The PIPPIN Machine: Simulations of Language Processing

RICK DECKER
Hamilton College
and
STUART HIRSHFIELD
Hamilton College

This paper describes 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 simulates the compilation of an assignment statement from a typical programming language into a mock assembly language. The second simulates the fetch-execute cycle on a computer built expressly to process that same assembly language. We describe the design and use of each simulator, and conclude with anecdotes about our experiences using these tools in class.

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

Authors' addresses: Both authors are affiliated with the Computer Science Department, Hamilton College, Clinton, NY 13323.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2001 ACM 1073-0516/01/0300-0034 $5.00
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 an 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.

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.

2. PIPPIN: THE COMPUTER AND THE LANGUAGE
Since we knew that we wanted these simulators to work together, we 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\(^1\) with the following attributes:

1. PIP has 256 bytes of memory, with addresses numbered 0 – 255.
2. PIP’s memory is addressable in 8-bit bytes.
3. PIP uses 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 are 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 fields, 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).

\(^1\) We will present the details of how this theoretical machine is implemented when we discuss the PIPPIN architecture simulator, below.
This results in the following interpretations of instruction formats:

<table>
<thead>
<tr>
<th>Machine Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mode</td>
<td>Direct Mode</td>
</tr>
<tr>
<td>0000 CCCC AAAAAAAA</td>
<td>OPN A</td>
</tr>
<tr>
<td></td>
<td>AAAAAAAA interpreted as an unsigned 8-bit address</td>
</tr>
<tr>
<td>Immediate Mode</td>
<td>Immediate Mode</td>
</tr>
<tr>
<td>0001 CCCC NNNNNNN</td>
<td>OPN #N</td>
</tr>
<tr>
<td></td>
<td>NNNNNNNN interpreted as a signed 8-bit number</td>
</tr>
<tr>
<td>No Operand</td>
<td>No Operand</td>
</tr>
<tr>
<td>0000 CCCC unused</td>
<td>OPN</td>
</tr>
<tr>
<td></td>
<td>last 8 bits ignored</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>PIPPIN code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>LOD X</td>
<td>Load contents of location X into the accumulator (ACC)</td>
</tr>
<tr>
<td>10100</td>
<td>LOD #X</td>
<td>Load value of X into the ACC</td>
</tr>
<tr>
<td>101</td>
<td>STO Y</td>
<td>Store contents of ACC in memory location Y</td>
</tr>
<tr>
<td>1100</td>
<td>JMP P</td>
<td>Jump to instruction at location P</td>
</tr>
<tr>
<td>1101</td>
<td>JMZ P</td>
<td>If ACC = 0, Jump to instruction at location P; otherwise, go to next instruction</td>
</tr>
<tr>
<td>1110</td>
<td>NOP</td>
<td>No Operation, but go to next instruction</td>
</tr>
<tr>
<td>1111</td>
<td>HLT</td>
<td>Halt execution, do nothing more</td>
</tr>
</tbody>
</table>

Data Flow instructions

Control instructions

Arithmetic-logic instructions

<table>
<thead>
<tr>
<th>Binary</th>
<th>PIPPIN code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD X</td>
<td>Add contents of location X to ACC</td>
</tr>
<tr>
<td>10000</td>
<td>ADD #X</td>
<td>Add value of X to ACC</td>
</tr>
<tr>
<td>1</td>
<td>SUB X</td>
<td>Subtract contents of location X from ACC</td>
</tr>
<tr>
<td>10001</td>
<td>SUB #X</td>
<td>Subtract value of X from ACC</td>
</tr>
<tr>
<td>10</td>
<td>MUL X</td>
<td>Multiply ACC by contents of location X</td>
</tr>
<tr>
<td>10010</td>
<td>MUL #X</td>
<td>Multiply ACC by value of X</td>
</tr>
<tr>
<td>11</td>
<td>DIV X</td>
<td>Divide ACC by contents of location X</td>
</tr>
</tbody>
</table>
Here, then, is a sample program written in PIPPIN shown in both its assembly language and binary forms. The program implements the assignment statement:

\[ X = (3 \times Y) + \left( \frac{2}{W} \right) \]

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Corresponding Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD #3</td>
<td>1</td>
</tr>
<tr>
<td>MUL Y</td>
<td>10</td>
</tr>
<tr>
<td>STO T1</td>
<td>101</td>
</tr>
<tr>
<td>LOD #2</td>
<td>100</td>
</tr>
<tr>
<td>DIV W</td>
<td>11</td>
</tr>
<tr>
<td>ADD T1</td>
<td>100</td>
</tr>
<tr>
<td>STO X</td>
<td>101</td>
</tr>
<tr>
<td>HLT</td>
<td>111</td>
</tr>
</tbody>
</table>

### 3. 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: \( \text{variable} = \text{expression} \), where \( \text{variable} \) is one of \( W, X, Y, \) or \( Z \) (case-insensitive), and \( \text{expression} \) 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, \( \times\) 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 the user is prompted to re-enter the statement. 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. The user can view either the parsing process or the code generation process. The mode in Figure 3 is set to “Parsing.”

![Fig. 3. Rosetta with an Equation Entered](image)

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.

