Three Simulator Tools for Teaching Computer Architecture: EasyCPU, Little Man Computer, and RTLSim

CECILE YEHEZKEL
Weizmann Institute of Science
and
WILLIAM YURCIK
Illinois Wesleyan University
and
MURRAY PEARSON and DEAN ARMSTRONG
University of Waikato

Teaching computer architecture (at any level) is not an easy task. To enhance learning, a critical mass of educators has begun using simulator visualizations of different computer architectures. Here we present three representative computer architecture simulators for learning which show that there is a growing consensus for computer simulation as a teaching tool for complex dynamic processes, such as underlying computer operations. Simulators also show the wide spectrum of pedagogical goals for teaching computer organization and architecture. Specifically, the three simulators we describe are (1) EasyCPU for the Intel 80x86 family of CPUs; (2) Little Man Computer for a general von Neumann computer architecture; and (3) RTLSim, a data path simulator for a MIPS-like CPU. An appendix is provided for more detailed descriptions of each simulator.

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**]: Computers Uses in Education

General Terms: Design

Additional Key Words and Phrases: Computer architecture simulators, education

1. INTRODUCTION

The description and explanation of internal computer operations are not easy tasks. The processes involved are sophisticated and occur at a very high speed. Students of computer science and electrical and electronic engineering acquire a cognitive model of the internal computer operation through many courses and labs, and this model is often incomplete or erroneous. One of these courses is computer architecture and assembly language programming, in which students learn about the internal structure of the computer and how instructions are handled and executed [Loui 1988]. There are several commercial tools for developing assembly language programs: tools that include editing, assembling,

Authors' addresses: C. Yehezkel, Department of Science Teaching, Weizmann Institute of Science, Rehovot, Israel; email: yehezkel@hait.ac.il; W. Yurcik, Department of Mathematics and Computer Science, Illinois Wesleyan University, Bloomington, IL 61702-2900; email: byurcik@iwu.edu; M. Pearson and D. Armstrong, Department of Computer Science, University of Waikato, Hamilton, New Zealand; email: {mpearson,daa1}@cs.waikato.ac.nz

linking, executing, and debugging utilities. These kinds of tools were developed for advanced students or for professional programmers, but do not usually meet the needs of novice students at the university level [Simeonov 1995]. The goal is to provide tools to allow students to experience different computer architectures. Visualization based on animated simulation[1] is widely used in the sciences to help researchers and students understand the systems they study. “Simulations provide a guiding context within which students can integrate what they learn”, but are relatively underutilized in computer science itself [Fishwick 2000]. In fact, simulating a system under study on the very system under study is recursive.[2] What is needed are intuitive “visual metaphors” for representing how a computer system works by extracting the important operations and eliminating product-specific operational noise [Hanrahan 1996]. Thus, as computer systems become more complex, visualization is one tool to deal with the emerging complexity and the increasing rate of execution processing.

This article is organized as follows: Section 2 presents a brief survey of state-of-the-art computer simulators for teaching and describing design parameters and how didactic goals induce simulation characteristics. Section 3 describes three simulation environments that were specifically developed for teaching introductory computer architecture. In Section 4 we discuss the benefits of computer simulators. We conclude in Section 5.

