The Liberty Simulation Environment as a Pedagogical Tool

Jason Blome Manish Vachharajani Neil Vachharajani David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
{blome, manishv, nvachhar, august}@cs.princeton.edu

Abstract

This paper describes how the Liberty Simulation Environment (LSE) and its graphical visualizer can be used in a computer architecture course. LSE allows for the rapid construction of simulators from models that resemble the structure of hardware. By using and modifying LSE models, students can develop a solid understanding of and learn to reason about computer architecture. Since LSE models are also relatively easy to modify, the tool can be used as the basis of meaningful assignments, allowing students to explore a variety of microarchitectural concepts on their own. In lectures where block diagrams are typically displayed, LSE's visualizer can be used instead to not only show block diagrams, but to demonstrate the machine in action. As a result, LSE can ease the burden of conveying complex microarchitectural design concepts, greatly improving the depth of understanding a computer architecture course provides.

1. Introduction

The goal of a computer architecture course is to teach students how microprocessor hardware operates and to give them an opportunity to experiment with microprocessor design. To do this, the students need to learn about existing microarchitecture design techniques ranging from simple concepts such as pipelining to advanced organizations such as out-of-order machines including features such as speculation, branch prediction, and register renaming.

Unfortunately for the student, the expanse of the computer architecture design space is vast and complex. Within a single design, each component plays an important role in facilitating the correct and efficient execution of a program, however, correct execution is only ensured when interaction of all the components is carefully orchestrated. These component interactions are often subtle making it difficult to convey enough information in lecture to foster a deep understanding.

To remedy this, courses are augmented with periodic assignments that encourage the students to discover some of the complexities on their own. Typically, these assignments involve drawing pipeline sketches and evaluating block diagrams. Unfortunately, these assignments do little to foster an understanding of the complex dynamic interactions and instead reinforce the students' understanding of the architecture's high-level structure.

An approach that leads to a much better understanding of a microarchitecture is to have the students design the hardware for a machine and run programs on it in a simulated environment. This way students will discover, on their own, the intricacies of how different parts of the architecture interact to facilitate the correct program behavior. The insight gained from this process of design, test, and debugging is often deeper than any knowledge gained in lecture. Unfortunately, specifying the hardware, even in a synthesizable hardware description language like VHDL or Verilog (as opposed to a gate level description), can take many weeks. As a result, when this method is employed, it is typically limited to a single course-length project. While a project of this type is better than no design experience, it only provides insights about the techniques employed in one particular design, which is often a small subset of the techniques taught in the class.

A promising alternative to specifying the hardware is using, building, and modifying higher-level simulation tools for the microarchitecture. Assignments could consist of asking students to modify a simulator to incorporate new behavior. Unfortunately, current simulation environments are too difficult to modify to make this practical for periodic homework assignments. Furthermore, the most common method of building a simulator (coding it by hand in C or C++) does little to convey the actual hardware structure or the functionality of its components [1]. Consequently the process of reasoning about and building the simulator is very different from the way in which a computer architect would design a microprocessor thus making it unsuitable as a pedagogical tool. Students should think like architects, not like simulator writers.

To be effective for use in assignments, the simulation system should have simulator descriptions that reflect the hardware being modeled. Components used in modeling should correspond to hardware blocks and they should be interconnected via communication channels like hardware blocks. On the other hand the simulation environment should not impose, upon the student, the burdens that hardware description languages like VHDL or Verilog often do. Instead, it should allow rapid construction and modification of models so that working with an executable model can be a part of

regular assignments.

LSE is a simulator construction tool that meets the requirements outlined above. In this paper we give an overview of the Liberty Simulation Environment (LSE) and describe how it can be used in a course to enhance student understanding of computer architecture. In the next section we describe the Liberty Simulation Environment. In Section 3 we describe the LSE visualizer and how it visualizes the LSE descriptions. Then, in Section 4 we give specific examples of how to use LSE in a course. Finally, we conclude in Section 5.

