DLSys: A Toolkit for Design and Simulation of Computer System Architecture
John L. Donaldson, Richard M. Salter, and Rebecca E. Punch
Oberlin College
Oberlin, OH
john.donaldson@oberlin.edu, rms@cs.oberlin.edu, rpunch@cs.oberlin.edu

ABSTRACT
DLSim 3 is a GUI-based digital logic simulator designed for use in Computer Organization and Architecture courses. Its key features are support for abstraction (allowing subcircuits to be abstracted as black boxes) and extensibility (allowing new circuit elements to be created as Java plug-ins). These features make DLSim 3 suitable for use at all levels of Computer Organization study. The purpose of this paper is to demonstrate the use of DLSim 3 at the system architecture level. We introduce DLSys, a collection of DLSim 3 circuits and plug-ins intended for use in the design and simulation of single- and multiple-CPU computer systems.

Categories and Subject Descriptors

General Terms
Design

Keywords
Logic Design, Simulation.

1. INTRODUCTION
Hardware simulators are popular software tools commonly used in Computer Organization courses. Many such tools exist, such as Logisim [1], MARS [12], and JLS [6], covering various aspects of computer organization, including logic design and assembly language programming. Nikolic et al [3] provide a survey and evaluation of more than 25 such simulators. They are effective because they allow students to gain valuable hands-on experience in design and programming, without the problems involved in maintaining a hardware lab. No soldering iron is required.

One problem with such simulation programs is that they tend to focus on only one aspect of the course. The DLSim 3 simulator [8] was created in response to this problem. DLSim 3 provides an extensible, visual simulation platform which allows it to be used in studying Computer Organization at any level, from low-level logic design using gates and flip-flops, through the design of mid-level components which can be used to design a complete CPU, and then up to the system architecture level where components such as CPUs, memories, buses, and I/O devices are used to design complete computers.

Extensibility is achieved through the use of Java plug-ins. A plug-in is a simulated circuit (e.g., register, ALU, etc.), written as a Java class, which can be added to DLSim 3’s palette of circuit components. Users can create their own customized plug-in components which can then be incorporated into circuit designs. Instructors can use this capability to set up particular demonstrations or exercises. By using appropriate plug-ins, an instructor can use the same simulator for all aspects of the Computer Organization course.

Our previous work [2, 7, 8, 9] has described the use of DLSim 3 at the low and medium levels of logic design. In particular, [2] and [9] describe the use of DLSim 3 as a tool for simulating several educational CPUs, such as the Mic-1 [10], the MIPS subset of Patterson and Hennessy [5], and the LC-3 [4]. The next logical step in utilizing DLSim 3 was to apply it to complete computer systems. To that end, we created DLSys, a toolkit of DLSim components and circuits intended for use in Computer Organization courses at the system architecture level.

DLSys contains plug-in components representing simulated CPUs, memory modules, I/O devices, and other supporting chips. It also contains a large number of complete circuits using these components. These circuits are computers utilizing a variety of bus designs and interconnection techniques.

We have two primary objectives:

- To create a visual tool for presenting typical system designs in class demonstrations. Such demonstrations should be visually appealing and illustrate important design concepts. Since we are operating at the system level, the circuits we are designing are complete computer systems. Therefore, the simulated systems should run real programs that interact with the user through simulated I/O devices.

- To create a platform for student homework in logic design: creating new circuits and plug-ins, or modifying existing ones. The platform should provide a faithful representation of the operation of logic circuits. It should be both suitable for debugging and fun to work with.

In this paper we describe DLSys in detail and how it can be used to supplement instruction in Computer Organization courses.
2. CIRCUITS

At the system level, the focus of logic design is on the design of computer systems, connecting CPUs and other devices through external buses. The objective of the DLSys toolkit is to provide a set of interchangeable components which can be combined to simulate a variety of computer system architectures. To that end, we have created a collection of computer designs, which are described in this section.

