SicSim: A Simulator of the Educational SIC/XE Computer for a System-Software Course

J. MIHELIČ, T. DOBRAVEC

Faculty of Computer and Information Science, University of Ljubljana, Ljubljana, Slovenia

Received 3 June 2013; accepted 21 September 2013

ABSTRACT: A modern computer system provides its support via system software that consists of applications such as an assembler, a linker, a loader and virtual machines. It is of prime importance to give students that are learning system-software concepts a solid base of knowledge without any unnecessary details. To make the subject easy to understand we designed a simulator for a hypothetical computer that is already used in several courses on system software. In the paper, we describe the simulator's behavior as well as its design and implementation. Additionally, we present three case studies of using a simulator in teaching and describe our experience of its use in a course on system software. From the experience of using the simulator in a pedagogical process we conclude that it decreases the time invested by the students to comprehend the topic, and at the same time it enables in depth understanding. © 2013 Wiley Periodicals, Inc. Comput Appl Eng Educ 23:137–146, 2015; View this article online at wileyonlinelibrary.com/journal/cae; DOI 10.1002/cae.21585

Keywords: system software; virtual machine; simulator; computer architecture; educational software

INTRODUCTION

Many computer science courses cover topics such as a run-time system, assembling, linking, loading, and virtualization. The understanding of these concepts is of great importance in the process of designing and implementing efficient system software. There are many factors, including the computer architecture, operating system, and execution environment, which influence the process; however, the core concepts stay the same.

Since the real-life circumstances, particularly computer organization and architecture, operating systems, system software are usually complex, a comprehensive explanation of the above concepts is often too cumbersome for educational purposes. Additionally, technical details such as instruction formats, addressing modes, system calls, the format of executable files, software debugging, etc., are usually not directly related to the presentation; instead they may blur the whole picture. Providing a real environment in the educational process is also often impossible or hard to implement due to financial, or any other difficulties.

In such cases, educators often turn to simulation. Although the simulation of a computer or its part may not reflect the real environment as a whole, it is extremely useful pedagogical tool, which improves the learning process. In addition, in certain cases, the use of simulators enables educators and students to perform operations that are impossible in the real-world environment (e.g., monitoring events in a small fraction of a time, or parametric change of hardware components).

For the above reasons, many hypothetical computers have been designed [1–5] and are mainly used for presentation purposes during various courses. In this paper, we focus on a hypothetical computer, called the *Simplified Instructional Computer* [1], which comes in two versions: the standard model (`SIC`) and the extra-equipped model (`SIC/XE`). The latter is an extension of the former with a larger address space, a larger number of registers, and additional instructions. The two versions are upward compatible and designed especially for teaching the course on system software. Due to their careful design and overall usability, the `SIC` and `SIC/XE` hypothetical computers are often adopted in many courses all over the world as a demonstration platform for the basic concepts of system software.

The main drawback of using a hypothetical computer is that programs cannot really be run. To overcome this, such a program can be run on a dedicated simulator. Various simulators are often used in practice for educational purposes [6–8]. Several `SIC/XE` simulators and assemblers were already designed [9–12], but to our knowledge none of them provides a sophisticated executing and debugging graphical user interface (GUI). To fill this gap, we have designed and implemented a GUI-based application, denoted by SicSim, that simulates the `SIC/XE` computer. The simulator

Correspondence to J. Mihelič (jurij.mihelic@fri.uni-lj.si)

can load and execute the SIC/XE machine code. Additionally, the simulator supports debugging with breakpoints, stepping through machine code, and several other extensions.

We use the simulator in our course on system software. The layered model of the computer system places the system software between the operating system and the user applications. The course is mainly based on [1] with some additional topics. The syllabus comprises of run-time systems, process virtual machines, assembling, macro processing, static and dynamic linking, and loading.

Since the course is given in a 3rd year of study at the faculty of computer science, participating students already have strong background in computing. In particular, they are acquainted with programming, basic algorithms, and computer organization, which are all important for the course.

The purpose of the simulator within the course is twofold. First, it enables students to understand system software concepts (e.g., virtualization, assembling, loading). Second, it serves as an example application, for one of the students' tasks in the course is to make a similar simulator on their own.

In this paper, we present our application SicSim from both the user and the designer perspective. In the next section, we describe all the functional parts of the application from the user point of view. The third section gives a detailed description of the simulator's design and implementation. In the fourth section, we present three case studies that is the loading mechanism, the virtual screen, and the educational experiences. In the last section, we conclude the paper.

RELATED WORK

In a pedagogical process, the use of computer simulators, through which expensive computer hardware is replaced with virtual environment, provides a switch from a bare theory to practice. Therefore, it is not surprising that many authors of books and other computer literature often reach out for different simulators.

There are several well-known textbooks using hypothetical educational computers mainly for demonstration and presentation purposes. For example, Knuth's famous multi-volume textbook series on algorithms introduces a simple mythical computer, called MIX, and also its simulator [3]. The MIX assembly language has been designed to be powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned. The input of a MIX simulator is a sequence of MIX instructions and data, stored in dedicated locations. The simulator imitates the precise behavior of MIX's hardware. Since the MIX simulator is written in MIX language, it is used mainly in the educational and the presentational purposes. In the later editions, MIX is superseded by an enhanced version (64-bit RISC), called MMIX.