2. The Liberty Simulation Environment

The Liberty Simulation Environment (LSE) is an excellent tool for students to learn about and explore microarchitecture. LSE descriptions resemble the hardware they model and are easy to modify. This section describes enough about LSE so that it is possible to understand how LSE can be used for instructional purposes. Details of how LSE enables rapid specification while still resembling the modeled hardware as well as details of all the concepts described in this section can be found elsewhere [1].

As shown in Figure 1, LSE consists of three main parts: the Liberty Structural Specification Language (LSS), a component library, and the Liberty Simulator Constructor. To use the system, the user describes a machine by specifying, in the LSS language, a set of interconnected instances of components. These components, called *modules*, are typically taken from the module library although custom modules can be created if necessary. The user then invokes the simulator constructor, and the constructor reads the specification and the code from the module library and builds an executable simulator for the described machine. This section will discuss in more detail the properties of modules, module communication, and collection of data from a run of the simulator executable.

2.1. Modules

Each module is a parameterized template that is instantiated in an LSS machine description to create a *module instance* (or simply instance). Much like components in hardware design, modules can be leaves of a hierarchy, or they can be constructed hierarchically by grouping collections of other interconnected module instances. Like hardware blocks, module instances execute concurrently [2] and communicate with other instances by passing data across communication channels.

However, unlike hardware design, the details of instance behavior (hierarchical or leaf) can be customized via module parameters. When a user instantiates a module in a machine description, the user specifies values for the parameters declared by the module (or accepts the default value specified by the module). These parameters are used to customize the behavior of the module instance for the particular description. Parameters can control simple configuration options (e.g. the cache line size or whether a 2-level branch predictor has a global or per-address predictor table). Further, parameters can also be used to allow control of algorithms allowing users to customize complex behavior. For example, the branch predictor has a parameter that allows users to override the predictor state-machine code to implement a custom predictor if none of the provided predictor options is suitable. Parameters can even control the instantiation of hardware structures in lower levels of the hierarchy.

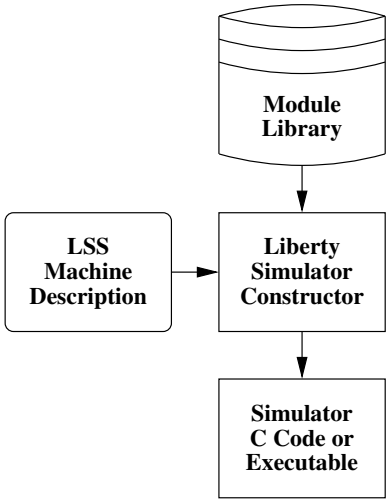

Figure 1: Overview of main LSE components.

2.2. Communication Channels

Modules specify a communication interface for module instances by declaring *ports*. Each instance may have one or more *port instances* per port. Each port instance is a communication channel and may have exactly one value sent on it per cycle. For example, the register file module may have an input port on which register read requests are made. Each port instance would accept one register read request per cycle. If two register reads per cycle were needed, there would be 2 port instances of the register read request port, and two instances of the output port on which these read requests were returned. Another example is the `tee` module which is used to fanout a given value to multiple receivers. The `tee` duplicates the value received on its input port instances on multiple output port instances.

Users specify how modules communicate by interconnecting port instances from one module to port instances on the same or other modules. While details regarding ports and the communication system can be found in [1], other work describes LSE's execution and messaging semantics [2].

2.3. Data Collection

To allow modularity and flexibility even for data collection, LSE provides a data collection mechanism that avoids

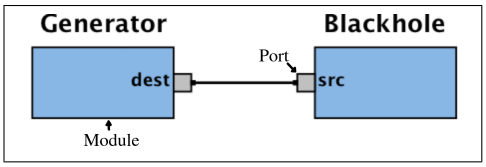

Figure 2: Simple source to sink description.

the pitfalls of embedding instrumentation code directly into the simulator code. Each LSE module may declare that its instances emit certain *events* during the execution of the simulator. Each event includes data related to the event and a dynamic identifier (dynID) that represents the system level object that caused the event to occur. Orthogonal to the declaration of events, users may associate, with any event, a *data collector* which captures the event and records data or computes statistics.

