Rc }
S4SUBL	Opr	10.0B	Scaled subtract longword by 4 { (Ra*4 - Rb) → Rc }
S4SUBQ	Opr	10.2B	Scaled subtract quadword by 4 { (Ra*4 - Rb) → Rc }
S8ADDL	Opr	10.12	Scaled add longword by 8 { (Ra*8 + Rb) → Rc }
S8ADDQ	Opr	10.32	Scaled add quadword by 8 { (Ra*8 + Rb) → Rc }
S8SUBL	Opr	10.1B	Scaled subtract longword by 8 { (Ra*8 - Rb) → Rc }
S8SUBQ	Opr	10.3B	Scaled subtract quadword by 8 { (Ra*8 - Rb) → Rc }
SEXTB	Opr	1C.00	Store byte
SEXTW	Opr	1C.01	Store word
SLL	Opr	12.39	Shift left logical
SRA	Opr	12.3C	Shift right arithmetic
SRL	Opr	12.34	Shift right logical
STB	Mem	0E	Store byte
STF	Mem	24	Store F floating
STG	Mem	25	Store G floating
STS	Mem	26	Store S floating
STL	Mem	2C	Store longword
STL_C	Mem	2E	Store longword conditional
STQ	Mem	2D	Store quadword
STQ_C	Mem	2F	Store quadword conditional
STQ_U	Mem	0F	Store unaligned quadword
STT	Mem	27	Store T floating
STW	Mem	0D	Store word

Table II. ISA Instructions (part 5 of 5)

Mnemonic	Format	Opcode	Description
STT	Mem	27	Store T floating
STW	Mem	0D	Store word
SUBF	F-P	15.081	Subtract F floating
SUBG	F-P	15.0A1	Subtract G floating
SUBL	Opr	10.09	Subtract longword
SUBL/V	Opr	10.49	Subtract longword (check overflow)
SUBQ	Opr	10.29	Subtract quadword
SUBQ/V	Opr	10.69	Subtract quadword (check overflow)
SUBS	F-P	16.081	Subtract S floating
SUBT	F-P	16.0A1	Subtract T floating
TRAPB	Mfc	18.00	Trap barrier (all prior insts are completed without arithmetic traps)
UMULH	Opr	13.30	Unsigned multiply quadword high (64 high-order bits of 128-bit result)
WMB	Mfc	18.44	Write memory barrier (control internal write buffers)
XOR	Opr	11.40	Logical difference
ZAP	Opr	12.30	Zero bytes (set selected bytes of register to zero)
ZAPNOT	Opr	12.31	Zero bytes not (set selected bytes of register to zero)

Table III. New ISA Instructions (multimedia)

Mnemonic	Format	Opcode	Description
CTLZ	Opr	1C.32	Count leading zero (number of leading zeros in Rb)
CTPOP	Opr	1C.30	Count population (number of ones in Rb)
CTTZ	Opr		Count trailing zero (number of trailing zeros in Rb)
ECB	Mfc		Evict cache block (hint: cache space can be reused)
FTOIS	F-P		Floating-point to integer register move, S floating
FTOIT	F-P		Floating-point to integer register move, T floating
ITOFF	F-P		Integer to floating-point register move, F floating
ITOFS	F-P		Integer to floating-point register move, S floating
ITOFT	F-P		Integer to floating-point register move, T floating
SQRTF	F-P		Square root F floating
SQRTG	F-P		Square root G floating
SQRTS	F-P		Square root S floating
SQRTT	F-P		Square root T floating
WH64	Mfc		Write hint: 64 bytes (cache block will never be read)
MAXSB8	Opr	1C.3C	Vector signed byte maximum (set max. byte in parallel)
MAXSW4	Opr	1C.3D	Vector signed word maximum (set max. word in parallel)
MAXUB8	Opr	1C.3E	Vector unsigned byte maximum
MAXUW4	Opr	1C.3F	Vector unsigned word maximum
MINSB8	Opr	1C.38	Vector signed byte minimum
MINSW4	Opr	1C.39	Vector signed word minimum
MINUB8	Opr	1C.3A	Vector unsigned byte minimum
MINUW4	Opr	1C.3B	Vector unsigned word minimum
PERR	Opr	1C.31	Pixel error (sum of absolute values of all byte differences)
PKLB	Opr	1C.37	Pack longwords to bytes (2 longwords → 2 bytes)
PKWB	Opr	1C.36	Pack words to bytes (4 words → 4 bytes)
UNPKBL	Opr	1C.35	Unpack bytes to longwords (2 bytes → 2 longwords)
UNPKBW	Opr	1C.34	Unpack bytes to words (2 bytes → 4 words)

ACKNOWLEDGMENT

Many thanks to Francisco Cruz and Francisco Perez., who developed the earlier prototypes of the graphical interface.

REFERENCES

BURGER, D. AND AUSTIN, T.M. 1997. The simple scalar tool set. Tech. Rep. TR-1342. Computer Science Dept., Univ. of Wisconsin, Madison.

EASYCPU. 2001. EasyCpu home page: http://www.cteh.ac.il/departments/education/cpu.htm.

HENNESSY, J. AND PATTERSON, D. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann, Los Altos, CA.

LC2. 2001. LC2 home page: http://www-personal.engin.umich.edu/~postiffm/lc2/lc2.htm.

LMC. 2001. Little Man Computer home page: http://www.acs.ilstu.edu/faculty/javila/lmc/home.htm.

MCFARLING, S. 1993. Combining branch predictors. Tech. Rep. TN-36, Digital Western Research Laboratory.

PATT, Y. AND PATEL, S. 2001. *Introduction to Computing Systems*. McGraw-Hill, New York, NY.

PATTERSON, D. AND HENNESSY, J. 1998. *Computer Organization and Design*, 2nd ed., Morgan Kaufmann, Los Altos, CA.

PEARSON, M.W., MCGREGOR, A.J., AND HOLMES, G. 1999. Teaching computer systems to majors: A MIPS based solution. *IEEE Comput. Soc. Comput. Architecture Tech. Comm. Newsl*., Feb. 1999, 22–24.

ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S.A. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1 (Jan.), 78-103.

RTLsim. 2001. RTLsim home page: http://www.cs.waikato.ac.nz/cs/Teaching/0657.201/Simulators. Simple Scalar. 2001. SimpleScalar home page: http://www.simplescalar.org/..

SPIM. 2001. SPIM home page: http://www.cs.wisc.edu/~larus/spim.html.

SALLINGS, W. 2000. *Computer Organization and Architecture*, 5th ed., Prentice-Hall, Englewood Cliffs, NJ.

TANENBAUM, A. 1999. *Structured Computer Organization*, 4th ed., Prentice-Hall, Englewood Cliffs, NJ.

WinDLX. 2001. WinDLX home page: ftp://ftp.mkp.com/pub/dlx/ftp.

YEHEZKEL. C., YURCIK, W., AND PEARSON, M. 2001. Teaching computer architecture with a computer-aided learning environment: State-of-the-art simulators. In *Proceedings of the International Conference on Simulation and Multimedia in Engineering Education* (*ICSEE 2001*, Phoenix, AZ, Jan.).

YURCIK, W., VILA, J., AND BRUMBAUGH, L. 2000. An interactive Web-based simulation of a general computer architecture. In *Proceedings of the International Conference on Engineering and Computer Education* (*ICECE 2000*, San Paulo, Brazil, Aug.).

Received November 2001; accepted February 2002.