The Pep/8 Memory Tracer: Visualizing Activation Records on the Run-Time Stack

J. Stanley Warford
Pepperdine University
24255 Pacific Coast Highway
Malibu, CA 90263
Stan.Warford@pepperdine.edu

Chris Dimpfl
Pepperdine University
24255 Pacific Coast Highway
Malibu, CA 90263
Christian.Dimpfl@pepperdine.edu

ABSTRACT

This paper presents a virtual machine simulator with a memory trace facility having two unique features. First, the machine is designed to illustrate the translation from C/C++ to assembly language and from thence to machine language. Instead of the more common memory dump labeled by address, the tracer displays a graphic representation of the cell labeled by its symbol. Second, the simulator displays in real time the growth of the run-time stack on function activation, detects and displays the boundaries of the activation record, and displays all the cells on the run-time stack labeled by symbol. The paper includes download information for the open-source application.

Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids – *Simulation*. 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, Simulation, Virtual machine, Memory model, Assembly language, Activation record, Visualization, Run-time stack, Heap

1. INTRODUCTION

Virtual machines in computer science are used both for teaching concepts and for implementing algorithms in industry. Yurcik gives a survey of machine simulators in [14], as does Rosenblum in [10]. The most famous virtual machine for teaching is probably Knuth's MIX [3], and the most widespread machine for implementing algorithms in industry is probably the Java virtual machine. [4] Pep/8 is a virtual machine for teaching computer systems concepts [13] based on the seven levels of abstraction popularized by Tanenbaum [12]: application, high-order language, assembly, operating system, instruction set architecture (ISA), microcode, and logic gate.

A common topic in programming language texts such as that by Ghezzi [2] is the implementation of function activation with an activation record, also known as a stack frame, on the run-time stack. Machines are invariably designed to support function activation through architecture features that provide for maintaining the run-time stack. Simulators for teaching computer architecture principles typically include such features. Examples are the SIMPLESIM simulator for the book by Ghezzi [2], the MARIE simulator for the book by Null [6], and the LC-3 simulator for the book by Patt [7]. The classic text by Patterson [8] has spawned many simulators for the MIPS machine, two of which are SPIM [11] and MARS [5].

The memory trace facility of the Pep/8 simulator has two unique features compared to the above simulators. First, the machine is designed to illustrate the translation from C/C++ to assembly language and from thence to machine language. Variable names at the C/C++ level correspond to symbols at the assembly level and to memory addresses at the machine level. Instead of the more common memory dump labeled by address, the tracer displays a graphic representation of the cell labeled by its symbol.

Second, the simulator displays in real time the growth of the run-time stack on function activation, detects and displays the boundaries of the activation record, and displays all the cells on the run-time stack labeled by symbol. The resulting display is similar to figures in textbooks that describe the function activation process. These features are in addition to the availability of the standard memory dump.

Section 2 describes the Pep/8 computer at the assembly and ISA level. Section 3 describes the simulator and shows the symbolic memory trace capabilities. 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 small 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 instruc-

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

tion 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 (`i`)	OpSp
Direct (`d`)	Mem [OpSp]
Indirect (`n`)	Mem [Mem [OpSp]]
Stack-relative (`s`)	Mem [SP + OpSp]
Stack-relative deferred (`sf`)	Mem [Mem [SP + OpSp]]
Indexed (`x`)	Mem [OpSp + X]
Stack-indexed (`sx`)	Mem [SP + OpSp + X]
Stack-indexed deferred (`sxf`)	Mem [Mem [SP + OpSp] + X]

At the assembly level, students learn the compilation process by translating complete C++ programs to Pep/8 assembly language. The Pep/8 addressing modes facilitate teaching the C/C++ memory model:

- Global variables are allocated at a fixed location in memory.
- Parameters and local variables are allocated on the run-time stack.
- Dynamic variables are allocated at run time from the heap.

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.

