OAMulator: A Teaching Resource to Introduce Computer Architecture Concepts

FILIPPO MENCZER and ALBERTO MARIA SEGRE
The University of Iowa

The OAMulator is a Web-based resource to support the teaching of instruction set architecture, assembly languages, memory, addressing, high-level programming, and compilation. The tool is based on a simple, virtual CPU architecture, called the One Address Machine. A compiler allows us to take programs written in a special programming language, called OAMPL, and transform them into OAM assembly. An OAM assembler/emulator interprets and executes OAM assembly code produced by the compiler or written directly by students. The OAMulator is targeted at non-CS students who take introductory courses in information technology or information systems. Such students are normally exposed to concepts of computer hardware and software, but it is difficult for them to make the connection between the two. The OAMulator takes the mystery out of CPU architecture by letting students gain confidence with the concepts of compilers and binary execution. The Web-based deployment allows students to work on problems in convenient locations, at their own pace, and with rewarding interaction.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization—General**]: Modeling of Computer Architecture; I.6.5 [**Simulation and Modeling**]: Model Development; K.3.1 [**Computers and Education**]: Computer Uses in Education

General Terms: Design

Additional Key Words and Phrases: Computer architecture simulator, education

1. INTRODUCTION

There is a large population of students outside computer science departments, at both undergraduate and graduate levels, who need a basic understanding of how a computer works. Example courses include introduction to computer science for non-CS majors, introduction to information technology for MBAs, introduction to information systems for MIS students without technical background, and informatics "service" courses for students in health sciences, engineering, library science, education, business, law, and so on.

Such students are normally exposed to concepts of computer hardware and software, but it is difficult for them to make the connection between the two. So computer organization and hardware issues are often taught orthogonally

Authors' address: Department of Management Sciences, The University of Iowa, Iowa City, IA 52242. {filippo-menczer,alberto-segre}@uiowa.edu.

from software issues. This does not allow the students in such courses to fully appreciate the mutual impact and constraints of software and hardware architecture on each other. Thus, concepts such as code portability, platform dependence, application distribution, open source, optimization, CPU speed, pipelining, and the memory hierarchy often remain abstract ideas, easily manipulated and abused for marketing purposes.

Computer science students generally study computer architecture once they are familiar with high-level programming and compilation, through assembly languages and with the aid of emulators. This allows them to grasp at a deep level the connections among software, compilers and the instruction set architecture and the effects of all these components on application performance. There are several emulators available for this goal, for real architectures as well as fictitious ones designed for teaching purposes [Ponta and Donzellini 1994; Da Bormida et al. 1997; Scragg 1993, 1996]. However, the level of complexity of such emulators is too high for students without a solid computational background. On the other hand, no compilers and assembler emulators have been developed for simplified instruction set architectures that would be appropriate for non-CS students [Englander 2000].

To support teaching of computer architecture to students without computational background, we have developed a friendly Web-based resource called *OAMulator*. The OAMulator takes the mystery out of CPU architecture by letting students gain confidence with the concepts of compilers and binary execution. The Web-based deployment allows students to work on problems in convenient locations, at their own pace, and with rewarding interaction. The OAMulator includes an emulator for a simple CPU architecture, the One Address Machine (OAM), and a compiler for a simple high-level language, the One Address Machine Programming Language (OAMPL).

The OAM emulator resource supports the teaching of the following concepts:

(1) Von Neumann architecture
(2) registers
(3) ALU and controller
(4) CPU stages of execution
(5) instruction set architecture
(6) assembly languages
(7) memory and addressing

The OAMPL compiler resource supports the teaching of the following concepts:

(1) high-level languages
(2) compilers
(3) I/O, assignment, and control statements
(4) variable reference resolution
(5) expressions and parsing
(6) optimization

In the remainder of this article we give a bit of background on the OAMulator, outline the OAM architecture and instruction set, overview the OAM Programming Language, and describe the OAMulator resource. We conclude by discussing our own experience with the OAMulator in the classroom and some early feedback collected through student evaluations.

