The PIPPIN Machine: Simulations of Language Processing

Introduction

As part of the laboratory materials that support our textbook, *The Analytical Engine: An Introduction to Computer Science Using the Internet* [Decker and Hirshfield 1998], we have developed two simulations* which together are intended to help students make the leap from writing programs in a simple high-level language to understanding how such programs come to be translated and executed on a simple computer. The first program, named "Rosetta" (Figure 1), simulates the compilation of as assignment statement from a typical programming language into a mock assembly language. The second, "PIPPIN" (Figure 2), simulates the fetch-execute cycle on a computer built expressly to process that same assembly language.

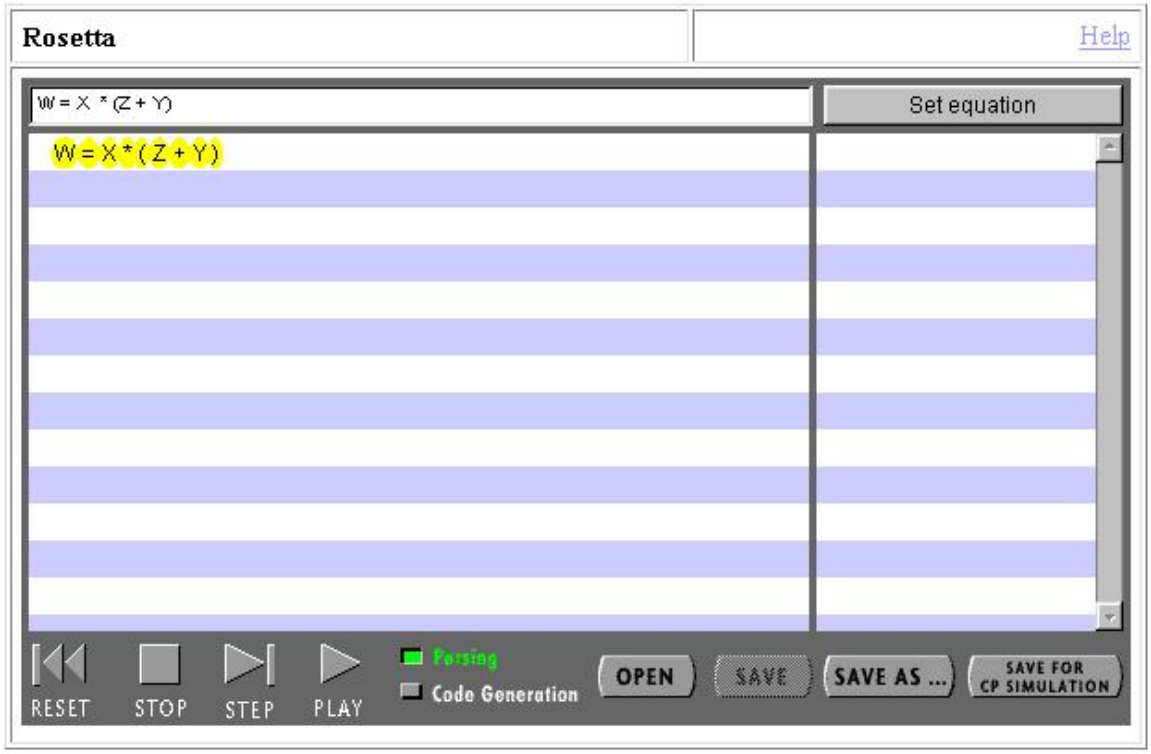

Figure 1: The Rosetta Translator

* We have developed other simulators as well, including a logic breadboard and a Turing Machine simulation. For descriptions of these programs see the aforementioned text, or visit the web site that is the home for all of these simulators, *http://www.brookscole.com/compsci/aeonline/course/index.html*.

