Pep8CPU:
A Programmable Simulator for a Central Processing Unit

J. Stanley Warford
Pepperdine University
24255 Pacific Coast Highway
Malibu, CA 90265
Stan.Warford@pepperdine.edu

Ryan Okelberry
Novell
1800 South Novell Place
Provo, UT 84606
rokelberry@novell.com

ABSTRACT
This paper presents a software simulator for a central processing unit. The simulator features two modes of operation. In the first mode, students enter individual control signals for the multiplexers, function controls for the ALU, memory read/write controls, register addresses, and clock pulses for the registers required for a single CPU cycle via a graphical user interface. In the second mode, students write a control sequence in a text window for the cycles necessary to implement a single instruction set architecture (ISA) instruction. The simulator parses the sequence and allows students to single step through its execution showing the color-coded data flow through the CPU. The paper concludes with a description of the use of the software in the Computer Organization course and its availability for download on the Internet.

Categories and Subject Descriptors
B.1.5 [**Control Structures and Microprogramming**]: Microcode Applications – *Instruction set interpretation.* C.0 [**Computer Systems Organization**]: General – *Modeling of computer architecture*. K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*.

General Terms
Design, Languages

Keywords
Pep/8, CPU, Data section, Simulation, Virtual machine, Microprogramming, Assembly language, MIPS

1. INTRODUCTION
Virtual machines in computer science are used both for teaching concepts and for implementing algorithms in industry. [4] The most famous virtual machine for teaching is probably Knuth's MIX [1], and the most widespread machine for implementing algorithms in industry is probably the Java virtual machine [2]. Pep/8 is a virtual machine for teaching computer systems concepts [7] based on the seven levels of abstraction popularized by Tanenbaum [5]: application, high-order language, assembly, operating system, instruction set architecture (ISA), microcode, and logic gate.

For a number of years we have used an assembler/simulator for Pep/8 in the Computer Systems course to give students a hands-on experience at the high-order language, assembly, and ISA levels. This paper presents a software package developed by an undergraduate student, now a software engineer at Novell, who took the Computer Organization course and was motivated to develop a programmable simulator at the microcode level.

Yurcik gives a survey of machine simulators [8] and maintains a Web site titled Computer Architecture Simulators [9] with links to papers and internet sources for machine simulators. Most simulators for teaching computer organization at the undergraduate level have a command line interface. Those that have a graphical user interface are usually for machines with smaller instruction sets and fewer addressing modes than Pep/8.

Section 2 describes the Pep/8 computer at the assembly, ISA, and microcode level. Section 3 describes the CPU simulator and shows some of its capabilities. Section 4 gives examples of how the simulator is used in the Computer Organization course. The concluding section includes details about the simulator's availability.

2. THE Pep/8 VIRTUAL MACHINE

2.1 The assembly and ISA levels
Pep/8 is a classical 16-bit von Neumann computer with an accumulator (A), an index register (X), a program counter (PC), a stack pointer (SP), and an instruction register (IR). It has eight addressing modes: immediate, direct, indirect, stack-relative, stack-relative deferred, indexed, stack-indexed, and stack-indexed deferred. The instruction set is based on an expanding opcode yielding a total of 39 instructions, which come in two flavors – unary and nonunary. The unary instructions consist of a single 8-bit instruction specifier, while the nonunary instructions have the instruction specifier followed by a 16-bit operand specifier (OpSp).

For the nonunary instructions, the addressing modes determine the operand from the operand specifier as follows:

Addressing mode	Operand
Immediate	OpSp
Direct	Mem [OpSp]
Indirect	Mem [Mem [OpSp]]
Stack-relative	Mem [SP + OpSp]
Stack-relative deferred	Mem [Mem [SP + OpSp]]
Indexed	Mem [OpSp + X]

SIGCSE'07, March 7?11, 2007, Covington, Kentucky, USA.

Stack-indexed	Mem [SP + OpSp + X]
Stack-indexed deferred	Mem [Mem [SP + OpSp] + X]

At the assembly level, students learn the compilation process by translating small programs to Pep/8 assembly language. The Pep/8 addressing modes illustrate the memory model of the high-order language. For example, global variables are accessed with direct addressing, local variables with stack-relative addressing, local arrays with stack-indexed addressing, and dynamic variables via pointers with indirect addressing.