In the textbook, Hennessy and Patterson [4] introduce the DLX (pronounced "Deluxe") computer architecture, which is used for a number of exercises and programming projects. It emphasizes a simple load-store instruction set, pipelining efficiency, and efficiency as a compiler target.

Another textbook by Tanenbaum and Goodman [5] presents an example microarchitecture, called MIC-1, and also a series of enhancements, the most advanced one called MIC-4. Together they are used to explain the basic principles of microarchitecture design, presenting all details such as data path, timings, micro-instructions, pipeline, etc.

Campbell describes an example of a simulator of a simple hypothetical computer in [2]. This computer consists of a main memory, a register, a CPU, and an instruction counter. Its assembly language recognizes only eight machine instructions and is thus simple enough to be easily understood by students and on the other hand complex enough to cover the main aspects of computer architecture and program execution.

Another simulator by Sayers and Martin [8] provides an implementation of the hypothetical computer found in [5]. It is based on textual user interface and permits development of programs at the conventional machine language programming level. It is being used in a systems programming class.

Beside the computer simulators, there are also simulators of a particular part of computer systems. For example, Moreno et al. [6] present a simulator of a computer-memory hierarchy. This multi-platform simulation tool, developed in Java, enables a huge set of concepts and parameters and has been validated and improved by students.

Another example of a simulator, Castilla et al. [7], used for teaching purposes, imitates instruction-level parallelism architecture. It supports dynamic and static instruction scheduling. It is a great aid-tool for teaching theoretical contents in Computer Architecture and Organization course.

Several simulators of SIC/XE are already available. The original one, by Beck [9], with textual user interface and implemented in Pascal, provides basic features of the SIC/XE specification.

Similarly, Microsoft Visual Studio C++ 2010 project, Sultan et al. [11], supports SIC/XE object code execution. The input of a particular simulation consists of the two files that is SIC/XE object code and input device data. The result of a simulation consists of the final values of registers and a list of memory changes.

Another SIC/XE simulator, Fox [12], provides a set of shell-based system tools such as assembler and virtual machine. To execute a program, user must assemble the source code, initialize the virtual machine, and load the obtained object code. The results of a simulation are provided to the user after the program stops.

All above SIC/XE simulators use a separate program to assemble the source code (which means that a user must execute assembler manually before the simulation). However, their main drawback is that they provide only the final results of the simulation, and all the intermediate results are not available to the user.

THE SicSim FEATURES

As a simulator of a hypothetical SIC/XE computer, SicSim loads and executes machine code and displays the state of the underlying virtual machine. The simulator is designed as a useful tool, enabling a user to easily observe important information about the executing program.

During the simulation, SicSim displays the following information: the registers, the instruction to be executed, and the values of the selected memory locations. The application consists of the three main windows: the "CPU window," the "Memory-dump window," and the "Screen window." Figure 1 shows the application windows during the execution of a bouncing-ball game.

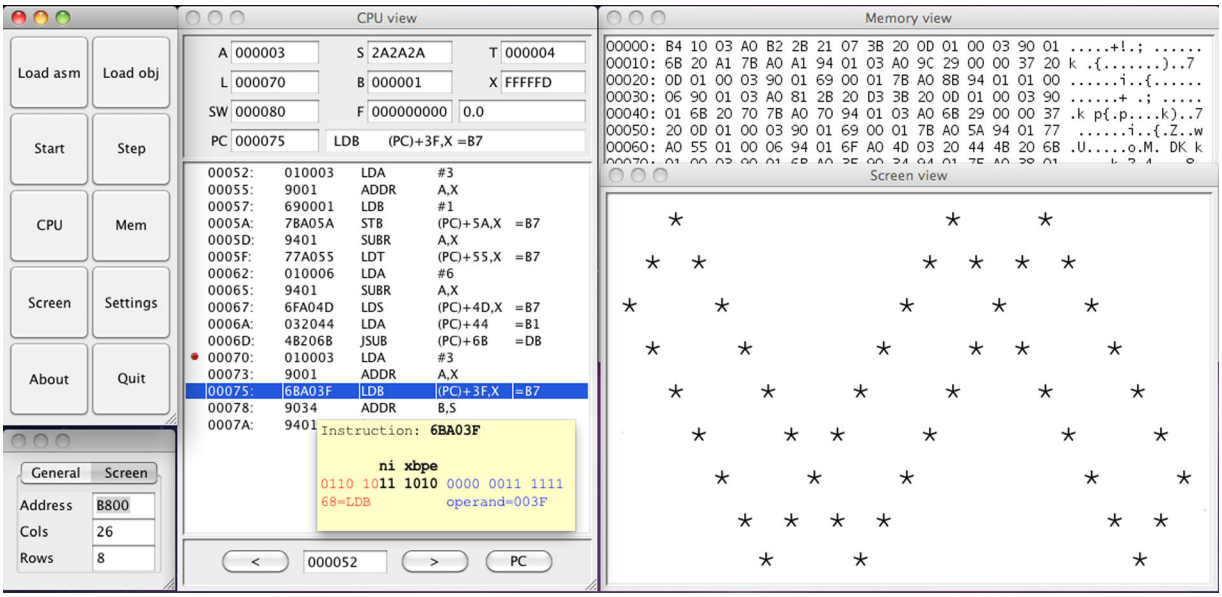

Figure 1 A typical layout of the windows in the SICSIM application. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Control Panel

The Control panel is used to control and access the main windows of the application. It is shown in Figure 2. Using the buttons "Load asm" or "Load obj" the user can load the assembly or object code into the main memory of the simulated machine. When loading the assembly code, the code is first automatically assembled behind the stage into an object code. If this operation is successful, the second phase (object-code loading) is performed, otherwise an error message is shown and the loading process is aborted.