For example, a branch predictor module may emit an event every time it makes a prediction. The event could include information about what prediction was made and what predictor made that prediction. The dynID for the event would identify the dynamic instruction instance that caused the prediction to occur. A user could hook this event with a data collector to count the number of predictions made or to calculate a branch misprediction rate for example.

In addition to recording data or computing statistics, the LSE visualizer (described in Section 3) hooks these events to visualize the flow of data through the machine at runtime. An example of this is described in Section 4.

3. Visualization

The LSE Visualizer provides a means to view LSS descriptions as block diagrams. In addition, the LSE Visualizer provides an interface for compiling an LSS design into an executable simulator, and it provides tools for observing, via animation, the execution of the simulator. In this section we will describe the visualizer in enough detail so that it is possible to understand the discussion of how the visualizer can be used in a computer architecture course as described in Section 4.

Figure 2 is an LSE Visualizer screenshot showing the block diagram of a simple system. In this system, a module instance, called Generator, sends data to another module instance, called Blackhole, which discards it. Generator is an instance of the `datasource` module, and Blackhole is an instance of the `sink` module. Both the `datasource` and `sink` modules are provided by the LSE module library. During simulator execution, a unit of data will be transfered from Generator to Blackhole in each and every cycle until the simulation is terminated manually.

In Figure 2 the module instances are represented by the large boxes, while their ports are represented by the small boxes. The single line between the box labeled `dest` and the box labeled `src` represents a connection between the respective port instances on the module instances Generator and Blackhole.

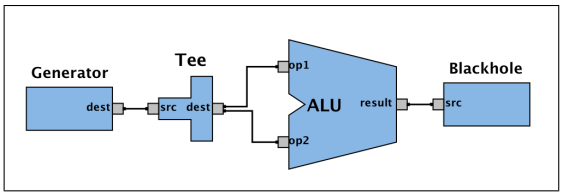

Figure 3: Simple $x + x$ description.

Figure 3 shows a screenshot of a slightly more sophisticated machine configuration. Here, the `datasource` module instantiated as Generator is connected to an instance of the `tee` module named Tee. The Tee in turn fans out the data originating from Generator into port instances of ports `op1` and `op2` of the instance ALU. ALU then computes the sum of the values passed into it on these port instances and sends the result to Blackhole to be thrown out. The function of this machine is simply to compute the value of $x + x$, where x is the value generated by Generator, and then throw away the result.

In this diagram, there are a few interesting features to note. First, notice that Tee's `dest` port is connected twice, meaning that there are two port instances of the `dest` port. Also notice that the Tee and ALU instances have been given custom shapes for their visual representation. In general each module instance can be given a custom shape. This feature allows the visualized modules to be recognizable on sight instead of having each module be a nondescript blue rectangle. We use this feature in the next section to make the machine visualizations resemble diagrams found in computer architecture textbooks.

In addition to the above schematic rendering features, the visualizer can interface the generated simulator executable and display execution information as it occurs. This is useful for following instructions as they flow through a pipeline or observing the status of ports in the system. Screenshots of the visualizer interacting with the generated simulator are shown in the next section.

4. Applications

In this section, we will give examples of how LSE can be used in lecture to illustrate computer architecture concepts and how LSE can be used to formulate assignments that allow students to explore the myriad of interactions between architectural techniques. All the examples are centered around a simple Tomasulo-style [3] machine that executes the DLX [4] instruction set.