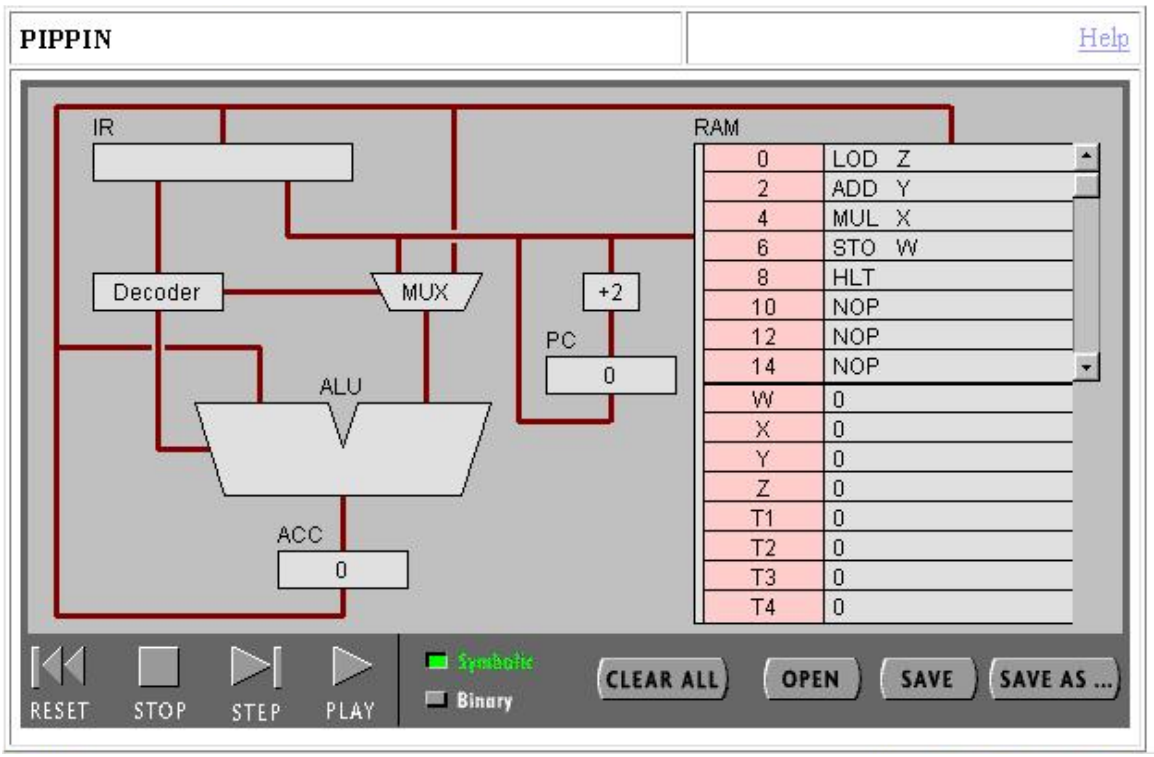

Figure 2: The PIPPIN Simulator

Given the introductory nature and the breadth of coverage of the text, these simulators are necessarily not general-purpose tools. They operate on highly constrained languages and project a similarly high-level view of a computer's architecture and operation. Still, the value of the whole package is greater than the sum of its parts due to the fact that the two simulators are tightly integrated, both in theory and in practice. The assembly language that Rosetta produces when it generates code for a given statement is exactly the language around which the architecture modeled in the PIPPIN simulator was designed. Furthermore, code produced by Rosetta can be saved and loaded directly into the PIPPIN machine simulator for execution with two clicks of a mouse.

We will start by describing the language and the abstract PIP computer, since they are central to both simulators. Next, we outline the design and use of the two simulators, first Rosetta and then the PIPPIN architecture simulation. We conclude with a brief discussion of our experiences using these tools as a package.

PIPPIN: The Computer and the Language

Since we knew that we wanted these simulators to work together, we effectively designed the PIP computer and the PIPPIN assembly language with each other in mind. Our goal in designing both was to keep them as simple as possible while still allowing them to address a representative range of basic architectural issues (for example, the relationship between instruction size and memory size; the relationships between instruction format

and the number of distinct instructions, the use of an accumulator, and available addressing modes). The result was a theoretical machine* with the following attributes:

1. 256 bytes of memory, with addresses numbered 0 - 255
2. memory addressable in 8-bit bytes
3. a single 8-bit accumulator
4. instructions use single-address format, with the accumulator serving to hold the second operand for binary operations
5. instructions represented in 2 bytes, with the right (high-order) byte storing the operand, and the left (low-order) byte storing the instruction code
6. instruction codes are divided into two 4-bit strings, with the right four bits providing the op code, and the left four bits indicating the address mode (binary "0000" for direct mode, and binary "0001" for immediate mode).

This results in the following interpretations of instruction formats:

Machine Language	**Assembly Language**
Direct Mode 0000 CCCC | AAAAAAAA	OPN A *AAAAAAAA interpreted as an unsigned 8-bit address*
Immediate Mode 0001 CCCC | NNNNNNNN	OPN #N *NNNNNNNN interpreted as an signed 8-bit number*
No Operand 0000 CCCC | *unused* ↑ ↑ *Operation Code* *Information*	OPN *last 8 bits ignored* *Where OPN is the PIPPIN instruction corresponding to op code CCCC*

Thus constrained, we defined the PIPPIN language to consist of 14 instructions, as follows.

Binary	PIPPIN code	Meaning
Data Flow instructions		
00000100	LOD X	Load contents of location X into the accumulator (ACC)
00010100	LOD #X	Load value of X into the ACC
00000101	STO Y	Store contents of ACC in memory location Y
Control instructions		
00001100	JMP P	Jump to instruction at location P
00001101	JMZ P	If ACC = 0, Jump to instruction at location P; otherwise, go to next instruction
00001110	NOP	No Operation, but go to next instruction

* We will present the details of how this machine is virtually implemented when we discuss the PIPPIN architecture simulator, below.