The "Start" button starts the execution of the instructions one by one. When this button is pressed, its label changes to "Stop" and if pressed again the execution is stopped. The execution also stops on any of the user-defined breakpoints.

The speed of the execution is set by the "Speed" parameter in the program settings. The unit of this parameter is kHz (i.e., if the value is set to X then X*1000 instructions are executed per second).

The application also supports step-by-step execution. In particular, the next instruction is executed when the user presses the "Step" button.

Figure 2 Control panel contains the buttons to access the main windows.

The buttons labeled "CPU," "Mem," "Screen," and "Settings" toggle the visibility of the "CPU window," the "Memory-dump window," the "Screen window," and the "Settings window," respectively.

CPU Window

The CPU window represents the heart of the SICSIM application. See Figure 3, which depicts the window with the assembly program from Listing 1 loaded into the simulator.

In the upper part of the window the registers A (accumulator), S and T (general-purpose registers), L (linkage register), B (base register), X (index register), SW (status word register), and F (floating-point register) are displayed. The value of the PC (program counter register) and the instruction to be executed are also available here. Entering a new value into the text field and pressing the Enter key can modify all the displayed values. Notice that the size of the registers is three bytes (one word), except the register F, whose size is two words.

In the middle part of the window, SICSIM provides a disassembly view of the 16 machine instructions. Each line of the view gives the following information: the instruction address, the operation code, the mnemonic, and the operands. From this presentation the user can read the instruction format, the addressing mode, and the target address. The content of the line varies slightly depending on the instruction format and the addressing mode as follows.

PC-Relative Addressing. In this addressing mode, the operand represents an offset of the target address from the PC register. Examples of such addressing are presented in Figure 3 at addresses 0000B and 00012. When SICSIM detects PC-relative addressing it shows both pieces of information: the formula used to calculate the address (e.g., (PC) + 7) and the calculated address (e.g., =1C).

Base-Relative Addressing. In the base-relative addressing mode, SICSIM shows the operand in the (B) + N form, where N is a 12-bit non-negative value (e.g., JSUB (B) + 0 at address 00008 in Fig. 3).

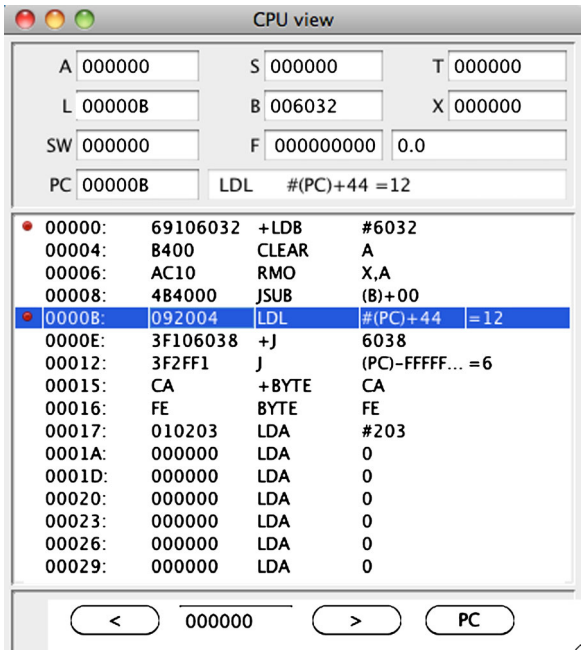

Figure 3 CPU window with the program from Listing 1 loaded onto the address 00000; the execution of the program is stopped at the address 0000B. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

```
PROG    START 0
FIRST +LDB   #SAVEA
       BASE   SAVEA
       CLEAR A
LOOP    RMO    X, A
       JSUB   SAVEA
       LDL    #ENDL
      +J      INCX
ENDL    J      LOOPX

DATA    BYTE   X'CA'
       BYTE   X'FE'
       WORD   X'010203'
       BYTE   X'01'
       BYTE   X'02'

LOOPX   J      LOOP

BUFFER RESW   8200

SAVEA   STCH   BUFFER,X
       RSUB
INCX    LDT   #1
       ADDR   T, X
       RSUB

       END    FIRST
```

Listing

Listing 1 Listing of a demo assembly program that illustrates several SIC/XE addressing modes and instruction usages. The program is also used throughout this paper to demonstrate the capabilities of SicSim.

Figure 4 Starting address of a dissembled code.

