Illustrating CPU Design Concepts with DLSim 3

John L. Donaldson, Richard M. Salter, Joseph Kramer-Miller, Serguei Egorov, and Akshat Singhal
Oberlin College, john.donaldson@oberlin.edu, rms@cs.oberlin.edu, joseph.kramer-miller@oberlin.edu, serguei.egorov@oberlin.edu, akshat.singhal@oberlin.edu

Abstract - **DLSim, a GUI-based digital logic simulation program developed by Richard Salter at Oberlin College, has been used for class demonstrations and homework exercises in the Computer Organization course at Oberlin for over ten years. Until recently its use has been limited to the component of the course dealing with low-level logic design using gates and flip-flops. A new version, DLSim 3, extends those capabilities through the use of Java plug-ins, making it possible to use the software for digital design at higher levels of abstraction. With DLSim 3, we are able to present the many levels of circuit design in a single environment, from low level combinational and sequential circuits through models of complete CPUs. The purpose of this paper is to give an introduction to DLSim 3 and to describe how we have used it in the classroom, focusing particularly on CPU design.**

Index Terms – Circuit Simulation, Computer Organization, Digital Logic.

INTRODUCTION

Computer Organization is a standard course in Computer Science and Computer Engineering programs. In Computer Science programs, which are geared toward software theory and design, it is often the only hardware-oriented course in the curriculum. For this reason, it must cover a wide range of material, from low-level digital logic through CPU design at the register-transfer level, design and programming of instruction set architectures, and system-level design involving buses and I/O interfaces. Although good software tools exist to assist in the study of each of these areas, there has not been a single tool which covers all of them.

The result is that in a typical Computer Organization course, the student must learn the use of a new tool for each area being studied. Just when the student becomes comfortable using, for example, a digital logic design tool, it is time to move on to learning to use an assembler or cache simulation program.

Many excellent GUI-based logic simulation systems have been developed and are available for download, such as Logicsim [1], JLS [2], Digital Works 3.0 [3], and Logisim [4,5]. These systems are very useful for studying basic combinatorial and sequential circuits. While they generally all provide some abstraction mechanism ("black boxes") that permits circuit reuse, they are limited by their GUI based environments to relatively small models.

DLSim 3 goes beyond these efforts through its innovative use of Java plug-ins. A plug-in is a software module, written in Java, which is added to DLSim's design platform and can be used as a component in more complex circuit designs. The user can write Java modules that simulate higher-level logic components (i.e., memories, registers, ALUs) which supplement DLSim's collection of built-in components. The plug-in facility is built around an interface that describes what functions a plug-in must perform in order to be installed in the system, and an API of functions which support the writing of plug-ins.

Plug-ins allow the simulator to scale up to whatever level of abstraction is appropriate in a course. For example, at one stage in a course, students might be asked to design an ALU using only logic gates. Later, when studying CPU design, the instructor might provide an ALU plug-in to be used as a component.

In addition, plug-ins can perform I/O, either GUI-oriented user interaction or file operations. For example, we have written plug-ins which can load a microprogram from a file, store results of a simulation to a file, and provide an interactive keypad for a simulated calculator.

The design philosophy behind DLSim was described by Salter and Donaldson in [6]. In this paper, we present an introduction to plug-ins and then describe how we have used DLSim to visually simulate the CPU designs presented in two of the leading Computer Organization textbooks, by Patterson and Hennessey [7] and Tanenbaum [8]. With these designs, we have been able to illustrate important CPU design concepts such as datapath construction (both conventional and pipelined) and control unit design (both hardwired and microprogrammed). In addition, we describe several student projects which have been implemented using DLSim 3.

PLUG-INS

The power of DLSim 3 to model large-scale circuit components, such as RAM chips and CPUs, is achieved through plug-ins. A plug-in is a Java class which represents a circuit component. Every plug-in is a subclass of DLPlugIn, which gives it the basic structure it needs to fit into the DLSim 3 simulation engine.

Every plug-in must specify at least some basic functionality by overriding methods in the base class:

- the constructor must indicate the number of input and output pins of the plug-in
- the evalState method defines the behavior of the plug-in. During a simulation, it is invoked whenever there is a change in state in any of the plug-in's inputs.