00001111	HLT	Halt execution, do nothing more
Arithmetic-logic instructions		
00000000	ADD X	Add contents of location X to ACC
00010000	ADD #X	Add value of X to ACC
00000001	SUB X	Subtract contents of location X from ACC
00010001	SUB #X	Subtract value of X from ACC
00000010	MUL X	Multiply ACC by contents of location X
00010010	MUL #X	Multiply ACC by value of X
00000011	DIV X	Divide ACC by contents of location X
00010011	DIV #X	Divide ACC by value of X
00001000	AND X	If contents of ACC and contents of location X are both 0, put 1 into ACC, otherwise put 0 into ACC
00011000	AND #X	If contents of ACC and value of X are both 0, put 1 into ACC, otherwise put 0 into ACC
00001001	NOT	If ACC contains 0, set ACC to 1, otherwise set to 0
00001010	CPZ X	If X = 0, set ACC to 1, otherwise set ACC to 0
00001011	CPL X	If X < 0, set ACC to 1, otherwise set ACC to 0

Here, then, is a sample program written in PIPPIN shown in both its assembly language and binary forms. The program implements the assignment statement:
X = (3 * Y) + (2 / W).

Assembly Language	Corresponding Binary		
LOD #3	0001	0100	00000011
MUL Y	0000	0010	10000010
STO T1	0000	0101	10000100
LOD #2	0001	0100	00000010
DIV W	0000	0011	10000000
ADD T1	0000	0000	10000100
STO X	0000	0101	10000001
HLT	0000	0111	00000000

Rosetta

The Rosetta program is an animated simulation of language translation. It operates on a single assignment statement, in the format of many common high-level programming languages, and demonstrates, under control of the user, both the parsing and code generation processes. The code that is ultimately generated is PIPPIN assembly language code.

The first step in using Rosetta is to enter the assignment statement to be processed. The statement must conform strictly to the following simple syntax rules: All statements must be of the general form: *variable = expression*, where *variable* is one of W, X, Y, or Z (case-insensitive), and *expression* is a standard algebraic formula composed of variable names (W, X, Y, or Z), integer literals in the range –128 to 127, operators ("+" for addition, "-" for subtraction and negation, "*" for

multiplication, and "/" for division), and parentheses. The order of evaluation for unparenthesized expressions is dictated by operator precedence, with negation having the highest precedence, multiplication and division the next highest level, and addition and subtraction the lowest level.

To enter a statement:

1. click in the text field at the upper-left of the screen (see Figure 3),
2. type the desired statement, and
3. click the "Set equation" button.

The result of correctly entering a statement is shown in Figure 3, where the statement appears highlighted in the scrolling frame below where it was entered (this is the "parse tree" frame). If any of the above syntax rules have been violated in entering the statement, an error message appears, and you are prompted to re-enter the statement.