The following code fragment is an example of what Pep/8 assembly language looks like.

```
         BR      main
data:    .EQUATE 0         ;struct field
next:    .EQUATE 2         ;struct field
;
;******* main ()
first:   .EQUATE 4         ;local variable
p:       .EQUATE 2         ;local variable
value:   .EQUATE 0         ;local variable
main:    SUBSP   6,i       ;allocate locals
         LDA     0,i       ;first = 0
         STA     first,s
         DECI    value,s   ;cin >> value
while:   LDA     value,s   ;while (value != -9999)
         CPA     -9999,i
         BREQ    endWh
         LDA     first,s   ;   p = first
         STA     p,s
         LDA     4,i       ;   first = new node
         CALL    new
         STX     first,s
         LDA     value,s   ;   first->data = value
         LDX     data,i
         STA     first,sxf
```

The fragment is part of a program that is a translation of a C++ program containing local pointers. An assembly language symbol such as `first` corresponds to the variable identifier in the C++ program and equates to its offset on the run-time stack. The letter `i` specifies immediate addressing, the letter `s` specifies stack-relative addressing, and the letters `sxf` specify stack-relative deferred addressing.

Most virtual machines for teaching have software support to provide students with hands-on experience in programming at the assembly level, and Pep/8 is no exception. In the Computer Systems course, students are given complete C++ programs and required to translate them as a compiler would. They have access to an assembler that translates their programs to the ISA level, which they can then test on a virtual machine simulator.

The Computer Systems course is a prerequisite for the Computer Organization course. So, students in Computer Organization already have a working knowledge of the high-order language, assembly language, and ISA levels of a von Neumann machine based on an implementation of the Pep/8 computer. Pep8CPU is a software package that takes students down to the microcode level in the Computer Organization course.

2.2 The microcode level

Figure 1 shows the data section of the Pep/8 central processing unit. It consists of the following parts:

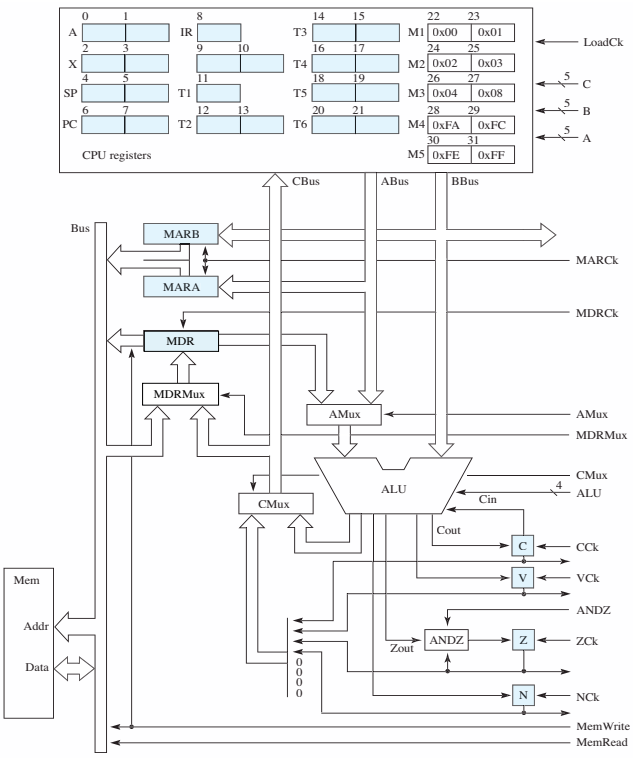

Figure 1.

- A two-port register bank
- A 16-function arithmetic logic unit (ALU)
- A set of four status bit latches (NZVC)
- A memory address register (MAR)
- A memory data register (MDR)
- Three 8-bit multiplexers (AMux, CMux, MDRMux)
- A three-input, one-output combinational circuit (ANDZ)

The control section, not shown in the figure, is to the right of the data section. The control lines coming in from the right are from the control section. There are two kinds of control signals – combinational circuit controls and clock pulses. Names of the clock pulses all end in Ck and all act to clock data into a register or latch. For example, MDRCk is the clock input for the MDR. When it is pulsed, the input from the MDRMux is clocked into the MDR.

