A Java-based Computer Simulator and its Applications

John K. Estell
Bluffton College

Abstract

This paper describes a learning philosophy for computer science that is based on having students write a simulation of a computer system, then adding features to the initial simulation that are appropriate for learning concepts being presented in a particular course. In the past the author has successfully utilized this method in teaching concepts related to CPU processor scheduling in an operating systems course[1]; however, as the simulation was written in a structured language using platform-dependent code it was not portable to other platforms nor was it possible to adapt the code to examine different concepts without a major rewrite being made necessary. The development and rapid growth of Java, a platform-independent object-oriented language, has provided a facility with which to better serve students. The fundamental components of a computer system, such as registers and memory, can be modeled as objects. Each component is written up into a Java class containing both data elements and methods for operating upon the data elements. As these classes need only be written once, for subsequent assignments students have only to import the appropriate classes and instantiate the required objects in order to create models of the components tailored to whatever specifications are provided. These components are then used as part of the new simulation. Because of Java's design, code can be ported from one platform to another and both graphics and graphical user interfaces can be incorporated into the simulator.

The Introductory Assignment

The introductory assignment occurs at the end of the first object-oriented programming course, which for our students is the third programming course that is taken. At this point the students have been exposed through both lecture and programming assignments to such topics as objects, GUIs, and events. Now the students are introduced to the problem of computer simulation via a brief explanation of a simple von Neumann architecture. The design features a single 8-bit register (the accumulator) for data processing and I/O, one flag (the zero flag) for conditional testing, an 8-bit register for use as the program counter, and a 256-byte memory store. A very simplistic instruction set containing a minimal amount of features is then provided. This instruction set, shown in Figure 1, allows for simple processing of integer values through the use of load, store, add, branch, input, and output operations. The load and add operations can be performed with either immediate or stored values. Both the input and output operations interact with the user through use of TextField objects. The instruction set also includes an end operation so that the simulation can be halted when execution is complete. At a minimum the students are instructed to implement classes for the memory, register, and flag modules, implement the main

Mnemonic	Op Codes	Description
`LDI`	`01 val`	Load immediate value ‘`val`’ into accumulator
`LDA`	`02 addr`	Load contents of memory address ‘`addr`’ into accumulator
`STA`	`03 addr`	Store accumulator into contents of memory address ‘`addr`’
`ADI`	`04 val`	Add immediate value ‘`val`’ to accumulator
`ADA`	`05 addr`	Add contents of memory address ‘`addr`’ to accumulator
`BEQ`	`06 addr`	Branch on zero flag set to memory address ‘`addr`’
`INA`	`07`	Read integer from input text field into accumulator
`OUT`	`08`	Write integer in accumulator to output text field
`END`	`09`	End program

Figure 1. The simple instruction set.

simulator class that extends the Applet class, and design the graphical user interface (GUI). In-class discussions are held to assist students in determining what methods and data structures are needed for each module and as to what should appear on the GUI.

The Flag class is the simplest to implement; it needs only a boolean instance variable to represent the state of the flag, and both get and set methods for accessing the value of the instance variable. A constructor method should be included so that when a Flag object is instantiated it is placed into a known state. The Memory class requires the use of a constructor method in order to specify the number of bytes of storage for the Memory object being instantiated. An array of integers is used to represent the actual memory. While an array of bytes could be used, this forces the use of the storage range [-128, +127], which is an unneeded complication. For this project it is sufficient to use unsigned representation but it is also desirable to have access to the full range of unsigned 8-bit values. Both the get and set methods for this class must verify that the specified memory address is legitimate; for an illegal memory access the value -1 is returned. Additionally, the set method needs to make sure that the value being stored in memory is a legitimate byte value; this can be easily done via the brute force method of storing the value modulo 256 into memory. The Register class is similar to the Flag class, except now one stores an integer instead of a boolean value. For this module one can present the students with a challenge by asking them to implement the class such that one can specify the number of bits needed for the register. This is beneficial as in a typical processor one often finds registers of various widths. From the width value a mask of all 1’s can be easily constructed and is then used to insure that the register value does not fall out of the allowed range. The traditional set and get methods are included; one can also include increment and decrement methods as well.