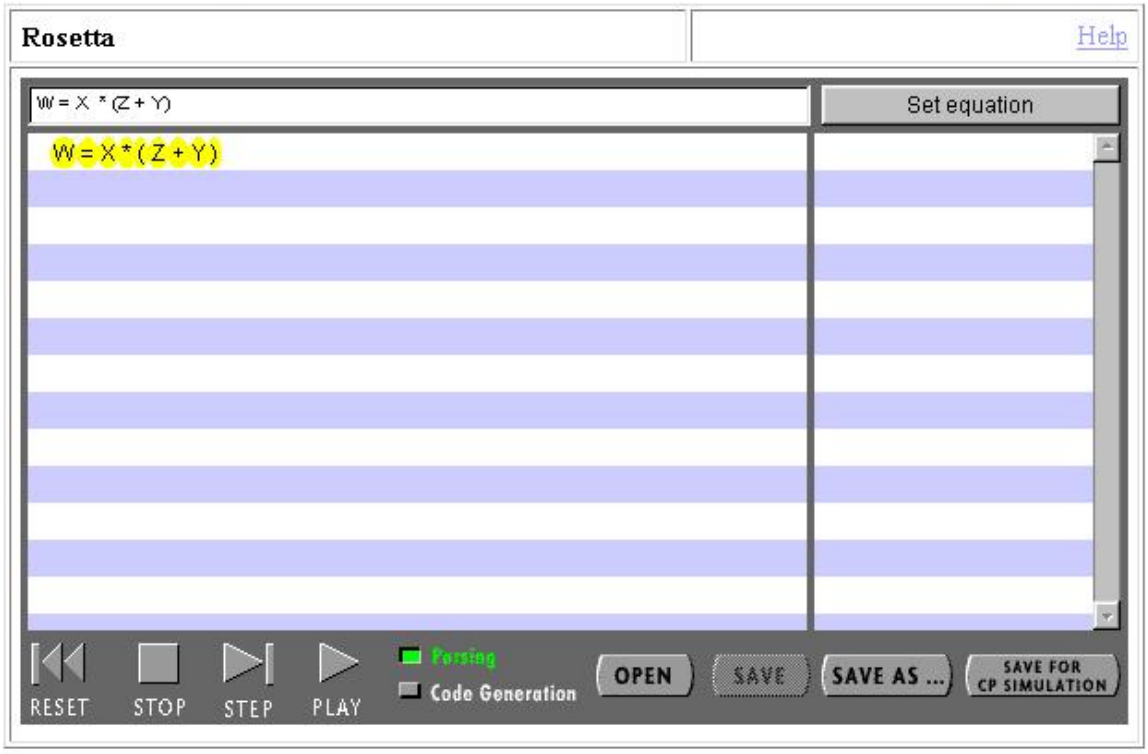

Figure 3: Rosetta with an Equation Entered

Processing of a statement occurs in one of two modes, "Parsing" or "Code Generation", as indicated by the two buttons in the middle of the control panel along the bottom of the display. We can view either the parsing process or the code generation process. The mode in Figure 3 is set to "Parsing."

The user controls both forms of processing by clicking on the four control buttons to the left of the control panel. Clicking on the "RESET" button returns Rosetta to its initial state with the current statement (as in Figure 3). Clicking on the "STOP" button stops the current processing in mid stream (from which processing can be resumed). Clicking on the "STEP" button performs one step of the current processing (i.e., produces

one node of a parse tree in Parsing mode, or generates code for one node of a parse tree in Code Generation mode). Clicking the "PLAY" button processes the statement to completion according to the processing mode. In Parsing mode, processing is complete when a parse tree for the given statement is displayed in the parse tree frame, as shown in Figure 4.

Figure 4: A Parse Tree Generated by Rosetta

The parsing process is animated so that as syntax rules are invoked within the program we see both the corresponding branches of the parse tree expand (in the parse tree frame), and the syntax rule that was invoked (in the scrolling frame to the right of the parse tree frame, the "instruction" frame). Notice in Figure 4 how the syntax rules in the instruction frame appear horizontally aligned with the branches of the parse tree that they correspond to.

To perform code generation on a syntactically legal statement, the statement must be parsed (so that its parse tree is displayed in the parse tree frame), and, the program mode must be set to "Code Generation." These two steps can be performed manually (that is, you can enter the statement, perform the Parsing operation, and the switch to Code Generation mode), or it can be done automatically. If, for example, you want to enter a statement and proceed directly to Code Generation mode, you can do so. In this case, the parse tree is generated without animation or step-wise control and is displayed directly in the parse tree frame.

Controlling the code generation process is done just as is controlling parsing. The process can be reset, stopped, stepped through one transformation at a time, or played through to completion by clicking on the appropriate control button.

Figure 5 shows code generation in process, after one click of the "STEP" button. Notice how the tope-most branch of the parse tree has been collapsed in the animation, and how PIPPIN assembly language code corresponding to that branch, representing the expression `(Z + Y)`, now appears in the instruction frame.