2. EDUCATIONAL SIMULATORS

2.1 Designing Simulators as Teaching Tools

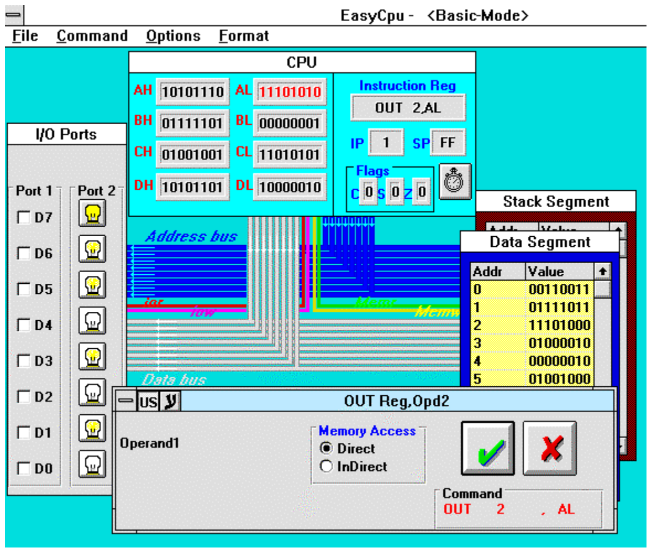
Many educators and academic researchers who try to communicate introductory computer architecture concepts relate significant difficulties [Cassel et al. 2001]. There are no adequate hardware tools to visualize the real-time dynamic interaction of software-with-hardware during program execution at the machine level. Because existing tools are mainly dedicated to debugging by professional programmers, they are too sophisticated for students. Educators, motivated by the intuition that visualization can help overcome teaching difficulties, have developed graphical simulators that focus on troublesome subject matter. The development of such simulators is often spontaneous, and has led to a variety of disparate simulators, each targeted to a small student population in order to meet local teaching needs. The goals of the developers have defined the different characteristics of their individual simulators. The targeted population is the dominating factor in determining the level of complexity and features provided to the student through the graphical interface.

2.2 Simulator Characteristics

Computer architecture simulators are either hypothetical or related to a specific computer architecture. Some of the models were invented by their developers or meant to comple-

[1] [Foley 1998] pointed out that the common goal of visualizations is: transforming information into a meaningful, useful visual representation from which a human observer can gain understanding. Hence the term visualization is in many cases more appropriate than simulation.

[2] We make the following distinction between emulation and simulation: Emulation is an attempt to imitate the internal design of a device, while simulation is an attempt to imitate the external functions of a device [Fayzullin 2002]. Thus, the tools we discuss in this article are simulators, in that their focus is on the external functionality of real machines for teaching computer operations and not on replicating internal operations, although it can be argued that most simulators attempt to do both.



AH	10101110	AL	11101010
BH	01111101	BL	00000001
CH	01001001	CL	11010101
DH	10101101	DL	10000010

Instruction Reg: OUT 2,AL
IP 1 SP FF
Flags C 0 S 0 Z 0

Addr	Value
0	00110011
1	01111011
2	11101000
3	01000010
4	00000010
5	01001000

Memory Access: (● Direct) ○ InDirect
Command: OUT 2 , AL

Fig 1. OUT 2, AL in EasyCPU in Basic Mode.

ment a textbook to fit to pedagogical goals. Others are based on a specific computer with a level of complexity that matches the target population. A computer may be modeled at different layers in the hardware-to-software hierarchy. For example, a simulator may model anything from a high-level CPU operation down to the execution of microcode. The software and hardware aspects within a model are balanced depending on the orientation of the target population. Developers create the learning environment by defining the role of the student in the simulation, contingent on the skill set a student needs to acquire. It may be appropriate for a student to operate a CPU control-unit or to program in assembly language. Simulators reinforce student understanding of CPU operations during instruction execution in the former case and the processes of program execution at the machine level in the latter case.

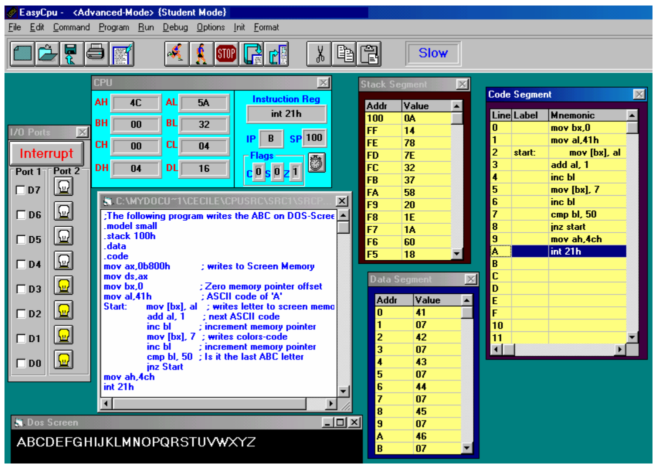
Each of us has been deeply involved in the development of a different simulator. These simulators are described in the next section, highlighting their specific characteristics due to different pedagogical goals.