The two-port register bank has 32 8-bit registers. Because Pep/8 is a 16-bit machine, the accumulator consists of the two registers at register addresses 0 and 1. The index register consists of registers 2 and 3, and so on for the stack pointer, program counter, and instruction register. Registers 11 through 21 are temporary scratch pad registers not visible to the programmer at the assembly or ISA levels. Registers 22 through 31 are read-only memory (ROM) that are convenient for masking operations.

All the thick buses in the figure, except for the system bus, are eight bits wide. The five lines labeled A provide an address to the register bank that selects one of the 32 registers to be placed on ABus. Similarly, the five lines labeled B select a register for BBus.

The ALU is strictly combinational with no storage. The four lines labeled ALU select which of the 16 functions the ALU will perform. Depending on the function selected, the ALU may use the carry-in bit Cin for its computation. The ALU produces an 8-bit result (bottom left of the ALU) and four status values NZVC – for negative, zero, overflow, and carry – which can be clocked into the data latches with NCk, ZCk, VCk, or CCk respectively. ANDZ controls whether Zout from the ALU is presented directly to the Z latch or is ANDed with the current value of Z.

The 16 functions of the ALU are

0	A	8	$\overline{A + B}$
1	A plus B	9	$A \oplus B$
2	A plus B plus Cin	10	$\overline{A}$
3	A plus $\overline{B}$ plus 1	11	ASL A
4	A plus $\overline{B}$ plus Cin	12	ASR A
5	$A \cdot B$	13	ROL A
6	$\overline{A \cdot B}$	14	ROR A
7	A + B	15	0

where $\cdot$ is logical AND, $+$ is logical OR, $\oplus$ is exclusive OR, and the overbar is logical complement.

The bus on the left is the main system bus that connects the CPU to main memory, which is in the lower left of the figure. The system bus consists of 16 address lines, 8 bidirectional data lines, and MemRead and MemWrite control signals.

A multiplexer control line routes the signal through a multiplexer as follows:

- 0 routes the left input to the output
- 1 routes the right input to the output

For example, if the MDRMux control line is 0, MDRMux routes the data from the system Bus to the MDR. If the control line is 1 it routes the data from CBus to the MDR.

Students learn how a sequence of control statements at the microcode level implements a single instruction at the ISA level. A simple control sequence language specifies a control sequence. In the language, an equals sign indicates a combinational control signal from the control section of the CPU, comma is the concurrent separator, semicolon is the sequential separator, and `//` indicates a comment that is ignored by the micro assembler.

For example, the following microcode sequence implements the store byte instruction with direct addressing.

```
// MAR <- OprndSpec.
1. A=9, B=10; MARCk
// MBR <- A<low>.
2. A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
// Initiate memory write.
3. MemWrite
// Complete memory write.
4. MemWrite
```

The operand specifier is in the instruction register at register addresses 9 and 10 in the register bank. Because the instruction uses direct addressing, the instruction specifier is the address of the operand.

Cycle 1 transfers the operand specifier into the MAR. `A=9` puts the high-order byte on ABus, `B=10` puts the low-order byte on BBus, and `MARCk` clocks ABus and BBus into the MAR registers.

Cycle 2 transfers the low-order byte of the accumulator into the MDR. `A=1` puts the low-order byte of the accumulator onto ABus. `AMux=1` routes it through the AMux into the ALU. `ALU=0` passes it through the ALU unchanged. `CMux=1` routes it onto the CBus. `MDRMux=1` routes it through MDRMux to the MDR, and `MDRCk` latches the data into the MDR.

Cycles 3 and 4 complete the memory write, storing the data that is in the MDR to main memory at the address that is in the MAR. In the Pep/8 system memory reads and writes require two consecutive cycles to illustrate the speed penalty of a main memory access compared to a register access.