![Rosetta interface with a parse tree and syntax rules](image)

**Fig. 4. A Parse Tree Generated by Rosetta**

The parsing process is animated so that as syntax rules are invoked within the program the user sees 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, the user wants to enter a statement and proceed directly to Code Generation mode, one 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 top-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.

Fig. 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 PIPPIN” 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.

4. 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 variables W, X, Y, and Z beginning at location number 128 (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.\(^2\)

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 opcode showing in the leftmost byte, and the operand in the right byte),
3. A “Decoder” and a multiplexor, “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.

---

\(^2\) The decisions to begin all programs at memory location zero, and to begin data storage at location 128 were made solely for the sake of simplicity. Indeed, given the limited sizes of “programs” and the limited number of accessible data locations, locations 112-127 and 142-255 of our theoretical machine are effectively useless.
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), an error message is displayed and the user is prompted to re-enter a legal value.

Instructions can also be loaded 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 operates in two visual modes. Users can view the contents of the PIPPIN machine in “Symbolic” form (as in Figure 7), or in “Binary” form, as shown in Figure 8. Clicking on these two buttons in the control panel toggles the visual 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 through the first instruction in RAM. 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 is fetched from RAM, the user can see the PC flash, and its connection to the RAM is highlighted. Then, the chosen instruction is highlighted and the user sees 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 10 shows the results of processing the current program with its original data. The contents of symbolic location W have changed to 57 (which equals X \times (Y + Z)), and
the PC has value 10, indicating that the last instruction executed was the HLT at location 8 in RAM.

5. USING THE SIMULATORS

These programs were both written as Java applets. As such, they can be run from within any Java-enabled browser. They are included in, and can be referenced at: www.brookscole.com/compsci/aonline/course/6/index.html. Everything needed to run them locally is included in a single zip file, named “PIPPINMACHINE.zip” on the JERIC web site. Unzipping this file produces (in addition to folders containing the two applets) a single HTML file named “PIPPIN.HTML”. Open that file with a browser to see a cover page that has links to each of the simulators.

Clicking on either of the links opens the corresponding applet in a new browser window. Because the applets were originally written to be run off of a web server (and because of the inherent I-O restrictions imposed upon Java applets), the user may be prompted to grant (or accept) a security certificate allowing the applet to run. Clicking on “Grant”, “Yes”, or the equivalent will start the applet running.

6. PEDAGOGICAL ISSUES

3 We know that these applets run without problems in Internet Explorer 5.0 and Netscape Navigator 4.7.
Courses in computer organization and architecture have long been recognized as central to every standard computer science curriculum. The topics covered in these courses lend themselves to – even beg for – simulation and visualization techniques for many reasons (most notably because these techniques provide students with a cost-effective and flexible means for seeing directly in a CPU). Simulators are currently available for a wide variety of computers, real and theoretical, varying across a number of dimensions, including the intended audience, intended purpose, ease of use, and degree of interactivity.

Looking, for now, just at our PIPPIN simulator, it is quite representative of what Yurick, Woffle, and Holliday [2001] describe as a “novice” simulator. It provides a straightforward, easy-to-use, easy-to-control view of the core components of a simple machine, along with an equally simple assembly language. That said, it combines features found in other simulators and texts, and extends others to produce a unique visual environment.

Like the “Little Man Computer” described in Yurick, Vila, and Brumbaugh [2001], it provides a simplified view of a CPU, but does so according to a slightly more traditional von Neumann interpretation. Like the “CPU Simulator” by SPA Corp. [2000], it extends a basic instruction-level simulation to include animations that force students to at least acknowledge communication-level issues. When combined with the Rosetta simulator, it bridges the gap between high-level programming languages and low-level machine language, much as (and even more directly than does) the simulator produced by the Compiler/Architecture Simulation for Learning and Experience (CASLE) project of Dietz and Adams [1996]. Finally, like the works of Pate and Patel [2001], Beck [1998], and Tanenbaum [1999] the PIPPIN simulator implements a theoretical machine that is tied directly to a text book.

What distinguishes our simulators from others that we are aware of, and what positions them most accurately within the broader context of teaching computer architecture, is the fact that the text to which they are tied is not a book about computer architecture, per se. Rather, it is a text book that offers a survey of computer science aimed specifically at non-majors. Our expressed purposes are to expose novices to the range of issues and activities that comprises the discipline, and to show how those issues and activities relate to one another – and to the rest of the human race who are not regularly involved and occupied with such things.

This explains not only our simplified approaches to the topics of language translation and computer architecture, but also why we chose to present the Rosetta and PIPPIN simulators as a package in this paper. We treat these topics as one piece – a critically important piece, to be sure – of what for non-majors is a much larger puzzle. It is the piece of the puzzle that bridges the gap between hardware and software. This gap can be
vast, even for computer science students. Many non-majors simply regard it as magic, which is why highly visual and controllable simulations are invaluable to them.

7. 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. The distinction of “program modes” is very helpful for students. In Rosetta, it helps to clarify for them what the phases of program translation are, and how these phases relate 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.

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.

ACKNOWLEDGMENTS

Many thanks to our colleagues at International Thomson Publishing for supporting and encouraging this work, and to our students over the years who have worked with us to make good use of these programs.

REFERENCES


Received November 2001; accepted February 2002.