For greater flexibility classes can be created both for the instruction set and for event handling. The Instruction class allows for one to play with several instruction sets at minimal cost. As long as no changes are needed with the architecture, the only modification needed is the creation of a new class containing the new instruction set; this can be best implemented through use of an abstract Instruction superclass from which all of the concrete instruction set subclasses are derived. The only methods necessary for operation are the constructor method, where references to the architecture are passed, and the execute method, which will execute the instruction stored in memory at the provided address. Two additional methods are included to provide information to the user. The toString method returns the mnemonic for a given operation code. The disassemble method returns a String object containing the raw assembler instruction based on the machine code stored at a particular memory location. For event handling there are four possible actions. First, one needs to be able to load an executable program into memory. This is implemented through use of a "load program" action event handler that is associated with a "load" button on the GUI. The event handler can load the executable code in one of two ways. The easier implementation is for the programmer to hardcode the executable code in an array contained in the handler; when the handler is invoked the contents of the array are copied into the Memory object. The other implementation is to read in a file whose contents contain the executable code; the name of the file is obtained through a TextField object located either on the GUI or on a modal dialog box. For the operation of the simulation two event handlers are implemented, each of which is associated with its own button on the GUI. The "step" action event handler allows only one instruction at a time to be executed; this allows the user to see what is happening as the program executes. The "run" action event handler allows the program to execute instructions unimpeded until either blocked by a request for input or when the end of the program is encountered. Finally, an action event handler is implemented for the user input TextField object in order to notify the simulation that the requested input has been entered so that the block can be removed. To coordinate the running of the program, execution states are implemented through use of a State class. The states implemented are ready, run, blocked input, input ready, step, end of program, and halt (for when an illegal instruction is encountered).

The simulation operates by having the event handlers make calls to the applet's repaint method. In the applet, the paint method contains the control unit code that determines what to do based on the current state of the machine. Instructions are executed only if the machine is in either the run or step states. As the machine state can change following the execution of an instruction (either by an I/O request or encountering either the end of the program or an illegal instruction) further state processing is performed after an instruction is executed. The init method of the applet is used to define the GUI. At a minimum, the GUI should contain TextField objects that will display the current accumulator, program counter, and zero flag values, have TextField objects for both user input and user output, contain appropriate Label objects for each TextField object, and have appropriately labeled Button objects for the load program, step, and run events. For students with time on their hands, extensions to the GUI could include a TextArea object showing a disassembled listing of the next few instructions to be executed, a scrollable TextArea object displaying the values stored in memory, and a graphical "console panel" display that would show a set of panel lights for each register. For those truly wanting a challenge, toggle switches can be drawn on the GUI, then manipulated through use of mouse events. While definitely in the "bells and whistles" category, it does bring back the flavor of the "good old days" of front panel interaction with the computer while allowing students to become more proficient with their graphics and event handling programming.

Additional Assignments

Once the base classes have been designed, it is a relatively simple task to make alterations to the applet that allow for the exploration of other topics in advanced courses. The simplest example is in the assembly language course. A "full-fledged" assembly language can be easily implemented by instantiating the appropriate Register and Flag objects, then writing an Instruction subclass containing the implementation of the operation codes. As part of the design students can be given the opportunity to better understand the various addressing modes normally found in an assembly language. In the original design they have already implemented both immediate and direct addressing modes; the students can now implement such concepts as relative, indirect, and indexed forms of addressing modes. It has been the author's experience that students do not readily grasp the more "exotic" addressing modes from either lecture or from their readings, but they do acquire a firm understanding of what transpires when they have to actually get a particular addressing mode to work in a simulation.