Pep/8 is a 16-bit computer, but the data section of the CPU is based on 8-bit registers and internal buses. To implement a 16-bit ISA instruction requires two cycles at the microcode level. For example, the following microcode sequence implements the add instruction with immediate addressing.

```
// A<low> <- A<low> + Oprnd<low>. Save carry.
1. A=1, B=10, AMux=1, ALU=1, ANDZ=0, CMux=1, C=1;
   ZCk, CCk, LoadCk
// A<high> <- A<high> plus Oprnd<high> plus
//    saved carry.
2. A=0, B=9, AMux=1, ALU=2, ANDZ=1, CMux=1, C=0;
   NCk, ZCk, VCk, CCk, LoadCk
```

The instruction sets N to 1 if the two-byte result is negative; otherwise it clears N to 0. It sets Z to 1 if the two-byte result is all zeros; otherwise, it clears Z to 0. So, unlike the N bit, the values of both the high-order and the low-order bytes determine the value of Z.

3. THE CPU SIMULATOR

While an undergraduate student, the second author implemented Pep8CPU with the C++ class libraries of Trolltech's Qt development system. [6] The application runs in one of two modes – a single-cycle mode and a programmable mode.

3.1 The single-cycle mode

Figure 2 is a screen shot of the application. Each control signal has a GUI widget that the user can set. A clock pulse signal has a check box to indicate that a clock pulse for that sequential circuit will be applied at the end of the cycle. In the figure, LoadCk is checked to indicate that a value from CBus will be clocked into the register bank.

A text input widget allows the user to enter a decimal input value for a control signal. In the figure, the user has entered 1 for the five-line A control signal, which represents register address 00001 in binary, the low-order byte of the accumulator. She has entered 24 for B, which represents register address 11000, ROM register with constant value 2. She has entered 1 for the ALU function, which selects A plus B. The instant a value is entered for an ALU function, the operation appears on the ALU block. In the figure the text "A plus B" appears on the ALU giving immediate visual feedback on which function is selected.

Text input widgets for the multiplexers allow the user to enter 0 or 1 to select the multiplexer input according to the convention described above. Namely, 0 selects the left input, and 1 selects the

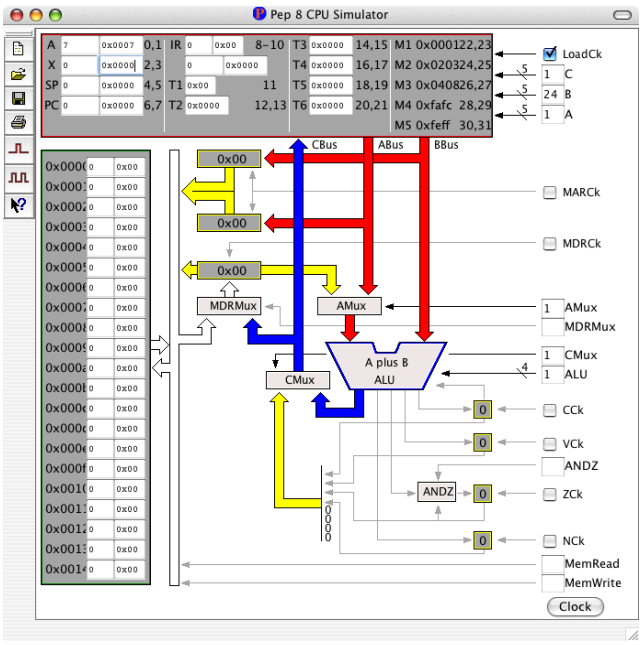

Figure 2.

right input. In the figure, the user entered 1 for AMux selecting ABus to be routed to the ALU, and 1 for CMux selecting the result from the ALU to be routed to CBus.

Text input widgets are also available for the user to enter values in the registers in the register bank and in the memory cells of main memory. Input text boxes for A, X, SP, PC, IR, and each main memory cell are arranged in pairs. The left cell of the pair is for decimal input and the right cell of the pair is for hexadecimal input. The instant a value is typed in one of the cells, the corresponding value appears in the other base in the other cell of that pair. For example, if the user enters "255" in the left cell of the accumulator, "0xff" appears instantly in the right cell. Alternatively, the user can enter "0xff" in the right cell and "255" will appear instantly in the left cell.