4.1. LSE in the Classroom

Standard presentation tools do a fine job of illustrating the static aspects of a design. However, dynamic interactions are generally only briefly described or illustrated with static pipeline diagrams. The LSE system and its visualizer, however, can demonstrate the *dynamic* behavior of the ma-

chine by displaying, over time, events produced by the executable machine model. In a lecture environment, this can be used to show the flow of data and update of state through the modeled architecture.

To illustrate this we will show a few screenshots of the Visualizer showing the flow of instructions through a simple Tomasulo-style pipeline. A screen shot of the Visualizer is shown in Figure 4. Notice that the structure of the machine is fairly obvious. The block labeled `Register File` is the register file, horizontally stacked tall vertical blocks are the reservation stations, the ALU looks like an ALU, and the shifter, LSU, and branch unit are clearly labeled. The machine does not support precise exceptions or speculation and so there is no need for a reorder buffer.

Figure 5 shows a screen shot of the visualizer displaying a table showing instruction arrival at various stages in the pipeline and reservation station occupancy of the Tomasulo-style machine shown in Figure 4. This table is dynamically updated with data from the running simulator.

In Figure 5 we can clearly see the `or` instruction that succeeds the jump instruction stall in the fetch stage starting at cycle 5 while the machine resolves the branch (recall that the sample Tomasulo-style machine does not support speculation). We can also see the `sll` instruction stuck in the reservation station awaiting operands during cycle 3. In cycle 4, we see the `sll` instruction issue to the EX stage but subseqently lose arbitration for the common data forcing a re-issue in cycle 5. The instruction once again loses arbitration and re-issues in cycle 6 and finally writes back in cycle 7.

The table is constructed by specifying the appropriate data collectors in the simulator description. The visualizer then monitors the output of these collectors to generate the table as the simulator runs. The table is completely generic and thus users of LSE may specify the column headings and how the table entries get filled via the specific messages emitted by the data collectors.

As discussed in the literature, the power of this kind of demonstration is invaluable since both the static machine structure and its dynamic behavior can be seen simultaneously [5]. With the Liberty Simulation Environment, these demonstrations can easily be constructed for many different types of machines so that students can easily understand the differences in the architectural techniques presented.

4.2. LSE for Student Exploration

As was described in Section 1, designing and implementing a machine with an RTL level description is too time consuming to do regularly throughout an architecture course. On the other hand, block diagrams do little to cement understanding of the dynamic elements of an architecture. The Liberty Simulation Environment, however, provides an excellent middle ground between hand-drawn hardware block diagrams and RTL level descriptions. Regular assignments can be given in which the student is required to modify an existing configuration to produce a new configuration. For example, students may be asked to modify a Tomasulo-style machine that does not execute loads and stores to one that does execute loads and stores in order, in 2 clock cycles. As the following example will demonstrate, this problem is certainly tractable for students in a week long assignment.

Figure 4, described earlier, shows a Tomasulo-style machine that does not execute loads and stores. Figure 6 shows that same machine with a load store unit added. Adding this load store unit is relatively straightforward.

First, the reservation station module needs to be augmented to force instructions to be issued in order. This augmented module will form the load-store issue queue (LSQ). This hierarchical module, shown in Figure 7 is built by taking the reservation station module and connecting it so that all the slots of the reservation station go to a `serializer` module, called `serialize` in the figure, followed by an `aligner` module, called `align`. The `serialize` and `align` instances, combined with the default control semantics in LSE, force instructions to come out of the `align` instance in order. Both the `serializer` and `aligner` are available in the standard LSE module library, and the reservation station is part of the original Tomasulo-style configuration.

Next, several additional module instances (created from modules in the library) are connected to the output of the LSQ and are used to extract the destination register (rd) from the data output by the reservation station, compute the load or store address, and generate the control signal that decides if the request will be a read or write. The specific fields that the module instance extracts and the function it performs are specified via algorithmic parameters (discussed in Section 2) provided by the modules.

The output of these module instances is then connected to a latch to end the first cycle of memory instruction execution. The output of this latch is then connected to the request ports of the data memory. Another module instance then combines the output of the data memory (generated for load requests) with the destination register field from the reservation station (arriving via the latch) and sends them off to the common data bus arbiter.