Direct Addressing. When the address of the operand is too far away from both the PC and B registers, the direct-addressing mode is used. In this case, SicSim shows the target address of the operand (e.g., Fig. 3, instructions at addresses 00000 and 00000E).

In the lower part of the CPU window, the starting address of the disassembled code is displayed in a text field (see Fig. 4). This address can be changed by entering a new value into the text field or by pressing one of the "<" (previous instruction), ">" (next instruction), or "PC" (instruction at PC address) buttons.

Tooltip Control

By pausing the mouse cursor over the instruction (i.e., the third column in the disassembly view), a tooltip control is popped-up (see Fig. 5) showing additional information about the instruction.

Tooltip control contains the instruction in the hexadecimal and binary notations, exposing the operation code, the mnemonic, the state of the nixbpe bits, and the value of the operand.

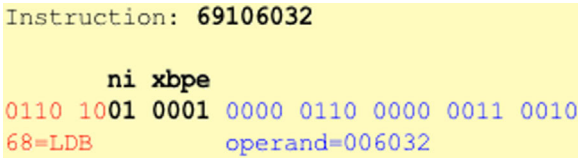

Figure 5 Tooltip control showing the information about the +LDB #SAVEA instruction. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Breakpoints

To monitor the code execution, SicSim provides the mechanism of breakpoints. By double-clicking on the first column of the disassembly view, the breakpoint is toggled. Breakpointed instructions are marked with a red dot (see Fig. 6).

The execution of the program is suspended whenever the PC register reaches any of the breakpoint addresses. When the program stops, the user may read and/or set the registers, check the state of the memory, and set or remove the breakpoints. Pressing the "Start" button resumes the program execution.

Disassembly

Notice that there is no general mechanism to distinguish between the data and instruction bytes. Hence, SicSim eagerly disassembles the content of the memory at the current address that is it always tries to disassemble the bytes as a SIC/XE instruction. However, since a sequence of bytes may not represent a valid instruction this may not always be possible. In this case, the disassembler shows a single byte of data as a BYTE directive.

On the other hand, the data bytes (e.g., the values of the variables) may accidentally represent an SIC/XE instruction.

Figure 6 Breakpoint at the LDL #ENDL instruction. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

For example, in Figure 3, the bytes at addresses 00015 and 00016 are shown correctly, while the bytes 010203 at address 00017 are displayed as LDA #203.

Even more complex situations may appear if the data bytes are followed by instruction bytes. In particular, when the last few bytes of data and the first few bytes of code form a valid instruction, the disassembly becomes misaligned. For example, in Listing 1, the last two bytes of data initialization (i.e., bytes 0102 at address 0001A) are merged with the first byte of the J instruction into a LDA instruction. The remaining two bytes of J instruction are displayed as BYTE directives (see Fig. 3, addresses 0001A to 0001E).

Memory-Dump Window

The memory-dump window displays the state of the selected part of the virtual machine memory (see Fig. 7). A total of 256 bytes are displayed in 16 rows with each row showing 16 bytes of memory. In particular, each row consists of:

- the address of the row;
- 16 bytes of memory shown in the hexadecimal notation; and
- 16 bytes of memory shown in the ASCII notation.

Since SIC/XE has a 20-bit address space (i.e., 1 MB of memory), the address of each row is shown with five hexadecimal digits.

The text field at the bottom of the window contains the starting address of the memory dump. The user can change this address by entering the value into the field or by using the buttons labeled "<", "<<", "<<<", ">", ">>", ">>>" where a button with $n = 1, 2, 3$ symbols ">" (resp. "<") increases (resp. decreases) the starting address by $2^{4(n+1)}$. In other words, by using these buttons one of the first three digits of the starting address is incremented or decremented.

Input/Output Interface

The SIC/XE computer, as defined in [1], provides a communication with the outside world through 256 character-oriented devices. However, such a communication offers only the basic functionality. To fill in this gap, our simulator offers a concept of standard input and output (as in the Unix family of operating systems [13]) and a virtual screen.

The first functionality is offered by mapping the devices, numbered 0, 1, and 2, to the standard input, the standard output, and the standard output for errors, respectively. Using this, users can easily write and simulate console-based programs.

Our simulator also includes a window showing a virtual screen. In particular, the screen is textual, capable of displaying 25 rows of 80 ASCII characters. The dimension of the screen can be changed via the settings window. Students, in their projects, often implement more advanced screens, i.e., including colors, or even graphical screens. The screen also enables students to write more entertaining programs, such as a bouncing-ball game, or simple image viewers. Figure 1 shows the screen window displaying the trail of a bouncing-ball game.

IMPLEMENTATION

The application consists of several software components belonging to the two groups: the application *front end*, which includes the GUI components, and the application *back end*, which includes the virtual machine. First, we briefly describe the former; afterwards, we provide a detailed description of the latter.

Front End

The look and feel of the views comprising the front end was already described in the previous section. Here, we give only a few implementation guidelines.

Several views show the state of the machine. During the automatic execution mode such views are regularly updated. These updates are achieved through the use of Java timers. Notice that the updating period of the screen is different from the period of the emulated-machine clock. Additionally, different views may have different updating periods: for example, the screen view has faster updates than any other view. All the updates are implemented in the method updateView() of a view.