2. BACKGROUND

The One Address Machine and the OAM Programming Language were developed in the early 1990's by AMS at Cornell University. They were parts of a suite of simple instruction sets and computer architectures designed to support instruction in an introductory computer science course for nonmajors.

First in the suite came the Stack Machine (SM) and its programming language SMPL. Then came OAM and OAMPL. Last came the Two-Address Machine (TAM) and TAMPL. Emulators and compilers for these machines and languages were written in Scheme.

OAM and OAMPL were used by the authors to support teaching computer hardware and software concepts in the Introduction to Information Systems course, part of the Master program in Management Information Systems at the University of Iowa. This is the first technical course for MIS students with no previous computational background other than an introduction to programming such as CS I. The course is also available to MBA students with emphasis on information technology and to students from a number of other programs, including accounting, industrial engineering, library science, geography, nursing informatics, and education.

3. OAM: THE ONE-ADDRESS MACHINE

The architecture of the One-Address Machine is shown in Figure 1. Three simplifying assumptions are made:

(1) infinite memory
(2) program instructions start at address 1 (PC = 1)
(3) memory mapped I/O (at address 0)

The OAM has five registers: the *accumulator* (ACC), the B register, the *program counter* (PC), the *instruction register* (IR), and the *address register* (AR). The OAM has three stages of execution: *fetch*, *execute*, and *increment*. In the fetch stage, an instruction is loaded from memory (at the address in the PC) into the IR. In the execute stage, the instruction in the IR is decoded and executed. In the increment stage, the PC is incremented by 1. The cycle then repeats until the program orders to halt execution.

Let us briefly overview the OAM instruction set. OAM assembly instructions can access the content of the ACC and, possibly, the content of *one* memory location (hence the name OAM). Whenever access to memory is required for either data or instructions, the memory location has to be loaded into the AR first. Note that comments are allowed in OAM assembly code; anything following a semicolon (including the semicolon) is considered a comment.

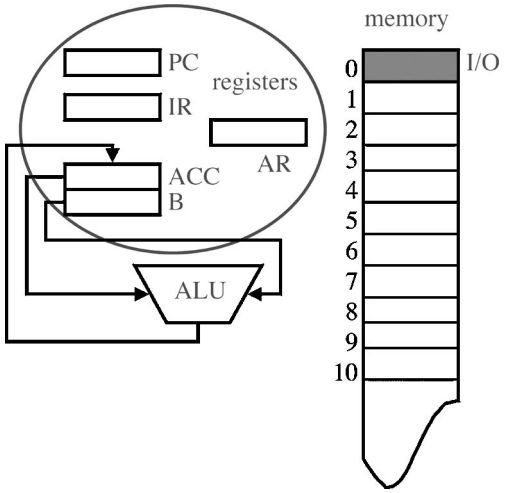

Fig. 1. One-Address Machine architecture.

There are four ALU instructions:

```
ADD address ; ACC := ACC + memory(address)
SUB address ; ACC := ACC - memory(address)
MLT address ; ACC := ACC * memory(address)
DIV address ; ACC := ACC / memory(address)
```

In each ALU instruction, the OAM performs the arithmetic operation indicated by the opcode (addition, subtraction, multiplication, or division) on the two operands: the content of the accumulator and the content of the memory location at the specified address. The latter is first loaded into the B register, since the ALU can only access the ACC and B registers. The result is stored back into the ACC.