These methods are part of DLSim 3's plug-in API. The API contains a variety of methods designed to assist plug-in writers, with capabilities such as

- setting input and output size
- getting input values
- setting output values
- pin grouping
- conversion of pin values to and from decimal or hexadecimal representation

By default, DLSim 3 displays a plug-in as a rectangle with inputs on the left and outputs on the right. The programmer may, however, provide a customized view for the plug-in by writing a separate view class. The plug-in view may

- change the shape of the component
- change the positioning of the input and output pins
- use the component as a Java Swing Container holding Swing Components such as labels, text fields, etc.

The CPU designs in Figures 1 and 2 illustrate several examples of plug-ins with customized views, such as the register file, input/output area, and ALU.

IMPLEMENTATION OF MIPS

We have implemented a simplified MIPS-architecture processor in DLSim 3. The design follows the description of the MIPS processor by Patterson and Hennessy [7]. They present a detailed design for a processor which implements a subset of the MIPS instruction set. This subset consists of a small but functionally complete handful of instructions (add and subtract, memory load and store, conditional branch). Figure 1 shows our implementation, which we have dubbed the *MicroMIPS*, as it appears in the display area of DLSim 3.

I. Basic Implementation

The Patterson and Hennessy implementation divides the execution of a single machine instruction into five steps: instruction fetch, instruction decode/register fetch, ALU execution, memory access, and register writeback. To implement the processor in DLSim 3, we used the following plug-ins:

1. Register. The register plug-in represents a 32 bit register. It is used in several places: the program counter, latches for fetched registers, an ALU result register, and the LMD register which is used to hold the value loaded from memory in memory load instructions. The register view shows the value currently stored in the register, displayed in hexadecimal.

2. Register file. The MIPS processor has a set of 32 general purpose 32-bit registers. The register file plug-in implements this register set. The view presents a scrollable window which contains all 32 registers displayed in hexadecimal, as shown in Figure 1. Note that the plug-in is a block with input and output pins. A pin may represent a pin bundle interfacing to a multiline signal. For example, the "Wr add" pin carries 5-bit register number; the "Wr data" pin carries a 32-bit data value. The face of the plug-in is a Java Swing visual component which holds a set of GUI controls for interactive input and output.

3. Memory. The memory design represents a random-access memory customized with the following parameters:

- Size in bytes
- Base address

This makes it possible to use the same plug-in for both the instruction memory and the data memory, which are treated as separate components in the Patterson and Hennessy design. Each memory component can be mapped into a different region of the processor's 32-bit address space.

4. ALU. The ALU plug-in accepts two 32-bit data inputs and a 4-bit function select input, and produces a 32-bit data value and a zero flag.

5. Multiplexer. Multiplexers are important in the CPU design to allow the processor to choose between inputs at various stages of execution. For example, the program counter's input value must be selected from the current program counter incremented by 4 and a branch address. The second ALU input is selected from a register value and an immediate operand. A simple plug-in selects one of two or more 32-bit inputs and produces a 32-bit output. The selection bits are generated by the CPU's control unit.

A multiplexer, of course, has a simple logical structure which could easily be implemented at the gate level. However, connecting all of the wires for a 32-bit mux is rather cumbersome. A plug-in is used here for convenience

6. Control unit. The hard-wired control unit generates control signals based on the current instruction. These control signals include multiplexer selection bits, register- and memory-write flags, etc.

7. Input/output console. Although it is not part of the MIPS design presented by Patterson and Hennessy, we have added an input/output console to our implementation. This makes it possible to write programs for the MIPS which perform interactive input and output. The I/O console uses memory-mapped I/O to perform its functions. It is written as a subclass of the memory plug-in, using the same pin interface, but with the following changes:

- Reads and writes to the memory I/O locations are intercepted and used to trigger I/O operations.
- A customized view provides input and output text areas for user interaction.