To express their ideas freely, students are encouraged to design and implement the front end with their own preference. However, in practice, most of the students opt for a design based on our sample application. Nevertheless, they may, for example, join several views or separate the others, or even implement a text-based user interface. In general, the front-end design and implementation is a very straightforward process.

Back End

On the other hand, the back-end design and implementation is slightly more involved. Hence, students need much more guidance, which also increases the low-level compatibility of the final applications, i.e., the same object code may be executed on various implementations.

The implementer of the machine needs to decide on a way to represent registers, memory, input/output devices, the instruction-set emulation technique, exception handling, etc. Notice that, all the Java integer data types are signed, while the SIC/XE instruction operands are unsigned.

Registers and Memory

To represent 24-bit integer registers of the SIC/XE architecture we use Java's 32-bit int type with special precautions taken to truncate the upper 8 bits. For each register, the getter and the setter are provided. Since many instructions address registers by their index, the support for the index getter and setter is also added. The registers and their indices are shown in Table 1 [1].

Another challenge is the implementation of the floating-point register F. Notice that F is a 48-bit register that is of a non-standard size (IEEE 754 standard prescribes 16, 32, 64, or 128 bits for representing floating-point numbers). The format of the register F is shown in Fig. 8a).

However, to support floating-point operations one need not to get involved with all the cumbersome details. Register F can be represented internally with the Java double data type, which conforms to the IEEE 754 standard 64-bit floating-point numbers (see Fig. 8b). The format of double resembles that of the register F: the position and number of bits used for the sign and exponents are the same, but the mantissa of double is 52 bits (16 bits longer). The exceeding bits may simply be truncated.

SIC/XE has 1 MB of memory. It is internally represented as a Java array of bytes. Since SIC/XE instructions manipulate bytes, words (3 bytes), or floats (6 bytes) the corresponding getter and setter methods are provided. They are summarized in Table 2.

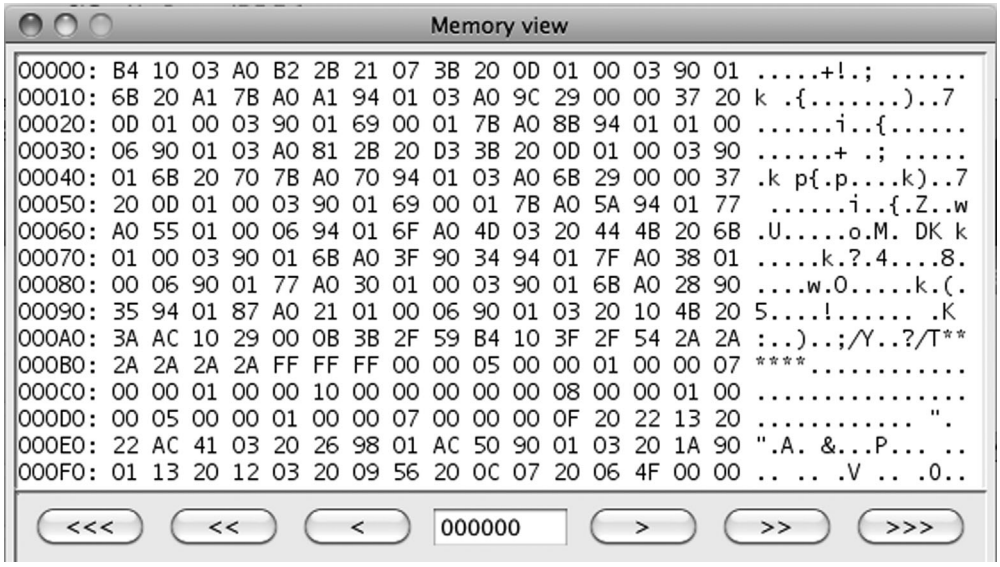

Figure 7 Memory-dump window.

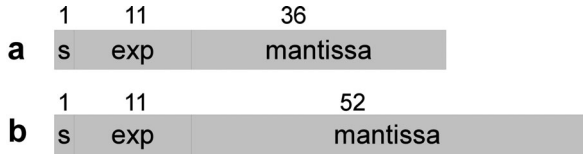

Figure 8 SIC/XE and IEEE 754 standard floating-point formats.

Input/Output Devices

To work with I/O devices the SIC/XE architecture provides three instructions (TD—test device, RD—read one byte form device, WD—write one byte to device).

Our implementation of the simulator supports three kinds of devices.

Input Device. Supports reading from the Java's `InputStream`. May be initialized with `System.in` and hence providing access to the standard input.

Output Device. Supports writing to the Java's `OutputStream`. May be initialized with `System.out`, thus providing access to the standard output.

File Device. Supports reading and writing to a file. On writing, if a corresponding file does not exist, it is automatically created. Reading past the end of file is indicated by returning zero.

Table 1 The SIC/XE Registers (First Row) and the Corresponding Indices (Second Row)

A	X	L	B	S	T	F	PC	SW
0	1	2	3	4	5	6	8	9

Each device implementation inherits the base class `Device` and must provide the three methods summarized in Table 3.