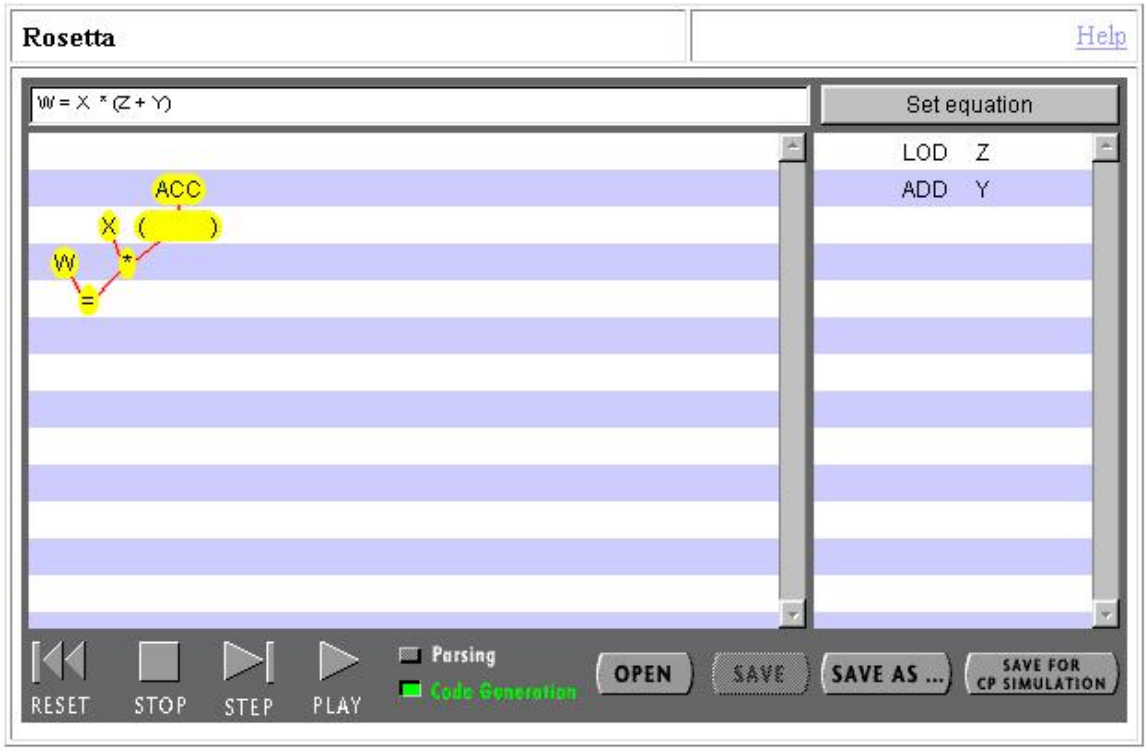

Figure 5: Rosetta in the Process of Code Generation

Figure 6 shows the screen as it appears when the code generation process has been completed. The parse tree has been completely collapsed, and all of the PIPPIN assembly code needed to implement the assignment statement appears in the instruction frame.

We mentioned earlier the fact that Rosetta is integrated with the PIPPIN architecture simulator. This integration is achieved by the "SAVE FOR CP SIMULATION" button at the far right of the control panel. Clicking on this button saves (using a standard file dialog box) the contents of the instruction frame (the PIPPIN assembly code) so that it can be read directly into the PIPPIN simulator.

Figure 6: Rosetta with Code Generation Completed

PIPPIN Architecture Simulator

The PIPPIN architecture simulator allows its user to control the execution of PIPPIN assembly language code on the PIP machine, and animates the process to illustrate which operations are performed, what order these operations are performed in, and how information flows through the machine to accomplish the processing.

The screen for the simulator (Figure 7) is similar to that of Rosetta, with a control panel along the bottom (holding nearly the same collection of control buttons as does Rosetta), and a display above that. In this case, the display shows the essential components and connections that make up our PIP machine.

The collection of fields to the right side of the display describes the machine's RAM. What is visually the "top" half of the RAM is designated to hold the PIPPIN instructions that are to be executed. These memory locations are numbered starting at zero, and since each PIPPIN instruction requires two bytes of storage, increase by 2 up to 110 (enough for programs containing 56 instructions). The "bottom" half of RAM is designated for data storage and, consistent with our language restrictions for Rosetta, contains pre-assigned storage for symbolic variable W, X, Y, and Z beginning at location number 256 (labeled "W"). It also contains pre-assigned storage for temporary variables (named T1, T2, T3, and T4) that might be required to implement certain assignment statements (from Rosetta), or can be used as general-purpose variables by any other PIPPIN program.

The rest of the simulator display is devoted to the remaining components and connections of the PIPPIN computer, as follows:

1. the box labeled “PC” is the 8-bit program counter
2. the box labeled “IR” is the 16-bit instruction register (with the op code showing in the leftmost byte, and the operand in the right byte),
3. a “Decoder” and a mulitplexor, “MUX,” which send control signals to both the RAM and the ALU,
4. the “ALU” is the arithmetic-logic unit which, when being used, displays both its currently selected operation and the data that it is operating on, and
5. the “ACC” is the 8-bit accumulator.