For example, the following complete program in C++

```
#include <iostream>
using namespace std;

int a, b;

void swap (int& r, int& s) {
   int temp;
   temp = r;
   r = s;
   s = temp;
}

void order (int& x, int& y) {
   if (x > y) {
      swap (x, y);
   }
}

int main () {
   cout << "Enter an integer: ";
   cin >> a;
   cout << "Enter an integer: ";
   cin >> b;
   order (a, b);
   cout << "Ordered they are: " << a << ", "
        << b << endl;
   return 0;
}
```

is translated to the following Pep/8 assembly language program.

```
         BR      main
a:       .BLOCK  2           ;global variable #2d
b:       .BLOCK  2           ;global variable #2d
;
;******* void swap (int& r, int& s)
r:       .EQUATE 6           ;formal parameter #2h
s:       .EQUATE 4           ;formal parameter #2h
temp:    .EQUATE 0           ;local variable #2d
swap:    SUBSP   2,i         ;allocate #temp
         LDA     r,sf        ;temp = r
         STA     temp,s
         LDA     s,sf        ;r = s
         STA     r,sf
         LDA     temp,s      ;s = temp
         STA     s,sf
         RET2                ;deallocate #temp
;
;******* void order (int& x, int& y)
x:       .EQUATE 4           ;formal parameter #2h
y:       .EQUATE 2           ;formal parameter #2h
order:   LDA     x,sf        ;if (x > y)
         CPA     y,sf
         BRLE    endIf
         LDA     x,s         ;   push x
         STA     -2,s
         LDA     y,s         ;   push y
         STA     -4,s
         SUBSP   4,i         ;   push #r #s
         CALL    swap        ;   swap (x, y)
         ADDSP   4,i         ;   pop #s #r
endIf:   RET0                ;pop retAddr
;
;******* main ()
main:    STRO    msg1,d      ;cout << "Enter ..."
         DECI    a,d         ;cin >> a
         STRO    msg1,d      ;cout << "Enter ..."
         DECI    b,d         ;cin >> b
         LDA     a,i         ;push the address of a
         STA     -2,s
         LDA     b,i         ;push the address of b
         STA     -4,s
         SUBSP   4,i         ;push #x #y
         CALL    order       ;order (a, b)
         ADDSP   4,i         ;pop #y #x
         STRO    msg2,d      ;cout << "Ordered ..."
         DECO    a,d         ;     << a
         STRO    msg3,d      ;     << ", "
         DECO    b,d         ;     << b
         CHARO   '\n',i      ;     << endl
         STOP
msg1:    .ASCII  "Enter an integer: \x00"
msg2:    .ASCII  "Ordered they are: \x00"
msg3:    .ASCII  ", \x00"
         .END
```

Variables `a` and `b` are global. Global variables are allocated at assembly time with the `.BLOCK` dot command. The statement `a: .BLOCK 2` allocates two bytes for `a`. The decimal input instruction `DECI a,d` inputs the value `a` using direct addressing indicated by the letter `d`. On the other hand, `x` and `y` are parameters in the `order` function. Stack variables are allocated with the subtract SP instruction. The instruction `SUPSP 4,i` in `main` allocates two bytes for `x` and two for `y` where the `i` indicates immediate address-

ing. The load accumulator instruction `LDA x,s` in the `order` function uses stack-relative addressing indicated by the letter `s`.

2.2 Trace tags

To trace a variable, the programmer embeds trace tags in the comments associated with the variables and single steps through the program. There are two kinds of trace tags:

- Format trace tags
- Symbol trace tags

Trace tags are contained in assembly language comments and have no effect on generated object code. Each trace tag begins with the `#` character and supplies information to the symbol tracer on how to format and label the memory cell in the trace window. Trace tag errors show up as warnings when the code is assembled, allowing program execution without tracing turned on. However, they do prevent symbolic tracing until they are corrected.

The global tracer allows the user to specify which global symbol to trace by placing a format trace tag in the comment of the `.BLOCK` line where the global variable is declared. For example, the line from the above program

```
a:  .BLOCK 2    ;global variable #2d
```

has format trace tag `#2d`, which is read as "two byte, decimal." This trace tag tells the symbol tracer to display the content of the two-byte memory cell at the address specified by the value of the symbol `a` formatted as a decimal value.

The legal format trace tags are:

`#1c` One-byte character

`#1d` One-byte decimal

`#2d` Two-byte decimal

`#1h` One-byte hexadecimal

`#2h` Two-byte hexadecimal

Local variables are allocated on the run-time stack with the `SUBSP` instruction. They are defined by their offset from the top of the stack with the `.EQUATE` dot command. In the above program, the lines

```
x:   .EQUATE 4    ;formal parameter #2h
y:   .EQUATE 2    ;formal parameter #2h
```

define `x` at four bytes below the top of the stack and `y` at two bytes below the top. Each of them is given a two-byte hexadecimal format because that is appropriate for addresses. (Note that they are called by reference.) At run time, `x` and `y` are allocated on execution of the instruction

```
    SUBSP  4,i   ;push #x #y
```

and deallocated on execution of the instruction

```
    ADDSP  4,i   ;pop #y #x
```