A single Clock push button is at the bottom right of the window. When the user presses it, Pep8CPU simulates the execution of a single CPU cycle. It verifies that meaningful control signals are present and issues a warning if they are not. For example, if the user wants to clock a value from CBus into the register bank, but has not entered a CMux control signal, a dialog box with a warning pops up explaining the problem.

A visually dramatic feature of the application that cannot be seen in black-and-white reproductions of this article is the color-coded rendering of the data flow. The register bank has a thin red outline around its outer perimeter. The ALU has a thin blue outline around it, MAR and MBR have thin yellow outlines, and the main memory block has a thin green outline. When the user puts a register address in the A text input widget, ABus immediately turns red. Similarly for the B text input widget and BBus. The buses emanating from MAR and MBR are always colored yellow. Consequently, AMux will have yellow input on the left from MDR and red input on the right from ABus. As soon as the user inputs 0 in the AMux text input widget, the output bus from AMux turns yellow, showing that "yellow" data from MDR is routed through AMux. If the user changes the AMux control signal to 1, the output of AMux immediately turns red, showing that "red" data from ABus is routed through AMux.

Similarly, the user can visualize "green" data flowing on the system bus from main memory through MDRMux and into MDR.

Another nice visual touch is the color of the control signal lines. They are usually gray, but when activated they turn black. In the figure you can see that LoadCk is checked and its control line is black, but MARCk is not checked and its control line is gray.

3.2 The programmable mode

While the single-step mode is good for visualizing data flow through the data section of the CPU with a single cycle, it is cumbersome for executing a sequence of control signals to implement a typical ISA instruction. The programmable mode provides a text editor and a micro assembler that permits students to write, execute, and save control sequences. Figure 3 is a screen shot of the microcode text editor. It shows the control sequence to fetch an instruction and increment PC as part of the von Neumann cycle.

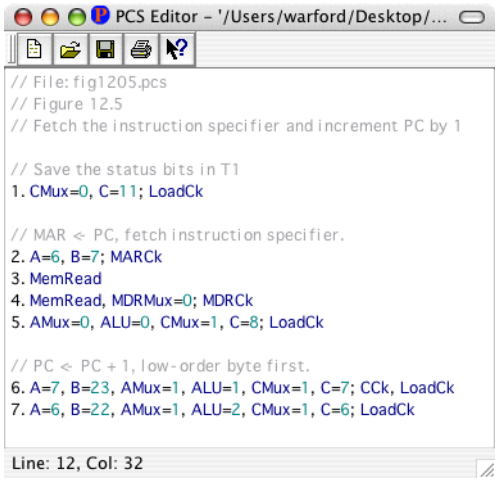

Figure 3.

The editor applies syntax-aware color coding to the text. Comments are colored gray, cycle number are black, constants are green and other valid code is blue.

After the user writes the control sequence she requests assembly. If there are no errors, the assembled code representing the sequence is loaded into the CPU simulator. The clock button at the bottom of the CPU window displays the cycle number, initially 1. Also, the input widgets are set according to the first cycle in the microprogram. At this point, all the color coded features are visible in the CPU window as if the user had entered the data manually into the input widgets.

With the first click of the clock button, cycle 1 executes and the student can inspect the effect of the cycle. The cycle number in the clock button changes to 2 and the input widgets are set according to the second cycle in the microprogram. Again, all the color coded features of the data flow are visible. With each click of the clock button the student traces the sequence visually and can detect and correct bugs in the control program accordingly.

4. USE IN THE COURSE

4.1 Lecture demonstration

The CPU simulator makes an excellent lecture demonstration in the Computer Organization course. The color coded data flow rendering as the teacher enters control signals in the input widgets draws the students' attention to that part of the CPU that is affected by the control signal.

One presentation technique that has proven effective is to write down an assembly language instruction on the board and then to go around the classroom asking each student in turn which control signal should be set, and to what value, to achieve an implementation of the instruction.