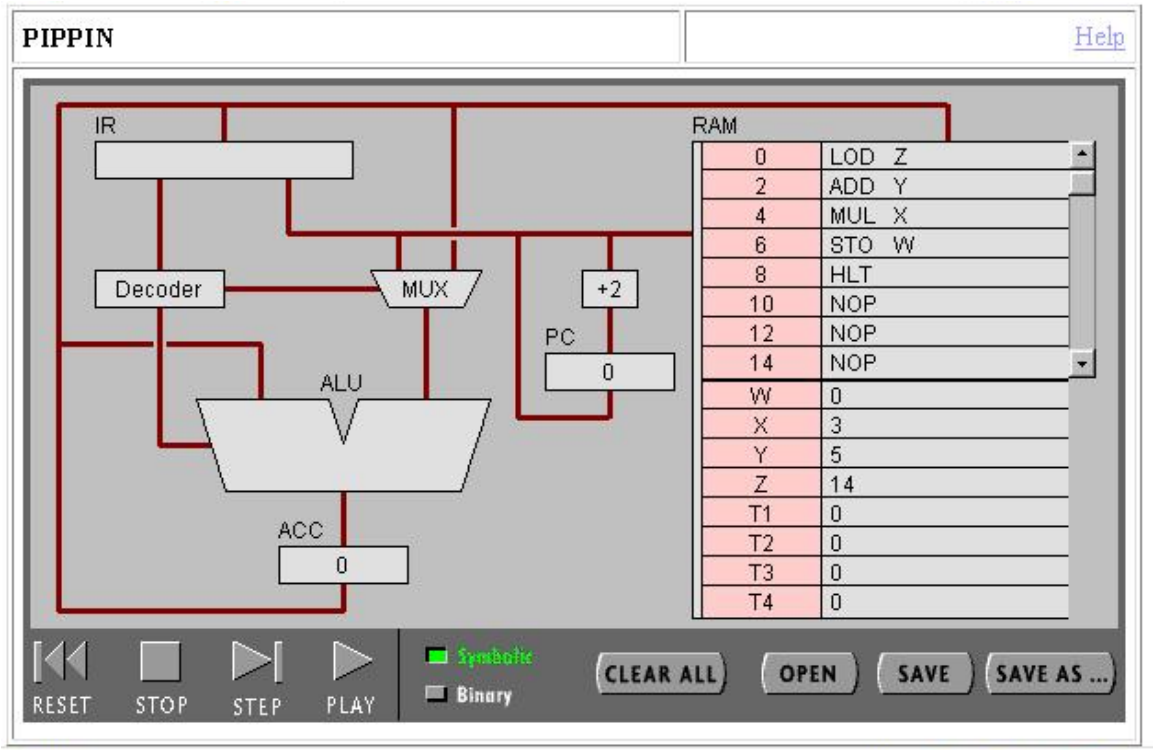

Figure 7: PIPPIN with Instructions and Data Values Loaded

The first step in using the simulator is to load PIPPIN code into the instruction section (beginning at location 0) of the RAM, and to enter values for any symbolic variables referenced in the code (W, X, Y, or Z) into the data section of the RAM.

Instructions and values can be entered directly by clicking in the desired location (the text box next to an address or variable name) and typing. Hitting the “Enter” or “Return” key closes the location and saves the value entered. If the values are illegal in any way (e.g., an instruction is entered that is not a recognized PIPPIN instruction or is improperly formatted, or an illegal data value is entered), and error message is displayed and you are prompted to re-enter a legal value.

Instructions can also be load by opening an existing PIPPIN program, as created by the Rosetta program. Clicking on the “OPEN” button in the PIPPIN simulator’s

control panel presents a standard file dialog box for opening a file which, in this case, will read only the specially formatted files produced by Rosetta. Having loaded a previously saved program, the user must still enter values for any data items referenced by the code, since Rosetta does not save these as part of its output. Figure 7 shows the state of the simulator with PIPPIN code and data values successfully entered.

Notice that, like Rosetta, the PIPPIN simulator operated in two visual modes. We can view the contents of the PIPPIN machine in "Symbolic" form (as in Figure 7), of in "Binary" form, as shown in Figure 8. Clicking on these two buttons in the control panel toggles the visual mode.