The purpose is to investigate such design issues as

- asynchronous vs. synchronous buses
- single bus vs. multiple bus designs
- address decoding
- memory-mapped vs. isolated I/O
- interrupt-driven I/O vs. polling
- bus arbitration
  - centralized vs. decentralized
  - prioritization
  - cycle stealing vs. block transfer
- cache memory design
  - write-through vs. write back
  - associativity
  - replacement policy
- multiprocessor design
  - switching

The DLSys toolkit does not contain examples of every possible design decision. Many are left as exercises for the student. Nevertheless, a large number of examples are provided to illustrate the results of a variety of design choices. Furthermore, the toolkit has the flexibility to be extended through the development of additional plug-ins and circuits. Some of the circuits we have implemented are:

1. Synchronous bus computer with CPU and memory.
2. Asynchronous bus computer with CPU and memory. In this case, we use the same clock-driven CPU as in the synchronous bus computer. However, an interface chip (i.e., plug-in) is inserted between the CPU and the bus so that it can communicate on the asynchronous bus.
3. Single-bus computer with CPU and 4 memory chips. The high-order bits of the address lines are used to select the chip, illustrating one aspect of memory interfacing.
4. Single-bus computer with CPU, memory, and I/O console. The CPU uses polling to interact with the console.
5. Single-bus computer with interrupts. In this design, two I/O consoles are connected to the bus, each with the capability of interrupting the CPU when input is entered. The bus has 8 interrupt channels, and an interrupt controller chip is used to prioritize the interrupts presented to the CPU (which has a single interrupt request pin). The algorithm executed by the CPU includes an interrupt handler.
6. Single-bus computer with a DMA-capable disk. A centralized bus arbitration protocol is implemented by a plug-in connected to the bus. Both the CPU and disk have lines to send bus requests to the arbiter. The disk can operate in either cycle stealing mode or burst mode. The disk signals completion of a data transfer using an interrupt.
7. Single-bus computer with cache memory.
8. Single-bus multiprocessor with 4 CPUs and 4 memory modules. The same bus arbiter chip that was used to implement DMA is used to control access to the bus by the 4 CPUs.

9. 4-CPU multiprocessor using a 4x4 crossbar switch for connection to 4 memory modules.

Figure 1 illustrates a simple single bus computer implemented in DLSim 3. It utilizes three plug-ins: the CPU, a random-access memory, and an input-output console. The horizontal lines connecting the components comprise the bus, which contains lines for data and address, grouped in 16-bit bundles. There are also single lines for a memory request signal, a read/write flag, a ready response from the memory, and the clock. The CPU is executing a sort program and a console echo program. As it sorts the integer data values visible in the memory display, it periodically polls the console for input. If an input character is available, it reads it and echoes it to the output window, and then resumes sorting. An interrupt-driven version of this circuit also exists.

Figure 2 shows a more complex design, a 16-bit single bus multiprocessor. Above the CPUs is the bus arbiter plug-in, connected to the CPUs by eight bus request and eight bus grant lines. Each of the four memory modules occupies one quarter of the 16-bit address space. The and, or, and inverter gates beside the memories are used to decode the high-order two bits of the address in order to select one of the memories. Each of the CPUs runs the same program, parameterized by the CPU id (0, 1, 2, or 3). The program copies data from an array in one of the memory modules to an array in the next.

In this version of the multiprocessor, only one CPU can access memory at a time. DLSys also contains a multiprocessor with a crossbar switch which allows different CPUs to access different memory modules simultaneously. There is still contention if two CPUs attempt to access the same memory at the same time. In that case, the switch needs to arbitrate between the requests and perform them in sequence.

3. PLUG-INS

It is through DLSim’s plug-in capability that it becomes possible to scale up from simple circuits involving gates and flip-flops to the level of system architecture. For the DLSys toolkit, we have produced a large collection of simulated plug-in components, including CPUs, memories, and I/O devices, suitable for use in designing systems.