There are four instructions that manipulate the content of the accumulator alone:

```
SET value ; ACC := value
NEG       ; ACC := - ACC
INC       ; ACC := ACC + 1
DEC       ; ACC := ACC - 1
```

There are two instructions that move data between memory and the accumulator:

```
LDA address ; ACC := memory(address)
STA address ; memory(address) := ACC
```

The former loads the content of the memory location at the specified address into the accumulator and the latter stores the content of the accumulator

into the memory location at the specified address. I/O in the OAM is memory-mapped, i.e., the special address 0 is used for input and output:

```
LDA 0 ; ACC := input
STA 0 ; output := ACC
```

There are four control instructions that alter the flow of execution from the sequential model:

```
BR  address ; PC := address
BRP address ; PC := address if ACC > 0
BRZ address ; PC := address if ACC = 0
HLT         ; end program
```

The first three control instructions are *branches*, the last one is required to halt execution. There are two conditional branches and one unconditional branch. The unconditional branch simply sets the content of the program counter to the specified address. Since after the execute stage the PC is always incremented (in the increment stage), this means that the next instruction to be fetched will be the one *following* the specified address. The conditional branches do the same thing, but only if the condition is true (i.e., if the content of ACC is greater than zero for `BRP` and if the content of ACC is equal to zero for `BRZ`). If the condition is false, the branch has no effect, and the next instruction to be fetched will be the one following the branch.

Finally, there is a dummy instruction (`NOOP`) that does absolutely nothing.

4. OAMPL: THE OAM PROGRAMMING LANGUAGE

Programs can be written in OAMPL, a simple high-level language for the One-Address Machine. OAMPL statements are naturally more complex than OAM instructions, therefore each OAMPL statement corresponds to (has to be compiled into) many OAM instructions. In OAMPL, as in OAM assembly, anything following a semicolon is a comment. Here is the classic "Hello, world" program example in OAMPL:

```
1. ASSIGN x "Hello, world!\n"
2. WRITE x                    ; could have done it in one step
```

Rather than giving a full formal specification for OAMPL, let us briefly introduce the syntax of its statements and illustrate their semantics by means of a few simple examples. Each OAMPL statement is written on its own line—newline characters separate statements. Key words are case-insensitive and variable names are case-sensitive in OAMPL. There are two I/O statements:

```
READ  variable
WRITE constant | variable | expression
```

Here are a few examples:

```
3. READ A        ; variable can be new or previously defined
4. WRITE 5       ; constant number
```

```
5. WRITE "Foo!"  ; constant strings in quotes
6. WRITE A       ; variable must be previously defined
7. WRITE (+ x 5) ; expression
```

When an OAMPL program is compiled, variable references must be resolved to memory addresses. Variables need not be declared explicitly. A variable is automatically defined if it is given some value, for example via the `READ` statement. In the example above (line 3), the value of the variable *A* is set to whatever the user inputs from the keyboard. The variable *A* might have been previously defined (in which case its value is overwritten), or else it would be defined by the `READ` statement. The `WRITE` statement must have a defined operand, whose value is printed to the screen. In the examples at lines 4 and 5 above the operands are constants, which are trivially defined. Note that strings must be placed between quotes. In the example at line 6 above the operand is a previously defined variable, so its value can be printed. In the example at line 7 above the operand is an expression, and its value is not defined unless the variable *x* was previously defined as a number. Trying to evaluate an undefined variable or expression results in a syntax error at compile time.

OAMPL expressions use prefix notation. The formal syntax of an expression is

```
expression := (operator arg arg) | (- arg)
operator   := + | - | * | /
arg        := constant | variable | expr2
expr2      := (operator arg2 arg2) | (- arg2)
arg2       := constant | variable
```

where the operators `+, *, /` take two arguments, while the operator `-` can take either two arguments (subtraction) or a single argument (negation). Arguments can be constants, variables, or other expressions, but the nesting of expressions is limited to one level. So, for example, `(+ a (/ b 2))` and `(* (- x1 x2) (- x1 x2))` are legal expressions, but `(- 1 (* 3 (- c)))` would result in a syntax error.

The assignment statement is the most common OAMPL statement:

```
ASSIGN  variable  constant | variable | expression
```

As with `READ`, the `ASSIGN` statement should handle both the situation where the first (target) argument is a brand new variable, and the situation where the first argument is a preexisting variable. In the former case the variable is defined by the assignment.