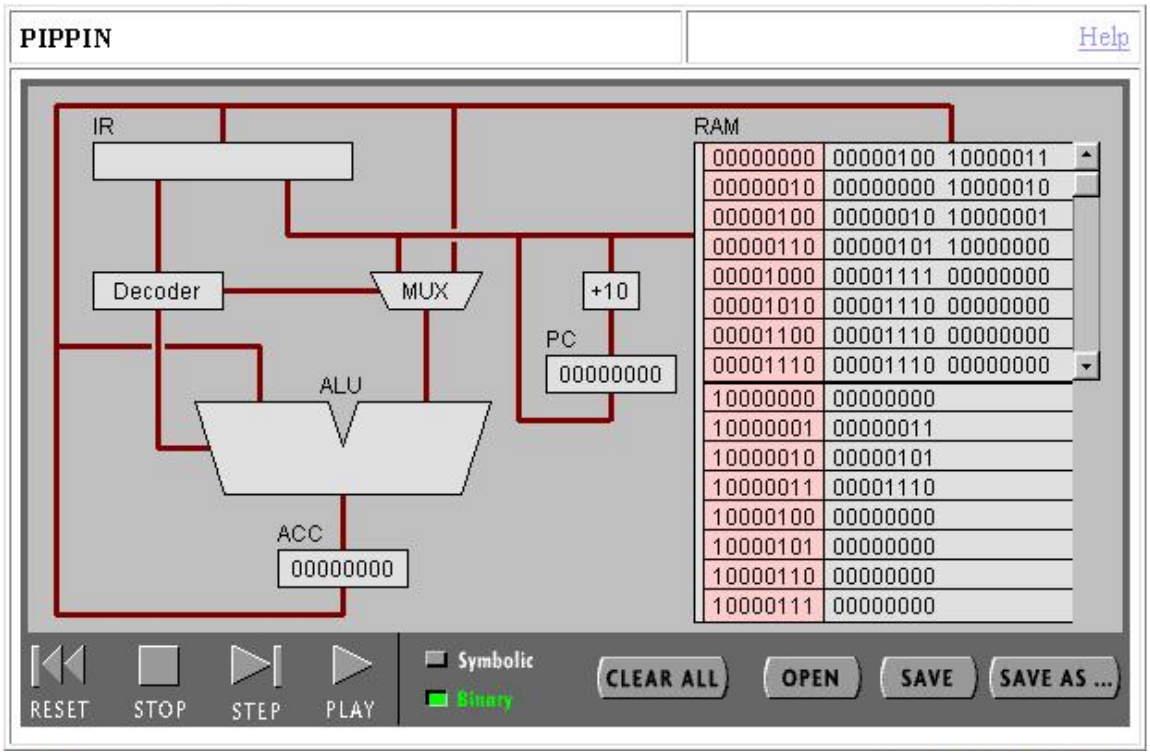

Figure 8: PIPPIN in Binary Mode

Once instructions and data have been properly loaded, running the simulator is simply a matter of clicking on the control buttons to either "PLAY" (run the program until a "HLT" instruction is executed) or "STEP" through the program one instruction at a time.

Figure 9 shows the Symbolic view of the machine immediately after stepping though the first instruction in RAM. You can see that the "LOD Z" instruction has been fetched and stored in the IR, the ACC has been set to 14 (the current value at location Z) which arrived via the ALU, and the PC has been set to 2 (the address of the next instruction to be fetched from RAM).

What is not clear from any of these pictures is that all of the intermediate steps taken to reach this state are fully animated by the simulator. When, for example, an instruction if fetched from RAM, we can see the PC flash, and its connection to the RAM is highlighted. Then, the chosen instruction is highlighted and we can see it "travel" from

RAM to the IR. All data movements and control signals are similarly animated so that the sequence of operations, and the flow of data are perfectly clear.

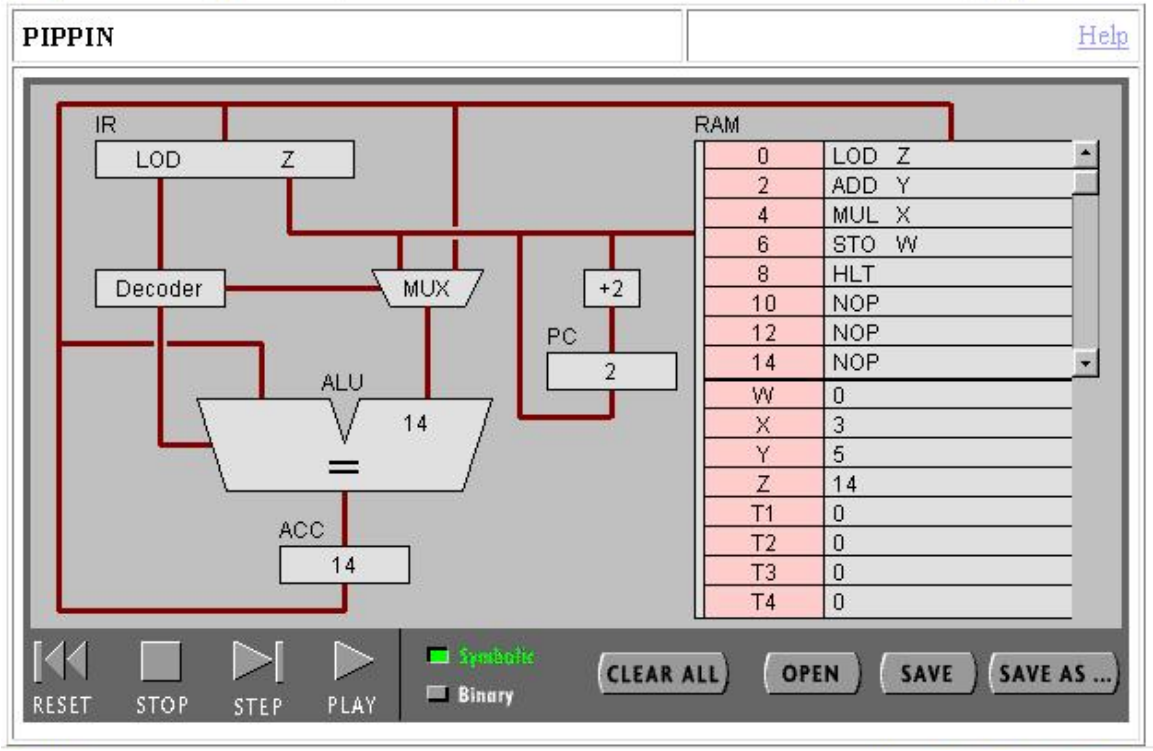

Figure 9: PIPPIN in the Process of Executing Code

Figure 10 shows the results of processing the current program with its original data. You can see that the contents of symbolic location W have changed to 57 (which equals X * (Y + Z)), and the PC has value 10, indicating that the last instruction executed was the HTL at location 8 in RAM.