The SIC/XE architecture can address a maximum of 256 devices. In our implementation addresses 0, 1, and 2 are reserved for the standard input, the standard output, and the standard output for errors, respectively. Notice that these addresses correspond to the standard Unix file descriptors [13]. All other addresses are initialized as `FileDevice` with the name of the file set to the hexadecimal device address with `.dev` extension. An overview of the device initialization is represented in Figure 9.

Emulation Background

Now let us focus on the emulation of the SIC/XE instruction-set architecture. There are two machines involved in the process.

Guest Machine. The machine that is emulated. In our case, the SIC/XE.

Host Machine. The machine on which the emulation is made. In our case, the Java Virtual Machine.

There are many approaches to emulation, ranging from *interpretation* on one end to *binary translation* on the other [14]. Here, we focus on the former. Interpretation works in several steps: first the instruction operation code is fetched and decoded, then the operand is fetched and decoded, and lastly, the instruction is executed.

Several techniques [14] for implementing interpreters exist.

Decode and Dispatch Loop. The basic technique consists of one big `switch` statement, which—based on the operation code—dispatches the program flow to the corresponding emulation code. The `switch` is often enclosed in a `while` loop. This technique is represented in Figure 10.

Table 2 SIC/XE Memory-Access Methods

int	**getByte** (int addr)
void	**setByte** (int addr, int v)
int	**getWord** (int addr)
void	**setWord** (int addr, int v)
double	**getFloat** (int addr)
void	**setFloat** (int addr, double v)

Indirect Threaded Interpretation. This technique surmounts the inefficiency drawback of the previous one by appending the fetch and dispatch code to the emulation code of each instruction. Additionally, dispatch is usually done via the dictionary that maps the operation codes to routines.

Predecoding. All the instructions and operands are decoded prior to their execution. A suitable data structure to represent the decoded instruction and operands is used for easier and faster execution.

Direct Threaded Interpretation. Combines predecoding and indirect threaded interpretation.

For demonstration purposes, our implementation of the simulator is based on the first technique, which is also the one that is most used by the students. However, more advanced students may choose other techniques as well. Notice, that other techniques may only be suitable for implementation in programming languages that support pointers such as C/C++.

Emulation

There are five instruction formats available in the SIC/XE architecture; they are shown in Figure 11.

Formats F1, F2, F3, and F4 are supported only by the (new) SIC/XE architecture, whereas the format SIC is the format of the (old) SIC architecture.

Complex instruction set computers, such as SIC/XE, demand the use of additional decoding techniques [14]. The decoding is usually divided into two (or more) phases: the first phase detects the format of the instruction, while the second actually executes the corresponding emulation code, which is (in our case) implemented with the decode-and-dispatch-loop technique.

One important observation is that the SIC, F3 and F4 formats consist of the same subset of instructions; only the operands are of different sizes. Hence, after the decoding of the operand a common emulation routine is called.

In general, the SIC/XE instructions are interpreted according to the following steps:

1. Fetch and decode one byte from the PC address; the high 6 bits of the fetched byte represent the instruction operation code.
2. If the F1 format is detected then execute the corresponding emulation code and return.

Table 3 SIC/XE Device-Access Methods

boolean	**test**()
byte	**read**()
void	**write**(byte value)

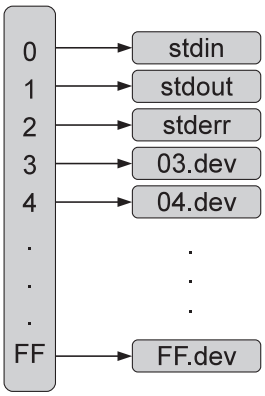

Figure 9 Initialization of input/output devices.

3. Fetch the next byte from the PC address.
4. If the F2 format is detected then decode the fetched byte (representing the F2 format operands), execute the corresponding emulation code and return.
5. Based on the n, i and e bits detect format: if both bits n and i are zero then the format is SIC, otherwise if e is zero then the format is F3. If none of the above, then the format is F4.
6. Fetch the right number of bytes (depending on the format detected in step 5) and decode the operand.
7. Execute the common emulation routine for SIC, F3, and F4.

The operand-decoding step 6 must also consider bits b and p to correctly calculate the *target address*. In particular, SIC/XE supports PC-relative addressing (the operand is an offset from PC), base-relative addressing (the operand is a displacement from the register B), and direct addressing. The meaning of the bits b and p is given in Table 4.

The use of the target address is encoded in the bits n and i. SIC/XE supports simple addressing (operand gives the address of a value), immediate addressing (operand gives an immediate value), indirect addressing (operand gives the address of the

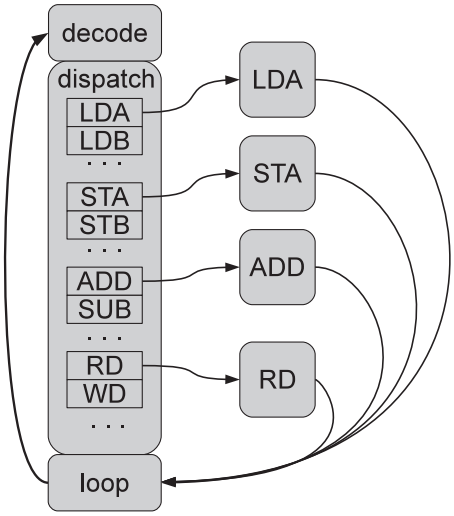

Figure 10 Decode and dispatch loop for the SIC/XE instruction set architecture.

address of a value). The meaning of the bits n and i is summarized in Table 5. Notice that these bits can also specify the SIC instruction format.