Finally, OAMPL has statements supporting two control constructs that can alter the flow of execution of a program: the conditional block and the loop. A conditional block is defined as follows:

```
IF  constant | variable | expression
...                                  ; block
ENDIF
```

The block is executed only if the condition following the `IF` is true, i.e., if it evaluates to a number different from zero. Note that `IF` statements can be nested. Here is an example that would print "Bar!":

```
8.  IF 3         ; true since 3 != 0
9.  WRITE "Bar!" ; block
10. ENDIF        ; continue
```

Loops support the notion of iteration. They are defined as follows:

```
LOOP  constant | variable | expression
...                                    ; block
END
```

The body of the loop is executed the specified number of times. Note that this might be zero times, and that loops can be nested as well. Here is an example that would print "Foo!" ten times:

```
11. LOOP 10        ; repeat 10 times
12. WRITE "Foo!\n" ; loop body
13. END            ; go back unless done
```

5. OAMULATOR TUTORIAL

The OAMulator resource is publicly available at the URL `http://dollar.biz.uiowa.edu/~fil/OAM/`. It is implemented in Perl for portability, and currently runs as a CGI script on a PowerPC G3 server with the Darwin OS and the Apache HTTP server.

The OAMulator's user interface is shown in Figure 2. The user can type an OAMPL program in the OAMPL Source Code pane and click on the Compile button. An Example button can help get the user started by generating a simple program in the OAMPL source pane. The compiled OAM assembly program will appear in the OAM Assembly Code pane. Alternatively, the user can type an assembly program directly into the OAM assembly pane.

When the user clicks on the Execute button, the assembly code in the OAM assembly pane is executed by the OAM emulator. If any input is required, it is read from the Input pane, which simulates standard input (the keyboard). If any output is produced, it appears in the Output pane, which simulates standard output (the screen) as well as an error console.

There are two optional features for the OAM emulator. The user can elect to see a trace of the OAM registers' content during execution. The state of each register is printed after each fetch/execute/increment stage of the One-Address Machine, based on the user's selected preference. Another option is to see the memory contents at the end of the program's execution. Both the register trace and the memory state appear in the output pane following any program output.

5.1 OAMPL Compiler

The OAMulator's OAMPL compiler allows us to illustrate software concepts. Students can write simple programs in OAMPL and see how they are compiled

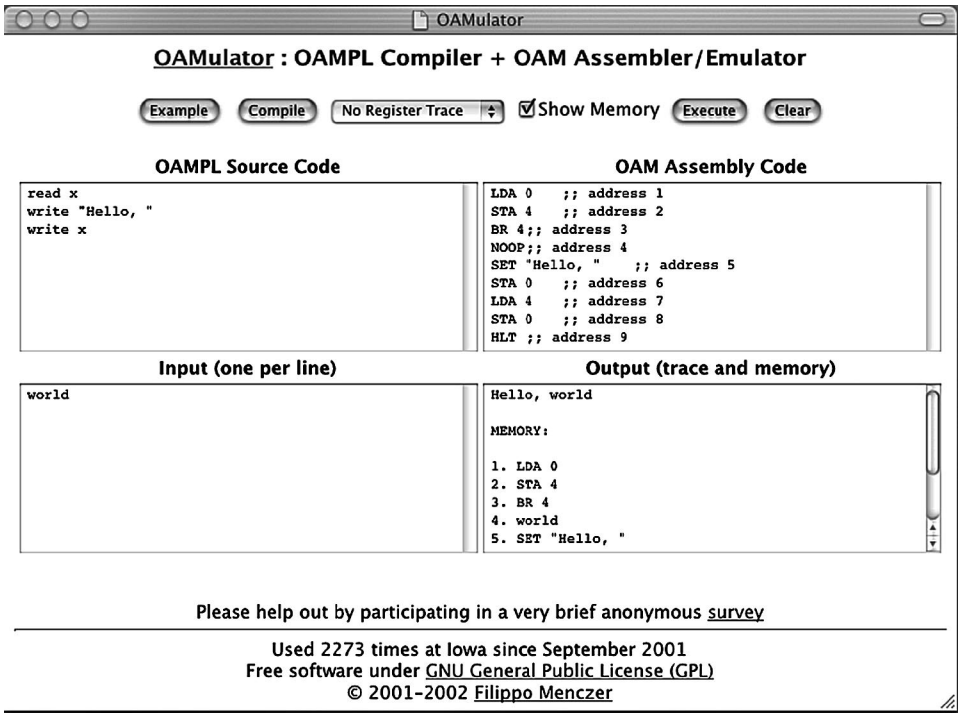

Fig. 2. OAMulator user interface.

into OAM assembly, so that the programming and compilation concepts taught in the classroom can be grounded and applied through examples and assignments.