These instructions show the use of symbol trace tags `#x` and `#y`. The programmer specifies which local variables or parameters are being allocated on the run-time stack with the `SUBSP` instruction and which are being deallocated with the `ADDSP` instruction. The stack tracer uses the symbol trace tags to label the memory cells in the memory trace pane of the application. Furthermore, the assembler verifies that the number of bytes being pushed by the `SUBSP` instruction corresponds to the total number of bytes taken from the format trace tags in the `.EQUATE` statements. If they do not match, a trace tag warning is issued and the symbolic tracer is disabled.

Arrays are handled by an extension to the format trace tags. The letter `a` indicates that a variable is an array, and is prefixed by the number of elements in the array. For example, the global array declared in C++ as

```
int vector[4];
```

is translated to Pep/8 assembly language as

```
vector: .BLOCK 8  ;global variable #2d4a
```

where the format trace tag `#2d4a` is read as "two-byte decimal, four-cell array." The assembler verifies that the number of bytes allocated by the `.BLOCK` (eight) is equal to the product of the cell size and the number of cells (two times four).

If the C++ `vector` array is declared as a local inside the `main` function as

```
int main () {
   int vector[4];
   int j;
```

the corresponding Pep/8 assembly language code fragment is

```
vector:  .EQUATE 2  ;local variable #2d4a
j:       .EQUATE 0  ;local variable #2d
main:    SUBSP 10,i ;allocate #vector #j
```

In this example, symbol `vector` equates to two, which is the offset of the first cell of the array, `vector[0]`, from the top of the stack. Similarly, local variable `j` has offset 0 from the top. As before, the assembler verifies that the total number of bytes allocated by `SUBSP` corresponds to the number of bytes specified in the format trace tags.

We encountered a pleasant serendipity with the implementation of trace tags in the Pep/8 assembler. Previously, we tried to enforce consistent documentation of assembly language code when students translated programs from C++. A nice side effect of trace tags is that the assembler forces students to document their code with meaningful comments. To avoid trace tag warnings, they must specify in their comments the number of bytes allocated for each variable as well as which variables are being allocated for each `SUBSP`. We now require all assembly language programs to contain trace tags with no warning messages from the assembler.

3. THE Pep/8 SIMULATOR

Previous versions of Pep/8 suffered from years of accumulated code added in an ad hoc manner. To add the symbolic memory trace feature the authors, the second of whom is an undergraduate student, rewrote the application from scratch based on the mediator object-oriented design pattern. [1] It is implemented with the C++ class libraries of Nokia's Qt development system. [9]

3.1 Memory tracing with scalar variables

Figure 1 is a screen shot of the application during a trace of the assembly language program in Section 2.1.

The Listing Trace pane, top left, is generated from the source code upon successful assembly. The source code is not visible in this screen shot, but is accessible by clicking the Code tab.

The CPU pane, top right, shows the current content of the registers in hexadecimal and decimal notation. The programmer is single-