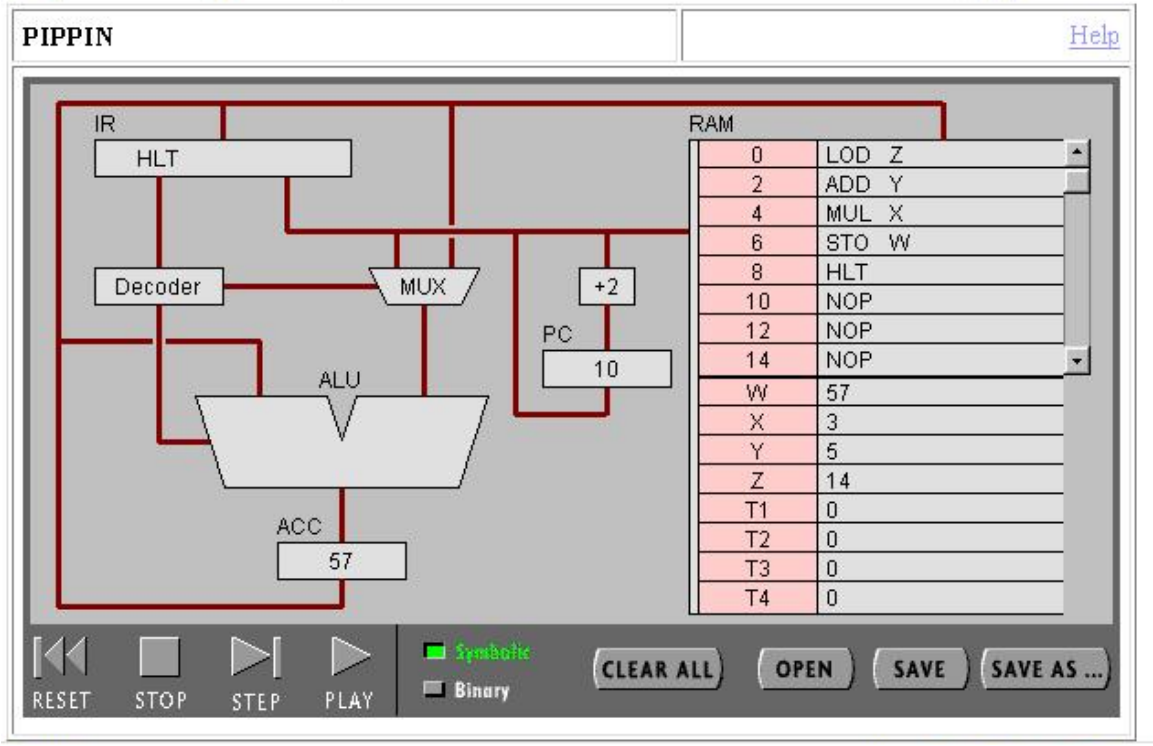

Figure 10: PIPPIN with Code Execution Completed

Conclusions

While both the Rosetta translator and the PIPPIN simulator are seriously constrained in the range of statements and operations that they support, our classroom experiences with CS0-level undergraduates indicate that these constraints can be translated into virtues. Our goal for these simulators was not to develop tools that would show students everything about how computers operate, but rather to show them enough about how computers operate to convince them that the rest is "mere details."

Both simulators accomplish this goal using common techniques, as reflected in their consistent interfaces.

1. The distinction of "program modes" us very helpful for students. In Rosetta, it helps to clarify for them what the phases of program translation are, and how these phases related to one another. In PIPPIN, switching between Symbolic and Binary modes convinces students that high-level code and data can indeed be processed by digital machines.
2. Stepping through both simulators gives students the required control to make sense out of each operation performed. In Rosetta, even true novices step through the translation process in fine enough detail that they become adept at translating between high-level assignment statements and PIPPIN code by hand. Furthermore, they can produce parse trees for these statements almost instantly. Stepping through PIPPIN simulations allow these same students to see, again, in fine enough detail, all that must happen for a single instruction to be processed.

References

[Decker and Hirshfield 1998] DECKER, R. and HIRSHFIELD, S. The Analytical Engine: An Introduction to Computer Science Using the Internet, PWS Publishing, Boston, 1998.