Students are taught that the job of a compiler is to translate high-level language instructions into a corresponding set of assembly or machine instructions without compromising the integrity of the algorithm that the original program implements. This process becomes evident by analyzing the OAM assembly code generated by the OAMPL compiler.

The compiler catches syntax errors in OAMPL programs, and this allows us to illustrate the concepts of parsing and lexical analysis.

The concept of variable reference resolution is explained by inspecting how the compiler translates I/O and assignment statements. Since OAM only understands memory locations, we need a map from variable names to memory locations. Students are taught that compilers generally do a first pass through the source code to figure out all variables, so that variable locations will not clash with the loaded program instructions. The OAMPL compiler illustrates a less sophisticated approach: interspersing variables with code. This allows us to compile in a single pass, which is less complex for students to grasp. The idea is to skip a location whenever a variable is first encountered. So the first time a statement like `READ A` is encountered, say as the first statement of the program, the compiler will generate the following code:

```
1. LDA 0 ; Read A into ACC
2. STA 4 ; Place A into location 4
3. BR 4  ; Skip memory location storing A
4. NOOP  ; Place holder for A
```

where the NOOP will be overwritten by the value of A, and will never be executed. Successive occurrences of the same statement will reuse the existing location for A, so that the compiler would generate:

```
12. LDA 0 ; Read A again (reuse location!)
13. STA 4 ; Place A into location 4
```

Learning how OAMPL expressions are parsed by the compiler provides further insight into lexical analysis. The strategy implemented by the OAMPL compiler for parsing expressions is as follows:

(1) Evaluate the second operand
(2) Store this intermediate result in a temporary location
(3) Evaluate the first operand
(4) Perform the specified operation between the accumulator and the intermediate result

Consider, for example, the statement:

```
WRITE (* 3 (- C))
```

According to the above strategy, the compiler would use the following template to generate OAM assembly code for such a statement:

```
    ...      ; Instructions to place value of second
    ...      ; operand for MLT instruction in ACC...
n   STA n+2  ; Place intermediate result in n+2
n+1 BR  n+2  ; Skip intermediate result
n+2 NOOP     ; Intermediate result place holder
    ...      ; Instructions to place value of first
    ...      ; operand for MLT instruction in ACC...
m   MLT n+2  ; Multiply by intermediate result
m+1 STA 0    ; Write result to output
```

So if we start at location 11 and assume C is stored in location 7, the OAM assembly code generated would be:

```
11. LDA 7    ; Load value of C into ACC
12. NEG      ; Negate
13. STA 15   ; Store intermediate result
14. BR 15    ; Skip intermediate result
15. NOOP     ; Place holder
16. SET 3    ; Load a 3 into ACC
17. MLT 15   ; Multiply by intermediate result
18. STA 0    ; Write ACC to screen
```

The proper translation of OAMPL control statements provides students with an opportunity to appreciate the subtleties of writing robust code and the complexities of high-level programming constructs. For instance, in order to translate an `IF` statement, the compiler needs to look-ahead, as in the following template:

```
    ...      ; Instructions to place condition in ACC...
n   BRZ m-1  ; Skip block if condition is false
    ...      ; Instructions for conditional block...
m   ...      ; First instruction following block
```