All of these modifications were performed in a few hours. Students moderately familiar with LSE should be able to complete such an assignment fairly easily. Furthermore, in the configuration just described, the default control semantics in LSE would allow students to vary the latency of the memory module (while keeping the initiation interval fixed at 1) and have the load-store logic stall waiting for the memory. Students could then explore the merits of their own design in the presence of different core-memory latencies with very little additional effort.

When used in this way, LSE enables instructors to give regular assignments that require students to build executable

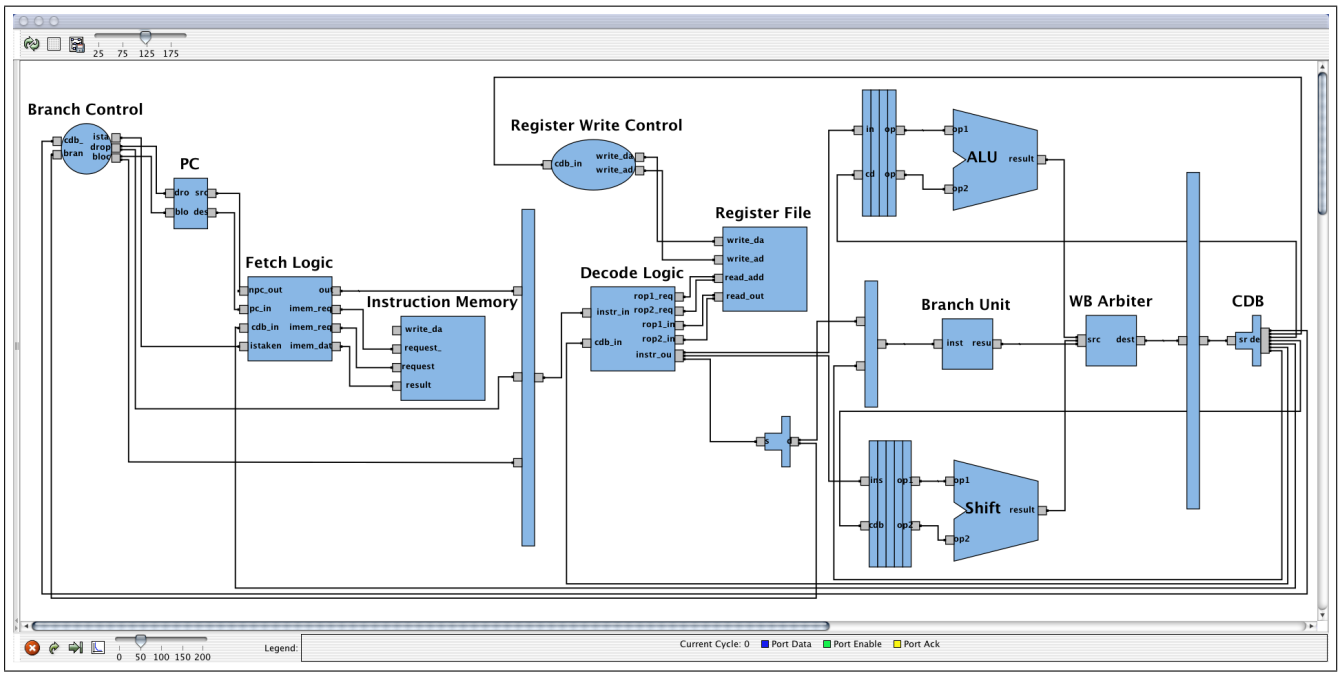

Figure 4: Simple Tomasulo-style pipeline that executes the DLX ISA.