II. Processor Versions

Patterson and Hennessy describe several different implementations of the MIPS processor. We have implemented both their single-cycle version and their five-cycle version of the processor. In the five-cycle version, each instruction cycle is divided into five discrete time periods, each one directed by a separate clock pulse. We use a clock generator plug-in to take a single clock signal as input and generate a sequence of five distinct clock signals.

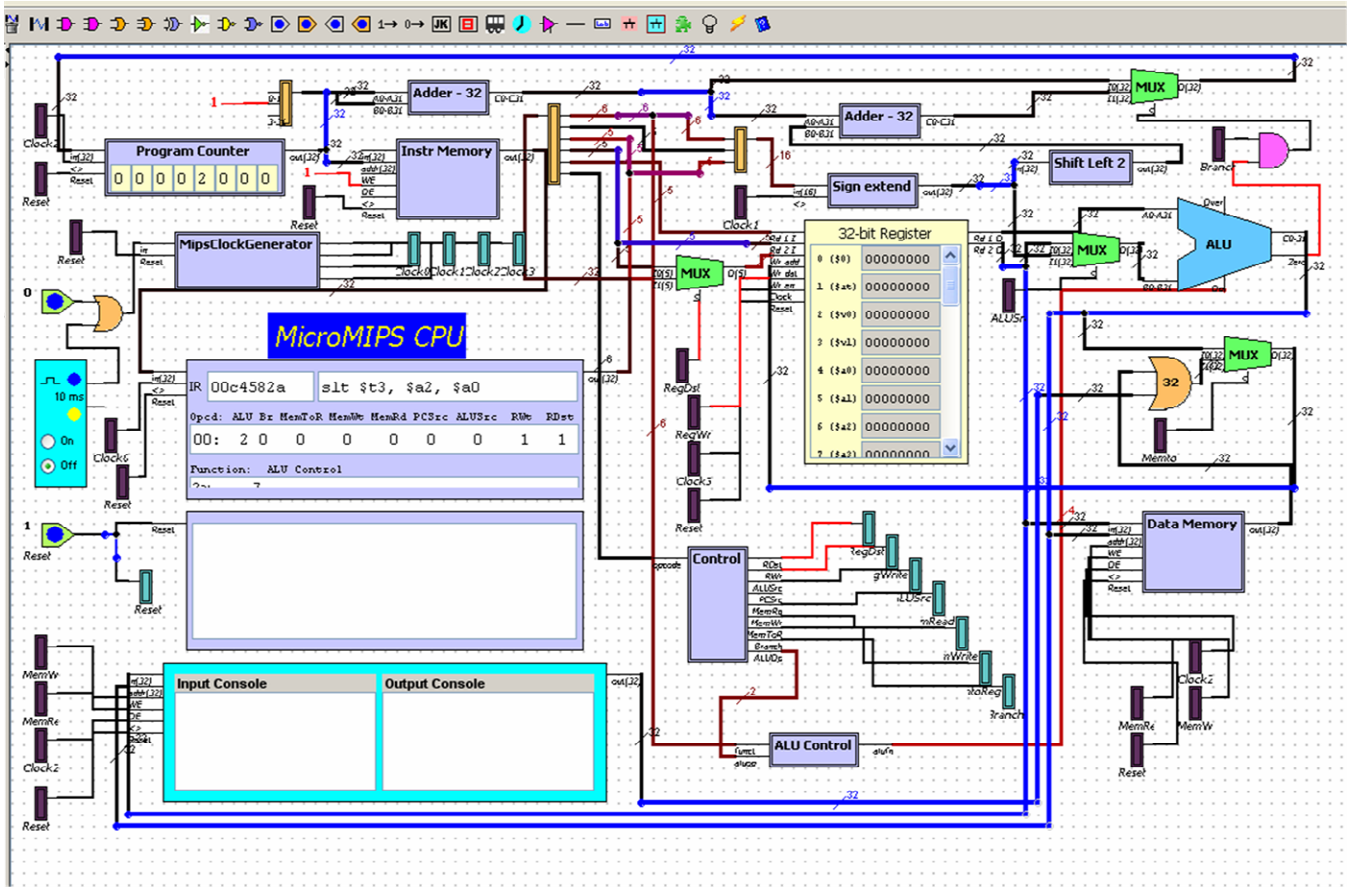

FIGURE 1
THE MICROMIPS PROCESSOR

Each clock signal causes a result value to be stored in an intermediate register, as follows:
Clock 0: fetch the next instruction and store it in the instruction register.
Clock 1: fetch register values from the register file.
Clock 2: store the current ALU result in the ALU result register.
Clock 3: store a value loaded from memory in the LMD register.
Clock 4: write back a value from the ALU result register or the LMD register to the register file; store the next instruction address in the program counter.