Additional complications for the compiler come from the observation that conditional blocks and loops can be nested. Moreover, loops may need to be executed zero times. Considering these constraints, loops are translated according to the following template[1]:

```
     ...     ; Instructions to place loop count in ACC...
n    BR  m+1 ; Jump to check loop count first
n+1  NOOP    ; Place holder for loop count
n+2  STA n+1 ; Store loop count
     ...     ; Instructions for loop body...
m    LDA n+1 ; Load loop count in ACC
m+1  DEC     ; Decrease loop count
m+2  BRP n+1 ; Go through another loop
```

Consider the example in Section 4 (lines 11–13); if we started at location 18, the OAMPL compiler would generate the following OAM assembly code:

```
18. SET 10       ; Load 10 into ACC
19. BR 25        ; Make sure it's not zero
20. NOOP         ; Place holder
21. STA 20       ; Store updated loop count
22. SET "Foo!\n" ; Load string into ACC
23. STA 0        ; Write to screen
24. LDA 20       ; Load loop count
25. DEC          ; Decrease loop count
26. BRP 20       ; Jump back
```

Finally, the compiler clearly illustrates the value of optimization. Students are taught in class that many (infinite) different, equally valid translations exist for the same program. All things being equal, we would prefer a shorter translation, since fewer OAM instructions that accomplish the same thing imply the program will execute faster. Since the OAMPL compiler in the OAMulator does not optimize, students can see for themselves how inefficient a compiler can be. Consider, for example, the following OAMPL program:

```
READ A
WRITE A
```

[1]Note that although look-ahead is necessary for conditional statements and loops, the compiler still works in a single pass.

It is obvious to students that this can be compiled into the following OAM assembly code:

```
1. LDA 0
2. STA 0
3. HLT
```

Instead, the OAMulator's compiler produces the following code:

```
1. LDA 0
2. STA 4
3. BR 4
4. NOOP
5. LDA 4
6. STA 0
7. HLT
```

This observation allows students to appreciate the efficiency that can be gained through optimization. Furthermore, upon discussing such an example, students can be asked what is necessary to implement this optimization. It can be explained that the variable *A* is not used elsewhere and that its integrity is not compromised by other operations. Hence, students understand that optimizers must make multiple passes through the code to check for many such conditions and transform the code to evenmore efficient forms, without altering the semantics. They also note that some optimizations are machine dependent. Ultimately, students learn how the complexity of a compiler increases with the sophistication of the programming language, with the size of the instruction set and with the desired efficiency of the output code.

5.2 OAM Assembler/Emulator

With the assembler/emulator component of the OAMulator, students can execute OAM assembly code and observe how input, memory, and registers are manipulated to produce output. The content of the registers during execution and the final memory state can be visualized to aid the students in debugging programs and understanding the mechanics of the Von Neumann architecture.

The assembler/emulator catches syntax errors in OAM assembly code, and also detects runtime errors such as missing `HLT`, divisions by zero, illegal addresses, and missing input. In order to detect endless loops, there is a 10-second timeout on (real) CPU time during execution. If the program has not terminated by then, it is assumed that there is some problem; the emulator process stops and the user is given an error message.

Students can practice implementing and testing simple programs directly in assembly, or testing programs written in OAMPL by executing the code produced by the compiler. During this process they observe the OAM at work and apply what they have studied about addressing, ALU, controller, stages of execution, and instruction set architecture.