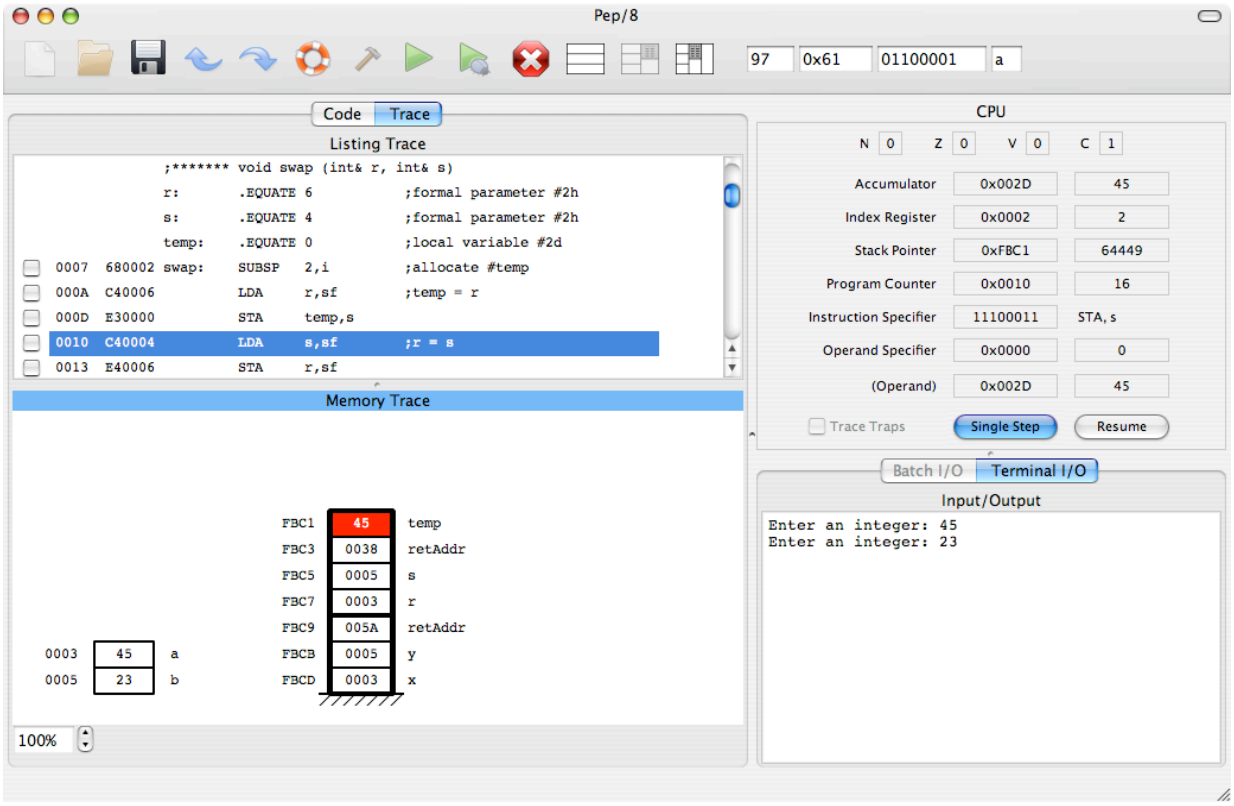


Figure 1. The Pep/8 main window.

stepping through the code and the current value of the program counter is 0x0010. Consequently, the instruction at address 0x0010 in the Listing Trace pane is highlighted, indicating that the `LDA` instruction will execute in the next cycle.

The Instruction Register is divided into two parts—a one-byte Instruction Specifier and, for non-unary instructions, a two-byte Operand Specifier. The CPU pane displays the Instruction Specifier in binary, to help the programmer visually decode the expanding op-code if desired. It also decodes the opcode and the addressing mode and displays the corresponding instruction mnemonic and letter indicating the addressing mode. Figure 1 shows that the `STA` instruction has just executed with stack-relative addressing.

The Pep/8 simulator supports both batch and interactive I/O as can be seen in the I/O pane, bottom right. When prompted, the user entered the two integers 45 and 23 in interactive mode.

The Memory Trace pane, lower left, displays the values of both global and local variables. The global variables, `a` and `b`, are allocated at a fixed location in memory and appear on the left. To the left of each cell is the memory address and to the right is the symbol taken from the trace tag. The format of the value within the cell in this case is decimal, also taken from the trace tag.

The run-time stack is to the right of the globals. As with the globals, each cell on the run-time stack has its address to the left of the cell and its symbol to the right. The `CALL` instruction pushes the incremented value of the PC onto the run-time stack. By default, it generates a label of `retAddr`, which stands for return address. Corresponding to the fact that the `STA` instruction has just stored

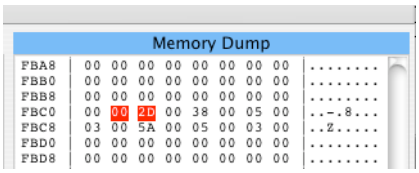


Figure 2. A traditional memory dump.

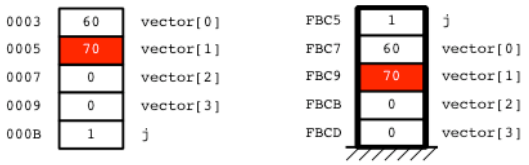
the value of 45 to local variable `temp`, that cell is highlighted in red on top of the stack.

Figure 1 shows two activation records. First is the activation record for function `order` consisting of the cells for parameters `x` and `y`, and for `retAddr`. Second is the activation record for function `swap` consisting of the cells for parameters `r` and `s`, for `retAddr`, and for local variable `temp`. Figure 2 illustrates how dramatic this visualization is by showing how the programmer confronts the same information in the traditional memory dump, also available in the Pep/8 application.