This version of the processor serves as the precursor to the pipelined version described below.

For the single-cycle version of the processor, all of the registers used for intermediate values between stages are removed, leaving only the program counter as a control register. Even the instruction register is eliminated. When a clock tick occurs, the program counter is loaded. This causes all of the processor actions to occur in combinational logic. The next clock tick causes the current instruction's results to be saved (i.e., register writeback and memory write) at the same time that the program counter is updated.

For both of these versions of the processor, the same set of plug-ins is used. They are simply connected differently. The instructor may choose to use both versions to demonstrate the differences between them and to visually show how each one works. Another approach would be to use the processors as a homework exercise; for example, provide one version of the processor and ask students to convert it to the other version.

III. MIPS with Pipelining

We have also implemented a pipelined version of the MIPS processor in DLSim 3. This version of the processor is based on the design presented by Patterson and Hennessy [1], constructed from the five-cycle version by inserting pipeline registers between stages and replacing the five-cycle clock with a single clock which causes each stage to operate on each clock pulse.

In class, the pipelined processor provides a dramatic illustration of how a pipeline works. Each register has a view which displays the register contents, so when the clock is single-stepped manually, students can observe each instruction passing through all stages of the pipeline, accompanied by its associated data values. When the clock is run at full speed, students can see a program run to completion with a significant speedup in comparison with the nonpipelined version of the processor.