3. EXAMPLES OF SIMULATORS

3.1 EasyCPU

The EasyCPU environment was developed for an introductory-level computer science target population. The model is a simple version of a microcomputer based on the Intel x86 microprocessor. It provides the student with basic tools to edit, assemble, run, and debug small programs in a window-friendly environment. It focuses on the visualization of the execution process of individual assembly instructions alone and within a program.

EasyCPU is a program for a PC/Windows environment that simulates the operation of a simplified microprocessor from the Intel 80x86 processor family. The EasyCPU assembly language is a reduced subset of Intel 80x86 assembly language instruction set. The instructions selected for the EasyCPU are basic instructions including the main


EasyCpu - <Advanced-Mode> {Student Mode}
File Edit Command Program Run Debug Options Init Format
STOP
Slow
CPU
I/O Ports
Interrupt
Port 1
Port 2
D7
D6
D5
D4
D3
D2
D1
D0
Flags
Dos Screen

AH	4C	AL	5A
BH	00	BL	32
CH	00	CL	04
DH	04	DL	16

Instruction Reg: int 21h
IP B SP 100
Flags: C 0 S 0 Z 1

C:\MYDOCU~1\CECILE\CPUSRC\SRC1\SRCP...

```
;The following program writes the ABC on DOS-Scree
.model small
.stack 100h
.data
.code
mov ax,0b800h          ; writes to Screen Memory
mov ds,ax
mov bx,0               ; Zero memory pointer offset
mov al,41h             ; ASCII code of 'A'
Start:     mov [bx], al   ; writes letter to screen memo
           add al, 1      ; next ASCII code
           inc bl         ; increment memory pointer
           mov [bx], 7    ; writes colors-code
           inc bl         ; increment memory pointer
           cmp bl, 50     ; Is it the last ABC letter
           jnz Start
mov ah,4ch
int 21h
```

Stack Segment

Addr	Value
100	0A
FF	14
FE	78
FD	7E
FC	32
FB	37
FA	58
F9	20
F8	1E
F7	1A
F6	60
F5	18

Data Segment

Addr	Value
0	41
1	07
2	42
3	07
4	43
5	07
6	44
7	07
8	45
9	07
A	46
B	07

Code Segment

Line	Label	Mnemonic
0		mov bx,0
1		mov al,41h
2	start:	mov [bx], al
3		add al, 1
4		inc bl
5		mov [bx], 7
6		inc bl
7		cmp bl, 50
8		jnz start
9		mov ah,4ch
A		int 21h
B		
C		
D		
E		
F		
10		
11		

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Fig. 2. Assembly program in the EasyCPU Advanced Mode.

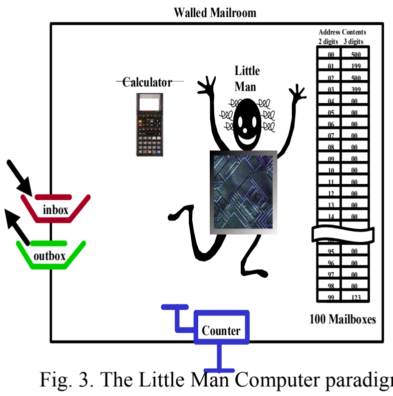
addressing modes and the principal sets of operations: move, arithmetic, and logic. EasyCPU provides students with friendly tools for developing small programs. During implementation, special care was taken to keep the EasyCPU assembler compatible with existing professional tools (assemblers, linkers, and debuggers).

EasyCPU has two modes of operation: Basic-Mode and Advanced-Mode. Basic Mode enables the novice student to learn the syntax and semantics of individual instructions. The Basic Mode screen describes the main computer units: CPU, memory segments, and elementary I/O (an array of buttons and an array of LEDs). The units are connected by three animated buses (control, address, and data).

The student is able to follow (on the screen) the data transfer between the different computer units during the instruction execution. For example: during the execution of the *"Out 2, AL"* instruction, the CPU copies the content of the AL register "11101010b" to the output Port 2*,* as can be seen in Figure 1.

The Advanced Mode is intended for students with prior knowledge of assembly language programming. It provides students with tools to develop their own programs. The students can edit a new program or open examples, assemble, link and execute the program step-by-step, or continuously (in slow, medium or fast motion). During execution, students can change the data in the CPU's registers, in the data, or stack segments. They are also able to choose the data format: binary, hex or decimal. Figure 2 shows a program running in Advanced Mode. The program depicted writes ABC on a simulated DOS-screen window.