3.1 CPUs

As a component in a larger system, the essential feature of the CPU is that it generates a sequence of memory and I/O requests. In designing a plug-in to represent a CPU that would carry out such requests, we had two goals. First, it needed to incorporate the logic of various protocols to interface with external devices on a bus. In addition, for the purpose of producing interesting demonstration circuits, we wanted the CPU to be able to execute programs that would perform interactive I/O. As a result, we did not consider the use of static traces of memory accesses, although that could easily be implemented in DLSim. We also determined that at this level, we were not concerned with the details of the instruction set architecture of the CPU. This is in keeping with the design philosophy of DLSim 3: the instructor can focus on certain aspects of the design, while hiding others in plug-ins.

With these goals in mind, we created several CPU plug-ins of increasing complexity, with these features:

- Each CPU plug-in has address, data, and control pins to interface with a bus or other components. Some of the
CPUs have additional pins for interrupt control and bus arbitration.
- It executes algorithms written in Java. Memory references are generated naturally as a result of execution of the algorithm.
- It is clock-driven. At each clock tick, it executes as much of the algorithm as it can without requiring a memory or I/O operation. At that time, it carries out the first step of an I/O protocol. When the I/O protocol (also clock-driven) is completed, the algorithm resumes.

The effect is that the CPU acts as a finite state machine, augmented by a large, external, random-access memory. The Java algorithms that it executes must be adapted to fit this model.

We note that the sequence of memory operations generated in this way does not completely reflect the pattern that would be seen on a real system. In particular, because the CPU is not actually executing a machine language program, it does not need to fetch machine instructions from memory. Nevertheless, the data read and write operations that are performed are entirely adequate for verifying the interfacing protocols.

We found that several of the most effective algorithms to be used with the simulator are sorting algorithms (e.g., insertion sort, quicksort, etc.) They are algorithms that students are familiar with, having seen them in other courses. They provide a good visualization of the step-by-step operation of the algorithm.

As we experimented with a variety of different algorithms, we spun off an Algorithm class from the main CPU class. The structure and operation of the Algorithm class is described in section 5.

3.2 Memories
The memory component of a computer consists of the memory chips themselves, together with an interface which connects the chips to a bus. For DLsys, we produced an integrated memory plug-in which combines the storage capability of the memory chips with the control signals needed to interface to a synchronous bus. We used this plug-in in most of our circuit designs. We also have plug-ins for simple memory chips, as well as plug-ins for both synchronous and asynchronous bus interface circuits. Because we were most interested in the system architecture at the bus level, we made no distinction between static and dynamic RAM chips. The more complex interfacing required for dynamic RAMs (invoking, for example, refresh cycles) could be implemented by instructors wishing to focus on that level of detail.

One of the customizable properties of the memory plug-ins is a time delay, representing the response time of the memory. The delay is implemented in two different ways in two versions of the memory. In the integrated memory, which includes an interface to a synchronous bus and therefore has a clock input, the memory receiving a read or write request records the system time and then returns control to the simulator. On each clock tick, it is awakened, at which time it compares the current time to the saved time. If the delay time has been exceeded, the memory carries out the requested operation and asserts its ready line.

For an asynchronous memory with no clock input, a different technique must be used. In this case, the plug-in returns control to the simulator after setting a Java Timer with the appropriate delay. When the Timer goes off, the memory is awakened and completes the requested operation.

In addition to conventional main memory, we implemented a cache memory plug-in, adapted in part from the cache plug-in described in [2]. The cache plug-in is designed with two interfaces: one to the CPU and one to the system bus or main memory. It is customizable (using plug-in properties) with respect to overall size, number of bytes per line, level of associativity (direct, fully associative, or set associative), and replacement policy.