Computer organization is another area where the use of a computer simulation can be beneficial. In the initial assignment for the simulator the students have already dealt with the concept of flags, registers, and memory, and in order to get the simulator to work a simple control unit was implemented. However, there are other areas that can be explored. For example, with respect to data representation the original design of the simulator used unsigned integers as the only data type available. It takes little effort for the I/O devices to be converted from using integers to using characters. With the addition of a sign flag to the architecture, one can start playing with two's complement integer representation. As Java does have bit-oriented operators, simple floating-point representations can also be implemented where bit masks and logical shifts are used to access pertinent information. Areas of computer architecture can also be explored with the simulator. A Harvard architecture can be explored by the instantiation of two Memory objects: one for storing the instructions, and one for storing the data. A RISC architecture can be implemented by instantiating as many Register objects as deemed necessary, although one would be advised to use a scrollable TextArea object if one desires to display the contents of all of the registers simultaneously. Stacks can be implemented either by reserving a block of addresses within the Memory object used to store program information or by instantiating a separate Memory object that is dedicated for stack use (similar to the way stacks were implemented on early microprocessors such as the 6502). Secondary storage devices such as tapes and disk drives can be created through the construction of their own classes, after which they can be incorporated into the simulator.

Operating systems is another area where simulation can be successfully used. The applet can be modified so that multiple programs can be loaded onto the system, each of which would possess a process control block. When instructed to begin execution, the simulator would use a provided CPU scheduling algorithm to determine which process would be allowed access to the processor. A new Time class is added to keep track of elapsed time; each instruction in the Instruction subclass would now include the number of clock cycles required for completing its task. The applet can perform the various statistical and data analyses for the processes on the fly and can display the results either numerically or graphically as part of the GUI, thereby relieving students of the need to write separate analysis programs. As Java has built-in networking functions, an industrious student could set up a network between different applets and play with a distributed operating system.

In the area of programming languages and compilers, one can design an assembler that will take assembly language source code and produce an executable code file that can then be read in by the simulator. A simple high-level language could then be created by developing a compiler that would generate either an assembly language source code file, which could then be processed by the previously written assembler, or it could directly generate the object code file for the simulator. For both the assembler and the compiler Java makes program development much easier for the student by providing the StringTokenizer class. The StringTokenizer class is used to process an input source code file into tokens, allowing for easy recognition of numbers, words, end of lines, and end of file. The methods for this class will even allow for the automatic skipping over of both the "slash-star" and "slash-slash" styles of comments if desired, making the writing of the parser almost trivial.

Conclusion

One of the best methods for encouraging students to learn the material being presented in a course is to let them "get their hands dirty" through the use of implementation activities. Having to produce something that works forces students into acquiring an understanding about what is going on. However, this should not necessarily mean that one has to reinvent the wheel by starting from scratch for each activity. The adaptation of a student-built computer simulator provides the framework upon which a variety of activities can be constructed. This allows the focus to be placed on implementing the concept being studied instead of having to focus on the implementation of both concept and framework.

The choice of the Java programming language meets many of the goals for providing a reusable framework. The benefits of software reusability and data encapsulation obtained through use of classes in an object-oriented language are well known. Java also provides built-in GUI capabilities, which makes the development of the user interface rather straightforward. Because of the portability of the Java language students – and departments – are free from having to deal with platform constraints. Furthermore, the cost for using Java is minimal as the Java Development Kit (JDK) can be downloaded free of charge from Sun Microsystems' Java web site (http://java.sun.com).

References

1. J. K. Estell, "A Simulation Project for an Operating Systems Course," 1996 ASEE Annual Conference Proceedings, June 1996.

JOHN K. ESTELL joined Bluffton College as an associate professor of computer science in 1996. He was previously an associate professor at The University of Toledo. He received a BS (1984) degree in computer science and engineering from Toledo and received both his MS (1987) and PhD (1991) degrees in computer science from the University of Illinois at Urbana-Champaign. His areas of interest include programming development tools and interface design. Dr. Estell is a member of ACM, ASEE, IEEE, Tau Beta Pi, and Eta Kappa Nu.