Cycle	IF	ID	ALU_res	Shifter_res	Branch_res	EX	WB
0	addi(0)						
1	sll(1)	addi(0)					
2	add(2)	sll(1)	addi(0)			addi(0)	
3	lhi(3)	add(2)		sll(1)			addi(0)
4	j(4)	lhi(3)	add(2)	sll(1)		add(2) / sll(1)	
5	or(5)	j(4)	lhi(3)	sll(1)		sll(1) / lhi(3)	add(2)
6				sll(1)	j(4)	j(4) / sll(1)	lhi(3)
7					j(4)	j(4)	sll(1)
8							j(4)

Figure 5: Table showing instruction arrival and reservation station occupancy in a Tomasulo-style DLX machine

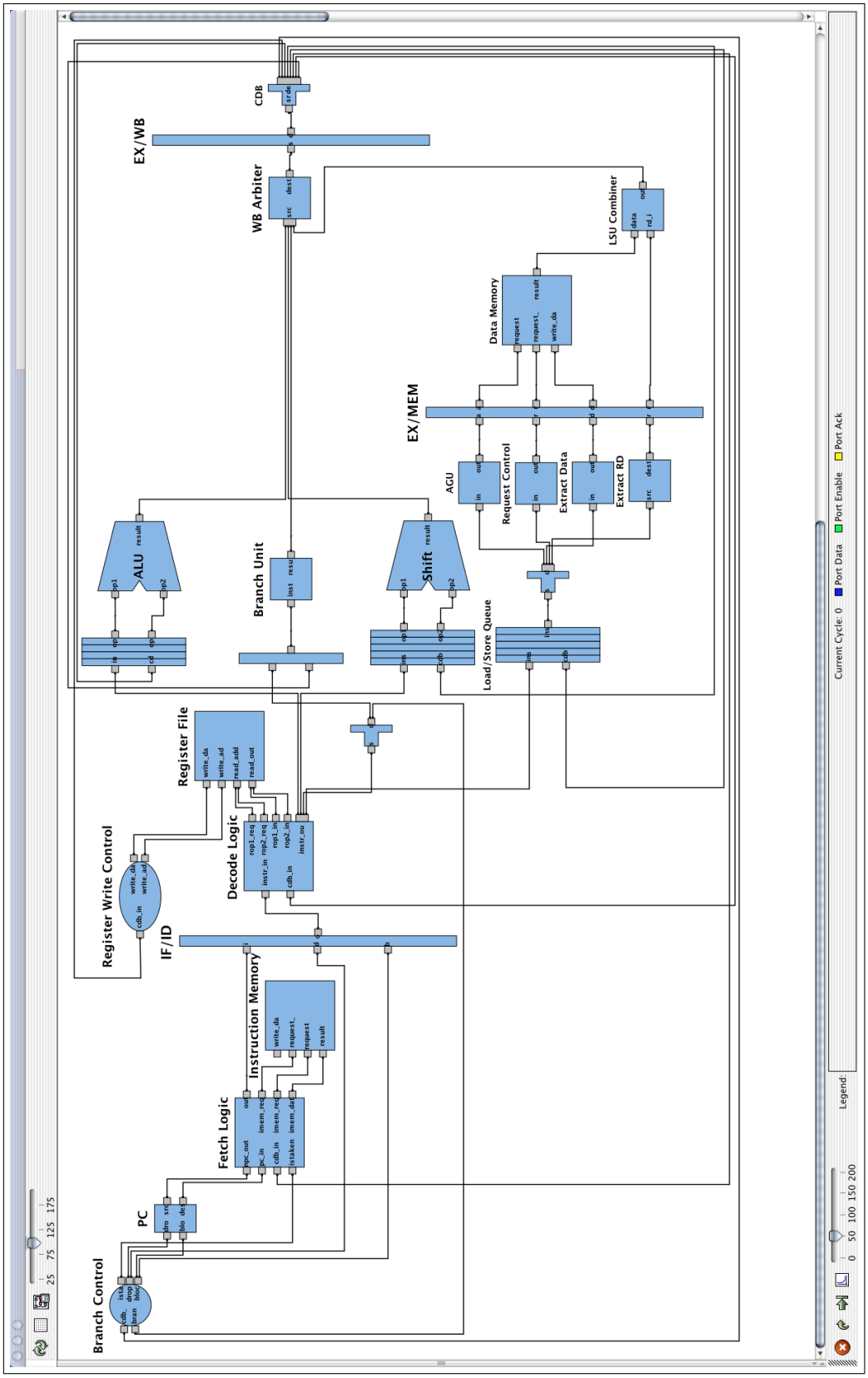

Figure 6: A Tomasulo-style DLX machine with a load-store capability.

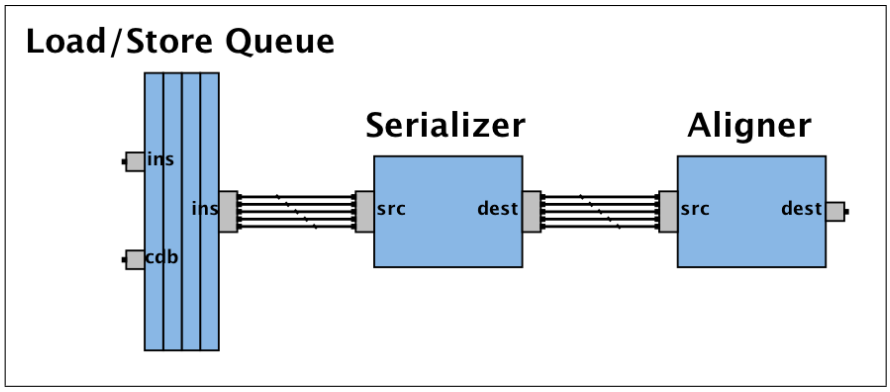

Figure 7: Load-Store issue queue.

models to verify that their understanding of an architectural concept is sufficient (i.e. the model runs programs correctly). The students can learn about most of the techniques presented in class with hands-on projects, instead of only a handful they would see by doing a single class project.

As a further example, students can explore machines not described in lecture by having them add architectural mechanisms to existing designs. For example, students could be asked to add rename logic to a scoreboarded machine before discussing advanced scoreboarded machines. In this way students can appreciate the relationship between renaming and WAR hazards and why scoreboards stall in circumstances where Tomasulo's machine does not. LSE makes this kind of exploration feasible in week long assignments.

5. Conclusion

To understand computer architecture, students must understand the *dynamic* interactions of all the hardware components in a microarchitecture. Unfortunately, conveying the many subtleties of this interaction during lecture is difficult. For many students, *static* illustrations and