During the interpretation one must also handle any decoding errors. For example, if no matching operation code is found then the invalid operation-code exception is thrown. The front end of the simulator catches the exception and prints a suitable error message. Similarly, if the bits specifying the addressing mode are wrong then the invalid-addressing exception is thrown.

CASE STUDIES

In this section, we present two case studies: the loader mechanism with bootstrapping, and the concept of memory-mapped input/output that is used by the screen.

Loader

A *loader* is a computer program that loads another program, whereas a *boot loader* loads an *operating system* or a *runtime environment*.

In our case, the loader is implemented in Java as a part of the simulator. The loader loads the program from a file called the *object file*. The format of the SIC/XE object files uses a human-friendly hexadecimal notation of binary values.

Basically, there are two kinds of loading:

Absolute Loading. The program is loaded to the address to which it has been compiled.

Relative Loading. The program may be loaded to any address and program relocation must be applied.

To support absolute loading we need at least the following records in the object file: H—header record containing program name, the origin (or start address) of the code and the code length, T—text record containing the code address followed by the length and the hexadecimally encoded code, and *E*—end record containing the start address of the program. The fields are summarized in Figure 12.

To implement the loader in the simulator one needs to consider all three records, correctly decode the hexadecimal notation, and take care to copy the loaded code to the corresponding part of the memory. After the loading, the PC register is set to the program start address.

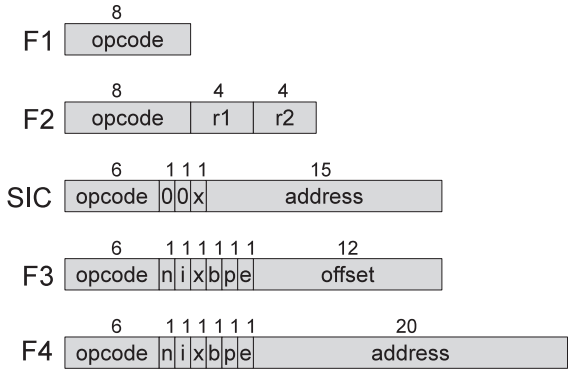

Figure 11 SIC/XE instruction formats.

Another challenge is the boot loader. An example loader implementation for SIC/XE is given in [1]. It does not support loading from the object files, but instead loads from a file containing a *raw machine code* (hexadecimally encoded).

Thus, before experimenting with the boot loader one must prepare the raw code by converting an object file. Observe that the text records in an object file may not initialize all the memory that the program uses, i.e., arbitrary gaps may be present between the text records. In the conversion such gaps may be filled with zero values.

Screen

The screen device is implemented using the memory-mapped input/output technique [4,5], i.e., a part of the main memory is reserved for the data representing screen content. In particular, the virtual screen regularly refreshes itself using the data in the reserved part of the memory.

For a simple textual screen, where a memory location corresponds to one character, our simulator reserves memory locations from SCR to SCR + COLS*ROWS − 1, where SCR is the starting address of the screen, COLS is the number of columns, and ROWS is the number of rows.

To put a character onto the screen a user simply writes an ASCII code to a corresponding memory location. The memory location l that corresponds to the position in the *x*-th column and the *y*-th row is calculated according to the following formula:

$$l = SCR + y * COLS + x.$$

Of course, different organizations of memory are possible. For example, when implementing a color screen one must also include information about a color, and when implementing a graphical screen, pixels are used instead of characters. For representing colors several options are possible. Full color depth (24 bits) may be used, which may be too much for the SIC/XE memory size, even for images of moderate size. Instead, one can employ an *indexed color* technique to reduce the space consumption. Here, a color is an index to an array of a full depth palette of colors.

Educational Experiences

The use of hypothetical computers and their software simulators is already pervasive in computer education. The main reason for this

Table 4 Target-Address Calculation Bits

b	p	Meaning
0	0	Direct addressing
0	1	PC-relative addressing
1	0	Base-relative addressing
1	1	Invalid addressing

Table 5 Target-Address Usage Bits

n	i	Meaning
0	0	Simple addressing
0	1	Immediate
1	0	Indirect
1	1	SIC format

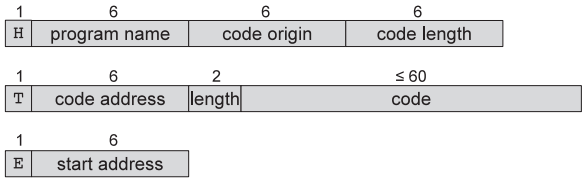

Figure 12 Formats of the object-file records.

may be found in the design of hypothetical computers, which often excludes unnecessary low-level technical details.

In our experience, software simulator of such computers can be implemented by students with reasonable effort as a one semester project and are much cheaper than corresponding hardware solution. Additionally, they are also easy to maintain and extend with additional features. Consequently, their use in teaching is quite apparent.