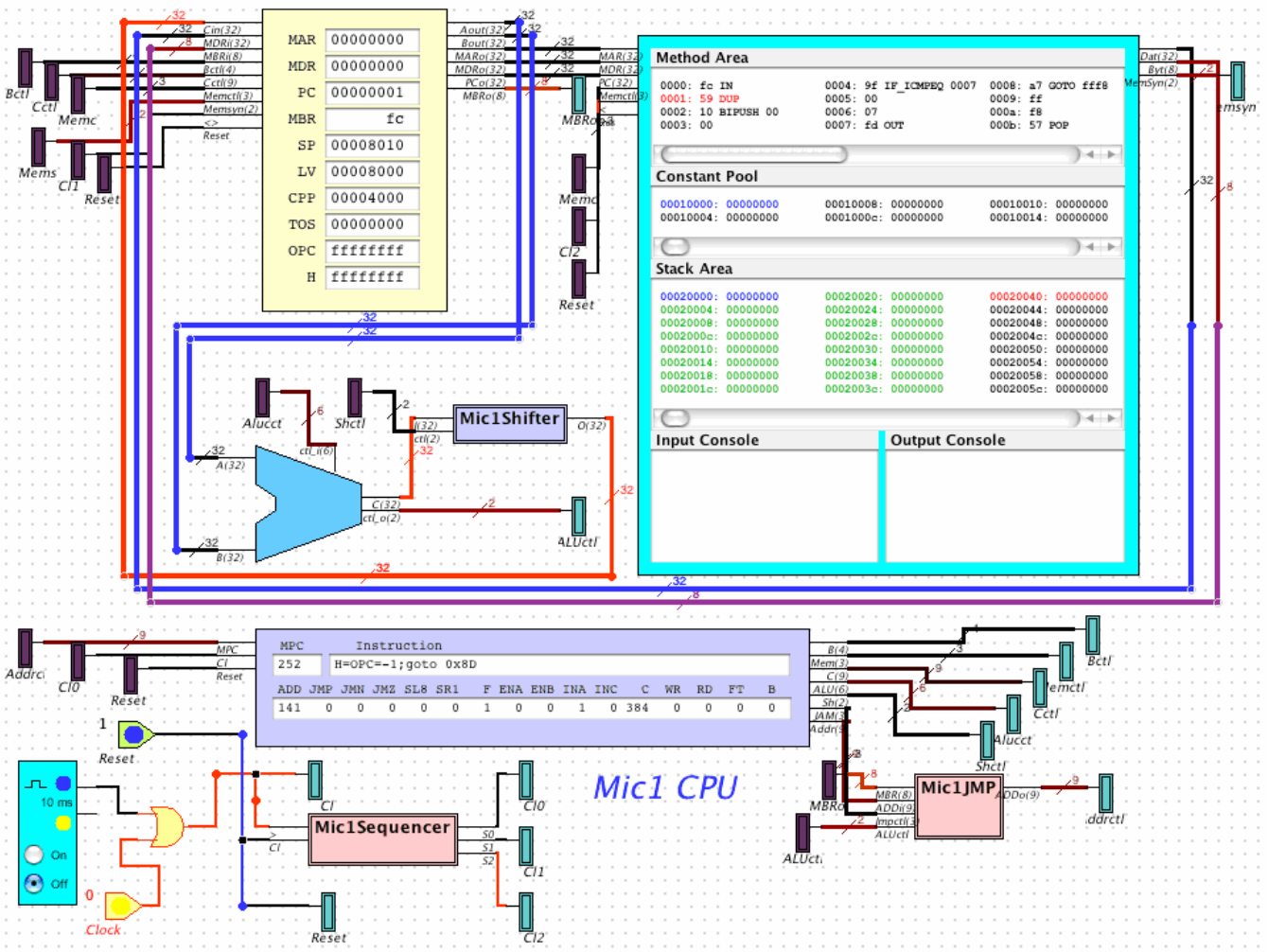

FIGURE 2
THE MIC-1 PROCESSOR

Several types of homework exercise are suitable for use with the pipelined processor, such as:

1. Converting from the five-cycle non-pipelined processor to the pipelined one. This is a substantial exercise, so as a first step, it is advisable to perform the datapath and control modifications without any hazard detection or removal.
2. Implementing forwarding. Students can be given a pipelined version of the processor that does not implement forwarding. This processor can run some programs, but will not run all programs correctly without explicitly inserting nop instructions in the program where data hazards would otherwise occur. The assignment is then to implement one or more instances of forwarding. Each type of forwarding will allow a greater number of programs to be executed without error. An alternative approach would be to provide students with a version of the processor in which some of the forwarding cases are handled, and let them add the rest.
3. Implementing stall cycles. The Patterson and Hennessy design can handle all cases of data hazards through forwarding, except when a memory load instruction is followed immediately by an instruction which uses the loaded value. In this case, a stall cycle (actually a nop instruction) must be inserted into the pipeline on the fly. The student exercise is to modify the design to detect and handle this hazard.

IMPLEMENTATION OF MIC-1

Tanenbaum [8] describes *MIC-1*, a microprogrammed CPU architecture which implements a subset of the Java virtual machine called *IJVM*. We have created a working model of this architecture in DLSim 3, shown in Figure 2.

The Mic-1 architecture maintains a microprogrammed control unit to manage a 32-bit wide datapath. This datapath runs between a set of 10 registers and an ALU and shifter. The microcode also controls interaction between the main memory and two address/data register pairs, respectively providing access to the ISA-level bytecodes and to program data.

The DLSim 3 model uses plug-ins to implement the following functional units:

1. Microcode store and control unit. On startup, this unit reads a file containing a microprogram implementing the IJVM interpreter. In each instruction cycle, it loads a new microinstruction into its microinstruction register and breaks

it down into appropriate control signals which are wired to the components of the datapath. The plug-in view displays the current microinstruction and control signal values.

2. The 10-element register file. This unit contains 9 32-bit registers and 1 special-purpose 8-bit "MBR" register. The latter serves as the instruction register for interpreting IJVM byte code. Registers are labeled by their designated roles in the interpreter (e.g., SP: stack pointer; TOS: top-of-stack, etc.) or their roles as address (MAR, PC) or data (MDR, MBR) registers for memory operations. Register content is displayed by the plug-in view.