assignments do not build intuition about dynamic systems. Class projects which require students to build RTL simulation models are extremely useful, but the overhead in low-level model construction and modification often limits the scope of concepts explored. Modifying or writing a high-level simulator in a sequential language such as C allows students to avoid getting bogged down in irrelevant low-level hardware details, but it does so by obscuring the model. Students spend much of their time dealing with sequential language simulator issues rather than thinking about computer architecture.

In this paper, we have shown that the Liberty Simulation Environment (LSE) is an alternative to tools currently used in lecture and take-home assignments. LSE's simulator description resembles hardware, allowing students to think about hardware rather than simulator design issues. LSE descriptions are relatively easy to modify and use, allowing students to study the dynamic execution behavior of a wide range of machines. Furthermore, the LSE Visualizer improves LSE's use as a pedagogical tool by tying this all together with an easy to use dynamic and graphical visualization system.

References

[1] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 271–282, November 2002.

[2] D. Penry and D. I. August, "Optimizations for a simulator construction system supporting reusable components," in *Proceedings of the 40th Design Automation Conference*, June 2003.

[3] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.

[5] C. T. Weaver, E. Larson, and T. Austin, "Effective support of simulation in computer architecture instruction," in *Proceedings of the 2002 Workshop on Computer Architecture Education (WCAE)*, May 2002.