In the beginning of the course the students learn the SIC/XE architecture and the corresponding assembly language programming. To decrease the learning curve and increase comprehension the simulator is used by lecturers for presentation and demonstration and by students for laboratory practice as well as when studying at home.

Another course topic is run-time systems and virtualization. From the hand-on experience with the simulator, students improve their understanding on creating a virtual machine, and also are able to create their own.

Finally, the simulator is used when studying assembling, linking and loading. Here most of the students use their own simulator. Hence, in the process, they test for and also remove any software bugs, and generally improve and extend their simulators.

By working on a project such as the computer simulator students grasp several topics (e.g., virtualization, assembling, software design) at the same time.

During the design and implementation of the simulator, we also encourage the students for any ideas and implementations of the extensions. Among the most popular are: GUI improvements, a disassembly view, an addition of devices, an execution statistics, and a graphical screen.

CONCLUSION

In the paper, we described a simulator for the SIC/XE computer. First, we focused on the functionality of the graphical front end (CPU window, Memory-dump window, and Screen window) from the user's point of view. Second, we presented the details of the implementation of the simulator's back end (registers, memory, devices, instruction emulation, loader, etc.).

In addition to the basic functionality, such as data representation and instruction execution, automatic and stepping execution mode, our simulator also supports several extended features, such as disassembly view, support for standard input, output and output for errors and textual screen.

The simulator is a part of a larger software toolbox used in teaching a course about system software. The other part of the toolbox is an assembler for the SIC/XE assembly language, which may be used independently and is also integrated into the simulator.

Our experiences in using a simulator within a system software course are positive. Instead of teaching a bare theory we offer our students a live system with lots of interesting and illustrative programming examples. Running and debugging these examples facilitates the students to understand the concepts of system software. Thus, it is not surprising that the students respond with an enthusiasm and a lot of fresh ideas about how to use and enhance the simulator.

From the pedagogical experience with teaching lessons as well as laboratory practices we conclude that the use of a computer simulator can enable students to quickly grasp the details of a new computer architecture and the corresponding assembly level programming, to experience the concept of a run-time system and process virtual machine from hands-on practice, and to, finally, be able to actually create their own virtual machine.

We believe that all of these features make our simulator a must-have tool when teaching a course on system software.

REFERENCES

[1] L. L. Beck, System software: An introduction to systems programming, Addison-Wesley, Reading, MA, 1997.
[2] R. A. Campbell, Introducing computer concepts by simulating a simple computer, SIGCSE Bull 28 (1996), 9–11.
[3] D. E. Knuth, The art of computer programming, vol. 1, Addison-Wesley, Boston, MA, 1997.
[4] J. L. Hennessy and D. A. Patterson, Computer architecture: A quantitative approach, 4th ed., Morgan Kaufmann Publishers, San Francisco, CA, 2007.
[5] A. S. Tanenbaum and J. R. Goodman, Structured computer organization, 4th ed., Prentice Hall PTR, Upper Saddle River, NJ, 1998.
[6] L. Moreno, E. J. González, B. Popescu, J. Toledo, J. Torres, and C. Gonzalez, MNEME: A memory hierarchy simulator for an engineering computer architecture course, Comput Appl Eng Educ 19 (2011), 358–364.
[7] I. Castilla, L. Moreno, C. González, J. Sigut, and E. González, SIMDE: An educational simulator of ILP architectures with dynamic and static scheduling, Comput Appl Eng Educ 15 (2007), 226–239.
[8] J. E. Sayers and D. E. Martin, A hypothetical computer to simulate microprogramming and conventional machine language, ACM SIGMICRO Newsl 19–20 (1989), 4–10.
[9] L. L. Beck, Accompanying software for "System Software: An Introduction to Systems Programming", http://rohan.sdsu.edu/faculty/beck.
[10] D. Chen, DAssembler for SIC/XE, a simple SIC/XE assembler implementation, 2011, goo.gl/fmvuX.
[11] S. Sultan, A. S. Mousa, and L. I. Zuhair, SIC/XE simulator and assembler, 2011, goo.gl/cSWu4.
[12] G. Fox, Sicvm—A SIC based virtual machine, 2006, goo.gl/nQbwp.
[13] W. Richard Stevens, Advanced programming in the UNIX environment, Addison–Wesley, Boston, MA, 1993.
[14] J. E. Smith and R. Nair, Virtual machines: Versatile platforms for systems and processes, Morgan Kaufmann, San Francisco, CA, 2005.

BIOGRAPHIES

Jurij Mihelič received the doctoral degree in computer science from the University of Ljubljana, Slovenia, in 2006. Currently, he is with the Laboratory of Algorithms and Data Structures, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, as an assistant and a researcher. His research interests include combinatorial optimization, heuristics, approximation algorithms, computational complexity, graph problems, and uncertainty in optimization problems.

Dr. Tomaž Dobravec received his dipl. ing. degree (1996) in mathematical science and his PhD (2004) in computer science, both from the University of Ljubljana. He is an Assistant Professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests are in algorithm design, computer security and networks.