3.3 I/O Devices
DLsys contains several plug-ins which simulate I/O devices: a button, an I/O console, and a magnetic disk. As we did with memory, we have chosen to integrate the device itself with its interface to a synchronous bus. The button is a simple device that, when pressed, triggers an interrupt request to the CPU. The I/O console, adapted from the I/O console described in [2], is actually comprised of two devices: an input area which takes characters input from the keyboard and an output area to display characters written by the CPU. Both areas are implemented as JTextAreas. Each device uses a set of registers (data, control, and status) in the I/O or memory address space for communication with the CPU.

The disk plug-in consists of a storage area, a set of interface registers, and a bus interface. The backing store for the disk data is actually a file in the file system; during a simulation using the disk, simulated disk operations are performed on the file, making permanent changes to the file contents. The disk is capable of performing direct memory access (DMA); that is, becoming the bus master in bus transactions. As such, it needs the ability to perform reads and writes, respond to memory read and write requests from the CPU, initiate bus requests, and generate interrupt requests.

3.4 Miscellaneous plug-ins
1. Interrupt controller. We have written an interrupt controller plug-in whose functionality is similar to that of the Intel 8259A chip found in PCs. It takes up to 8 interrupt request lines from devices, prioritizes them, and delivers a single interrupt request signal to the CPU. The prioritization method (fixed vs. round-robin) is a customizable property of the plug-in.
2. Bus arbiter. On many systems, the CPU itself contains bus arbitration logic. It is also possible to implement a decentralized bus arbitration scheme. We have chosen to implement a centralized arbiter as a separate plug-in. The arbiter takes as input up to 8 bus requests and may assert a bus grant on one of 8 bus grant lines. The arbitration method (using fixed priorities or round-robin) is customizable.
3. Crossbar switch. A crossbar switch plug-in was created for the design of a 4-CPU multiprocessor.

4. PROTOCOL DESIGN
One of the things that set system architecture apart from other aspects of Computer Organization is its emphasis on communication. Because of this, some of the design techniques involved overlap with methods used in the design of computer
networks to develop communication protocols. In DLSys, the logic for implementing a communications protocol is found in the plug-in code. A plug-in is made of several components:

- A static description of the plug-in’s input and output pins.
- An optional view class, which defines the appearance of the plug-in in the visual simulation window. The view may include GUI controls to be used for input and output with a human user.
- An evalBundle method which defines the dynamic behavior of the plug-in. It is called by the simulator whenever a change is detected on one of the plug-in’s input pins. The plug-in executes the evalBundle code, which may result in a change in the internal state of the plug-in, the values on its output pins, or both.

Thus, the evalBundle method contains the basic logic of the plug-in, written in Java code. As a result, the focus of plug-in writing at the system architecture level is on the development of protocols to perform tasks such as data transfer, bus arbitration, and interrupt control.

An effective method for designing a protocol between components is to use a finite state machine to model the system. The situation is complicated somewhat by the fact that the state of the system is actually a combination of the states of its interacting components. This technique is illustrated in Tanenbaum’s text on Computer Networks [11].

Once a FSA model has been designed, it is then a fairly straightforward task to translate the logic of the FSA into Java code. Nevertheless, communication protocols are not easy to design and usually require some debugging effort. We have found that the DLSim 3 simulation platform is an excellent tool for testing and debugging a protocol.

Figure 3 illustrates a FSA used to design a synchronous protocol for data transfer between CPU and memory. Each circle indicates a system state combining the CPU state (idle, readwait, or writewait) and the memory state (idle, readcycle, writecycle, or completed). For a read operation, the following steps occur:

1. CPU places an address on the address bus, asserts READ, and enters the readwait state.
2. Memory receives the READ signal, records the system time, and enters the readcycle state.
3. Memory continues to read the system time. When its programmed delay expires, it performs the read (placing the data on the data bus), asserts RDY, and enters the completed state.
4. CPU receives the RDY signal, clocks the contents of the data bus into the MDR, and enters its IDLE state. Memory enters its IDLE state.