3. The main memory holding the IJVM program and data segments. The main memory is segmented into a method area, a constant pool and a stack area, each of which is displayed in a text area in the plug-in view. Reads and writes involving the two data segments interact with the 32-bit memory address (MAR) and data (MDR) registers in the register file. The method area is read-only and interacts with the PC address register and 8-bit MBR instruction register described above. The main memory visually tracks IJVM program execution and stack operations. The memory unit also contains a console for memory-mapped I/O, similar to the one described for the MIPS processor.

4. The ALU and shifter. Control signals can select from as many as 16 useful ALU operations, including addition, subtraction, and standard logical operations (AND, OR and NOT).

Other functions are implemented as subcircuits. These include a clock sequencer and JMP unit used to implement conditional control at the microcode level, as described in Tanenbaum [8].

The system clock cycle is divided into three time periods. The clock sequencer circuit generates three distinct pulses, respectively triggering the following operations:

Clock 0: The microinstruction register is loaded with the next microinstruction. Control signals are activated for subsequent actions in the cycle.

Clock 1: Registers are selected for gating onto the datapath. The ALU and shifter, using operations selected by control bits, process the selected register values. Any required memory operations are initiated. The value computed by the ALU and shifter is stored back in the register file.

Clock 2: Completed memory operations are synchronized.

Student exercises using this model fall into two categories. The first involves extending and experimenting with the microcode interpreter and its connection to the IJVM level; for example, we have assigned students the task of extending the microcode interpreter to implement several new IJVM instructions and to write an IJVM program to test them. The second involves direct manipulation of the simulated hardware. Students might be asked to replace ALU and shifter plug-ins with circuits built on the DLSim platform. (We have ourselves built an alternate implementation of *Mic-1* in which this has been carried out.) Finally, Tanenbaum describes several extensions to the *Mic-1* architecture that could readily be applied to extend the DLSim model.

STUDENT PROJECTS USING DLSIM 3

The processor models described here are suitable for a wide variety of student design projects. These may be assigned to the class as a whole, or as a term project where each student may choose his own project. In this section we describe some projects which have been carried out by students at Oberlin.

- **Programmable Logic Array (PLA).** This project involved writing a plug-in to simulate a PLA. The design was based on a PLA with 8 inputs, 50 and gates, and 6 outputs, but is parameterizable to any number of gates. It was used to implement the control logic for the *MicroMIPS* processor described earlier.

  The PLA "program" is a text file containing binary data representing the state of each programmable fuse on the PLA. The file can be loaded into the PLA through the DLSim 3 GUI. The same approach could be used in developing plug-ins to represent other types of programmable logic devices, such as FPGAs.

- **Cache Memory.** It is common in Computer Organization courses to assign as a programming assignment the simulation of a cache memory, parameterized by factors such as size, associativity, and replacement policy. Using DLSim 3, this project was written as a plug-in. It incorporates the usual types of cache parameterization, such as size, block size, associativity, and replacement policy. It has an interface which allows it to be inserted in the *MicroMIPS* CPU implementation between the main CPU circuit and its memory components. It also collects statistics on cache hits and misses.

  To complete this project, it was necessary to write a program to simulate the basic operation of the cache *and* to interface it to the CPU so that it would actually work on real programs. Thus it was a more complete project than simply writing a simulation program which reads a memory trace from a file.

  Using the cache plug-in in a CPU circuit does limit the scale of the simulations it can perform to programs written with the small instruction set of the *MicroMIPS*. To facilitate larger scale simulations, a TestGenerator plug-in was also written which reads a memory trace from a file and feeds it to the cache plug-in.

- **Calculator.** The calculator project involve the design of a hand calculator, backed by the *MicroMIPS* CPU. The design required addressing several issues:

  Design of a calculator plug-in with a view containing a display area and keypad.

  Design of an interface to the *MicroMIPS* processor, using memory-mapped I/O. Each keystroke on the calculator keypad triggers an input operation on the CPU. CPU output operations cause characters to be displayed on the calculator display.