4.2 Homework assignments

Section 2.2 showed the control sequence for the store byte instruction with direct addressing. One instructive assignment is to require implementation of the same instruction but with one of the other addressing modes. For example, to implement indirect addressing requires fetching two bytes from memory which represent an address. That value must be sent to MAR in preparation for the memory fetch that finally gets the operand.

Other homework exercises illustrate basic von Neumann organization principles. For example, execution of the branch instruction requires that the program counter be replaced by the operand. The text [7] has a set of 24 exercises from which assignments can be given. Pep8CPU provides students with a tool to check their solutions. Furthermore, their solutions can be handed in electronically as text files and tested by the grader on the simulator.

4.3 Performance issues

After students have implemented several instructions on Pep8CPU they see that some aspects of the organization could be improved. The course explores two possible approaches to increasing performance: specialized hardware units and increased data bus width.

To access adjacent bytes in memory requires incrementing a 16-bit address. Because the ALU performs 8-bit additions, and because there is no direct data path from the ALU to MAR, such computations consume many cycles. A special purpose 16-bit incrementer can be inserted near the MAR to achieve a reduction in the number of cycles.

The course also explores the consequences of increasing the data bus width from 8 to 16 bits in the main system bus. Students study a modification that requires two 8-bit MDRs and explore memory alignment issues that arise from such a design.

Both of these performance improvements are paper studies. Students are given assignments to re-implement an assembly language instruction and compute the percentage increase in performance based on the decreased number of cycles. However, they cannot test their implementations on a simulator. Future work on Pep8CPU might incorporate additional simulators to realize these alternate designs.

4.4 MIPS comparison

The architecture of Pep/8 is motivated primarily by one goal: simplicity in teaching the classic von Neumann machine, in particular simplicity at the high-order language and assembly levels. The eight addressing modes are designed to teach the memory model of an imperative language by making it easy to translate high-level language programs to assembly language.

This design puts Pep/8 squarely in the CISC architecture camp. The Computer Organization course also teaches the MIPS architecture as an example of a RISC design. [3] Although the MIPS literature does not use the ABus/BBus/CBus terminology it is possible to present the MIPS CPU with that terminology in a form that closely resembles the Figure 1. Once students are familiar with the data flow in the Pep/8 CPU it is not difficult for them to understand the data flow in the MIPS CPU.

The comparison of CISC versus RISC at this point in the course is is really meaningful for students who have used Pep8CPU. They have implemented CISC instructions that require many cycles and are amazed when they see a machine design that requires only *one* cycle for each ISA instruction. The course presents cache memory and pipelining in the context of the MIPS machine.

5. CONCLUSION

While most machine simulators operate at the assembly language and ISA level, Pep8CPU gives students hands-on experience at the microcode level. The software is available free of charge under the GNU General Public License at

`ftp://ftp.pepperdine.edu/pub/compsci/pep8`

It was developed with the Qt 3 class library. [6] Unlike many free simulators for teaching computer organization, the application is available not only in source code form, but with executable builds for Windows, Mac OS X, and Linux. At the time of this writing we are in the process of converting the application to Qt 4. See the above link for the current status of the project.

6. REFERENCES

[1] Knuth, D. E. *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1997.

[2] Lindholm, T. and Yellin, F. *The Java(TM) Virtual Machine Specification*. Sun Microsystems, Inc., Palo Alto, CA, 1999.

[3] Patterson, D. A. and Hennessy, J. L. *Computer Organization and Design*. Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[4] Rosenblum, M. The Reincarnation of Virtual Machines, *ACM Queue*, vol. 2, no. 5, July/August, 2004.

[5] Tanenbaum, A. S. *Structured Computer Organization*. Pearson Prentice Hall, Upper Saddle River, NJ, 2006.

[6] Trolltech ASA, Oslo, Norway, `http://trolltech.com/`

[7] Warford, J. S. *Computer Systems*. Jones and Bartlett Publishers, Inc. Sudbury, MA, 2005.

[8] Yurcik, W., Wolffe, and Holliday A Survey of Simulators used in Computer Organization/Architecture Courses, *Proceedings of the 2001 Summer Computer Simulation Conference*, 2001.

`[9] http://www.sosresearch.org/caale/caalesimulators.html`