![Figure 3. FSA for a synchronous data transfer.](image-url)
5. ALGORITHMS

The simulated CPUs found in DLSys execute programs. As a result, they are able to interact with live input data entered by a human user and generate a series of memory and I/O operations that is the same as if the program were running on a real CPU. The details of the CPU’s instruction set architecture, however, are not simulated. Instead, the algorithms are written in Java. However, the Java code that one might find in a normal program must be modified to fit the needs of the simulator. In particular, read and write operations must now be treated as I/O operations, which may incur some delay caused by such factors as memory and I/O response time and bus access time. Furthermore, a waiting CPU must not enter a wait loop; this would cause the simulator to be held up indefinitely.

Instead, programs are written in a style that is essentially a finite state machine. When an algorithm initiates a read or write operation, it saves its current state and then exits, thus returning control to the simulator. When the memory operation is completed, the CPU is signaled on its READY input pin, the saved state is restored, and execution of the algorithm resumes where it left off. Figure 4 shows the FSA that was created to represent the insertion sort algorithm. Note that after every memory operation (initread, initwrite, or finishread), control passes to a new state. This represents the limit of what can be done in one invocation of the CPU’s evalBundle method prior to the next timer tick.

We have written algorithms in this style for various sorting algorithms (e.g., bubble sort, insertion sort, merge sort, quicksort, radix sort) as these algorithms provide a good test of memory read and write operations. For I/O testing, we have programs such as a chat program and a program to perform a sort on disk data. Algorithm is a separate Java class. This makes is possible to plug an algorithm into any of our CPU plug-ins.

6. SUGGESTED EXERCISES

In order to give the flavor of how DLSys can be used in the classroom, we present a set of suggested student exercises. Many other exercises are possible; in particular, any of the plug-ins can be given to the class with the evalBundle logic missing. The exercise is then to fill in the missing code so that the plug-in performs correctly in a given circuit.

1. Write the evalBundle method for a given CPU plug-in so that it employs the proper protocol for connection to memory through an asynchronous bus.
2. Modify the single-bus computer to use a decentralized arbitration protocol, eliminating the need for a bus arbiter chip.
3. Connect several I/O devices to a single interrupt request line using daisy chaining.
4. Modify the CPU plug-in so that the CPU takes on the role of the bus arbiter. Then modify the single-bus computer to reflect this change.
5. Modify the CPU plug-in so as to separate the I/O address space from the memory address space. This will require an extra pin to distinguish memory read/writes from I/O read/writes.
6. Separate the I/O bus from the memory bus in a given computer design, thus creating a two-bus computer.
7. Design a NUMA (non-uniform memory access) multiprocessor.

7. CONCLUSIONS

With the addition of the DLSys toolkit, DLSim 3 provides a rich simulation environment for exploring Computer Organization concepts at the system architecture level. Instructors can use it to enhance their courses with dynamic, visual in-class demonstrations and a variety of hands-on student exercises.
The simulator provides a good test of logic. In developing our own designs, we found a number of instances in which something that appeared correct on paper did not run correctly. In each case, it was our logic that was flawed, not the simulator’s. In fact, the simulator proved to be an excellent tool for debugging our designs.

It is accessible to undergraduate students. Indeed, one of the authors of this paper is an undergraduate student, who wrote many of the plug-ins and circuit designs described here.

We have not yet had the opportunity to use DLSys in the classroom, but we expect it to have an impact similar to that of DLSim 3, which we have been using to illustrate lower-level logic design concepts in our Computer Organization course for several years. We have found that presenting live demos of important concepts gives a strong “wow” factor in the classroom, motivating students to experiment with their own designs.

The complete DLSim 3 simulation platform, including DLSys and many other example circuits and plug-ins, is available for free download at our website www.dlsim.com.

8. ACKNOWLEDGEMENT
The authors thank Nitun Poddar for his contribution to the quicksort algorithm and asynchronous bus plug-ins.

9. REFERENCES