The functions traced in Figure 1 are both void. The symbolic memory trace handles non-void functions as well, the only difference being an extra cell at the bottom of the activation record to hold the returned value, also specified by the programmer with trace tags. Our naming convention is to label that value `retVal`.

3.2 Memory tracing with array variables

Figure 3 shows two separate screen shots of the Memory Trace pane that correspond to the code fragments for `vector` in Section 2.2. Part (a) shows the case where `vector` is declared to be global



(a) Global **(b) Local**

Figure 3. Tracing an array

and is allocated with the `.BLOCK` dot command at load time. Part (b) shows the case where `vector` is declared to be local and is allocated with the `SUBSP` instruction at run time.

3.3 Tracing the heap

Pep/8 also features a rudimentary heap that can be traced with the same trace tag system. Figure 4 shows a trace of an assembly language program that is a translation of a C++ program implementing a linked list. A node is a struct with two fields — `data`, which is an integer, and `next`, which is a pointer. The same frame that outlines an activation record on the runtime stack outlines a node on the heap.

A feature not illustrated in this paper but available with Pep/8, is the ability to trace a global `struct`. Each cell of the `struct` is rendered symbolically with a period separating the variable name and the field name.

4. CONCLUSION

While most machine simulators have a standard memory dump facility for tracing the content of memory during program execution, the Pep/8 simulator is unique in its ability to visually show the structure of the activation record on the run-time stack.

This feature has several pedagogic advantages. First, the system of trace tags that are necessary to enable the stack trace forces students to document their assembly language code more precisely than would otherwise be the case. Format trace tags require them to specify the number of bytes for each variable, and symbol trace tags require them to specify which variables are being allocated on the run-time stack. To do these specifications correctly, they must understand the structure of the activation record. This activity is a prime example of the educational principle that students learn better by doing than by simply studying.

A second advantage of the visual representation of activation records is its illustration of the mechanism of recursion. Space limitations in this article preclude showing a lengthy sequence of snapshots to illustrate how the run-time stack grows and shrinks during the execution of a recursive algorithm. Such demonstrations are both enlightening and entertaining in the classroom. A favorite exercise is to have students write the Towers of Hanoi problem in C++ and then translate it to assembly language and run it on the simulator. This is an example of learning the mechanism of recursion by programming the detailed structure of the activation record.

A third advantage is the reinforcement in students' minds of the memory model of C/C++. This paper shows how the Pep/8 simulator displays the memory structure for global variables at a fixed location, local variables and parameters on the run-time stack, and dynamic variables on the heap.

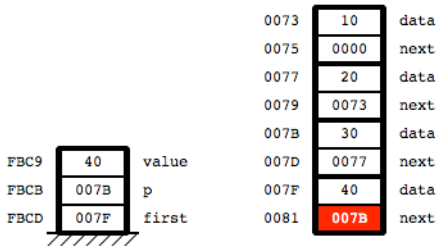


Figure 4. Tracing the heap.

Pep/8 software is available free of charge under the GNU General Public License at

http://code.google.com/p/pep8-1/

The site contains the source code in an SVN repository as well as executable builds for Windows, Mac OS X, and Linux. The Windows build runs under Windows XP, Vista, and System 7. The OSX build is a universal binary that runs on PowerPC and Intel Macs. The Linux build is for Ubuntu, but the source is easily compiled for other flavors of Linux with the Qt Creator IDE from Nokia [9], which we enthusiastically recommend for open-source, free, cross-platform development and deployment.

5. REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA, 1995.

[2] Ghezzi, C. and Jazayeri, M. *Programming Language Concepts*, third ed. John Wily & Sons, Inc., New York, NY, 1998.

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

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

[5] MARS, http://courses.missouristate.edu/KenVollmar/MARS/

[6] Null, L, and Lobur, J. Computer Organization and Architecture, second ed. Jones and Bartlett, Publishers, Inc. Sudbury, MA 2006.

[7] Patt, Y. N., and Patel, S. J. Introduction to Computing Systems, second ed. McGraw-Hill, New York, 2004.

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

[9] Qt, http://qt.nokia.com/

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

[11] SPIM, http://pages.cs.wisc.edu/~larus/spim.html

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

[13] Warford, J. S. *Computer Systems*. Jones and Bartlett Publishers, Inc. Sudbury, MA, 2010.

[14] Yurcik, W., ed. Special Issue on Specialized Computer Architecture Simulators, *Journal on Educational Resources in Computing*, vol. 2, no. 1, March 2002.