  Software. A program was written in MIPS assembly language to support the calculator. It contains a polling loop which checks the status of the calculator. When a keystroke is available, it is read into memory

from the calculator's data register. A subroutine is then called to process the keystroke.

OTHER APPLICATIONS OF DLSIM 3

This paper has focused on the use of DLSim 3 at the processor design level. It is a versatile tool, however, that can be used at every level of study in a Computer Organization course.

1. **Conventional Logic Circuit Design**. Without plug-ins, DLSim 3 can be used as a low-level logic design simulator. The basic building blocks of gates and flip-flops can be used to implement any combinational or sequential circuit. Some projects of this type, which make use of DLSim 3's card and chip abstractions, are described in [6].

   Conventional circuit design projects can be given added impact by incorporating them into a CPU-level platform. For example, a typical assignment at this level is to design an ALU at the gate level. With DLSim 3, students could be given a working CPU with its ALU implemented as a plug-in. Their assignment would be to replace the plug-in with a chip (using the identical pin interface) of their own design. The full CPU could then be used as the test platform for the chip.

   Other CPU components, such as a register or multiplexer, could likewise be used as design exercises. The *MicroMIPS* control unit is simple enough to be used as an exercise, built either with gates or a PLA plug-in.
2. **Assembly Language Programming.** The *MicroMIPS* and *Mic-1* platforms implemented in DLSim 3 are suitable for exercises in assembly language programming. The simulated CPUs provide a test and execution platform for programs which have been translated into machine code by an assembler. For the programs used in development of our MIPS implementation, we used the MARS assembler [9]. MARS has its own GUI-based execution environment but also supports a non-GUI interface which translates an assembly language program into machine code in a text file which can be loaded into the DLSim 3 memory plug-in and then executed by the simulated processor.
3. **Higher-level System Design.** One of our goals is to write a plug-in to represent a complete CPU. A variety of system design exercises could be constructed using this plug-in, including bus design (both data transfer and arbitration), multiprocessor design, and cache memory design.

SUMMARY

DLSim 3 is a highly versatile software tool designed for the design and simulation of digital logic circuits. Through its use of software plug-ins, it is capable of simulating digital circuits of varying levels of complexity. The examples presented in this paper demonstrate some of the ways that DLSim 3 can be used as a pedagogical aid in Computer Organization courses. In particular, we have been successful in using it to simulate the CPU designs presented in the popular textbooks of Tanenbaum and Patterson and Hennessy, in order to demonstrate a variety of implementation techniques; for example, hard-wired control, microprogramming, and pipelining. We intend to continue to expand our library of plug-ins and circuit designs, which are available, along with the DLSim 3 software, at our website www.dlsim.com.

REFERENCES

[1] Masson, A, Logicsim, http://wuarchive.wustl.edu/edu/math/software/mac/LogicSim/, 1996.

[2] Poplawski, D. A., "A pedagogically targeted logic design and simulation tool," In WCAE'07, Proceedings of the 2007 workshop on Computer architecture education, pages 1–7, June 2007.

[3] Barker, D. L., Digital works 3.0, http://matrixmultimedia.com/datasheets/ eldwk.pdf, 2006.

[4] Burch, C., "Logisim: A graphical system for logic circuit design and simulation," J. Educ. Resour. Comput., 2(1):5–16, 2002.

[5] Burch, C., Logisim 2.1.6. http://ozark.hendrix.edu/~burch/logisim, 2007.

[6] Salter, R. M., and Donaldson, J.L., "Abstraction and Extensibility in Digital Logic Simulation Software," *ACM SIGCSE Bulletin, 41, 1* (Mar. 2009), 418-422.

[7] Patterson, D. A., and Hennessy, J., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2007.

[8] Tanenbaum, A. S., *Structured Computer Organization*, Prentice Hall, 2005.

[9] Vollmar, K., and Sanderson, P., "MARS: An Education-oriented Assembly Language Simulator," *ACM SIGCSE Bulletin, 38, 1* (Mar. 2006), 222-226.