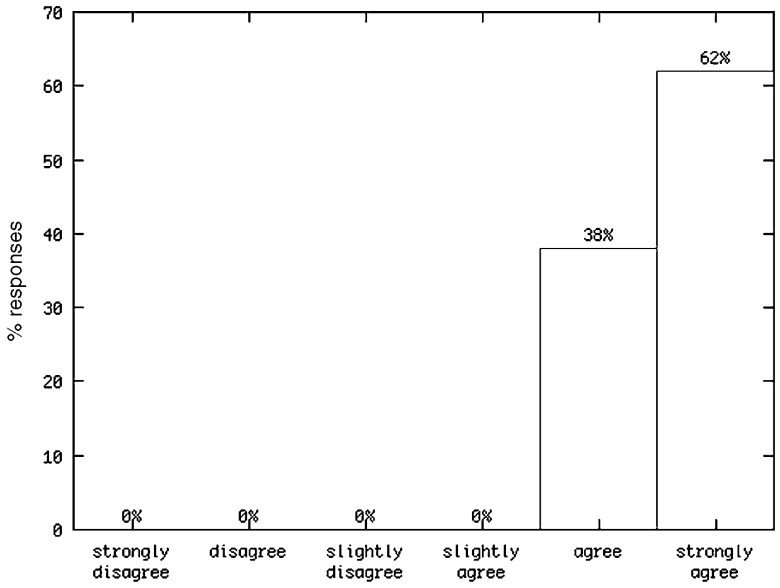

Fig. 3. Histogram of responses to the statement "OAMulator is a helpful resource" among students participating in the OAMulator anonymous survey.

6. CONCLUSION

In this article we described the OAMulator, a Web-based teaching resource that allows students without technical background in computer science to study computer architecture concepts in a convenient and interactive fashion. The OAMulator emulates a simple virtual CPU. Students can write and execute assembly code for this instruction set architecture. They can also write programs in a special high-level programming language and see how these programs are compiled into assembly. The process allows the students to understand the crucial relationship between hardware and software, which would otherwise remain an abstract concept.

We are currently using the OAMulator in a master-level Introduction to Information Systems course, which is part of the Management Information Systems program at the University of Iowa. The class includes MIS, MBA, accounting, library science, engineering, and nursing students. We have gathered some feedback on the usefulness of the OAMulator through an anonymous survey. Students have described the OAMulator as "fun" and as "an extremely helpful tool" to "better understand the assembly language," "think analytically," and "understand the way computers processes instructions." As one student put it, "[the OAMulator] will help me to work through some of my own programs to make sure I understand what happens as the program is executed." No negative comments were made.

We also asked students to provide us with quantitative ratings on the helpfulness of the OAMulator. A sample of 13 students responded, as shown in Figure 3. Although the data is too limited to draw any strong conclusions yet, the OAMulator seems to be an effective teaching resource.

In the future we would like to address two current limitations of the OAMulator resource. First, the implementation does not check for the type of data stored in memory. Each memory cell can hold an arbitrary amount of data of an arbitrary type, i.e., strings, integers, floating point numbers, etc. It is left to the programmer to ensure that variables store the appropriate data types. The compiler could be extended to perform some simple checks and detect errors caused by inappropriate data types, for example to avoid unexpected arithmetic that may occur when mixing strings and integers.

Another limitation of the OAMulator stems from its Web-based deployment. The stateless nature of the HTTP protocol makes it difficult to let users step through the execution of their programs, as allowed by most emulators. A stepwise execution feature would be a nice addition to the resource, but would require the cumbersome use of session cookies, which in turn would limit the ease of use and deployment of the resource.

ACKNOWLEDGMENTS

We are grateful to three anonymous reviewers for their helpful suggestions on how to improve the OAMulator resource. The MySpiders server hosting OAMulator was made available through a Teaching Improvement Award to FM from the University of Iowa's Council on Teaching.

REFERENCES

DA BORMIDA, G., PONTA, D., AND DONZELLINI, G. 1997. Methodologies and tools for learning digital electronics. *IEEE Trans. Education 40*, 4.

ENGLANDER, I. 2000. *The Architecture of Computer Hardware and Systems Software*, 2nd ed. Wiley, New York, 147–162.

PONTA, D. AND DONZELLINI, G. 1994. Learning electronics with hypermedia and computer tools. In *Proceedings of the International Conference on Computer Aided Learning and Instruction in Science and Engineering* (*CALISCE*), J. Dessalles, Ed. Telecom, Paris.

SCRAGG, G. W. 1993. Marina. Software. http://www.cs.geneseo.edu/~scragg/Software/# Marina.

SCRAGG, G. W. 1996. GAL & GEM. Software. http://www.cs.geneseo.edu/~scragg/Software/# GAL & GEM.

Received November 2001; accepted February 2002