The EasyCPU environment offers the possibility of controlling external hardware in addition to the on-screen I/O simulation. When an I/O instruction is used with indirect addressing, it is actually executed by the computer running the EasyCPU program. This makes it possible to control the hardware of the computer or external hardware connected to the computer. For example, it is possible to activate the computer speaker and to play tones using an output instruction and indirect addressing to the speaker port. This option enables the students to read and write binary data to any external hardware connected to



Address 2 digits	Contents 3 digits
00	500
01	199
02	500
03	399
04	00
05	00
06	00
07	00
08	00
09	00
10	00
11	00
12	00
13	00
14	00
95	00
96	00
97	00
98	00
99	123

Fig. 3. The Little Man Computer paradigm.

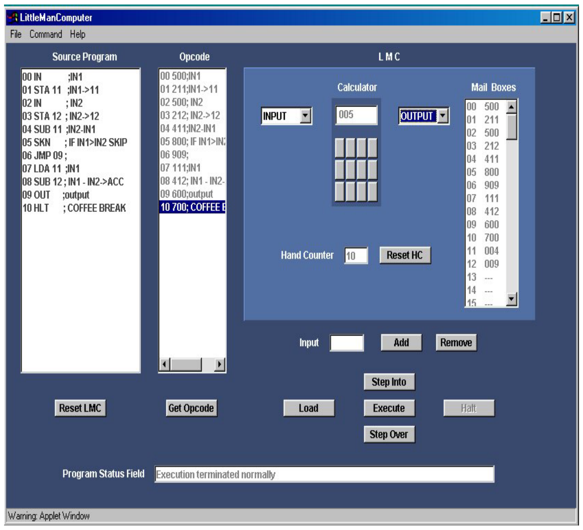
the computer and also to develop their own hardware project in which they can apply controlled mechanisms.

EasyCPU was specifically developed for educational purposes as an integral component of an introductory-level course "Computer Architecture and Assembly Language." The course was taught to 11th grade high school students as a part of a new computer science program. Since 1995, EasyCPU has been tested in 30 high schools with the involvement of over 3,000 students. Students who have used the EasyCPU environment have achieved significantly higher grades in the matriculation exam (a laboratory examination) than students who used the Turbo Assembler environment. Further analysis revealed the specific contribution of the EasyCPU environment to develop students' debugging skills. We intend to incorporate EasyCPU into other relevant curricula, particularly electronics and technology at both the high school and university levels.

3.2 Little Man Computer

The Little Man Computer (LMC) paradigm consists of a walled mailroom (see Figure 3), 100 mailboxes, a calculator, a location counter, and input/output baskets [Yurcik et al. 2001]. The calculator can be used for input/output, temporary storage of numbers, and for addition and subtraction. The location counter is used to increment a running count each time the Little Man executes an instruction. Finally, there is the "Little Man" himself, depicted as a cartoon character who performs specific tasks within the walled mailroom.

The intent is to provide simultaneous visualization of all components within a computer architecture, an ability to observe the fetch-execute cycle during program execution and a highly interactive problem-solving environment with flexible input/output demands. In Figure 4, the LMC simulator visually shows a one-pass assembly process (mnemonic assembler source code to machine code) and a load process (moving machine code into mailboxes). The program finds the absolute value of the difference between two non-negative integer inputs. In the edit mode, users can write source code (which is automatically checked for syntax errors) or load source code from an existing file. For the programmer's convenience, three different execution modes are provided: (1) Burst Mode, where all instructions in the program are executed until a HALT instruction is encountered; (2) Step Into, which executes one instruction at a time; and (3) Step Over, which is similar to Step Into, except for SKIP instructions. A flag



```
00 IN         ;IN1
01 STA 11   ;IN1->11
02 IN         ; IN2
03 STA 12  ; IN2->12
04 SUB 11  ;IN2-IN1
05 SKN      ; IF IN1>IN2 SKIP
06 JMP 09 ;
07 LDA 11  ;IN1
08 SUB 12 ; IN1 - IN2->ACC
09 OUT      ;output
10 HLT      ; COFFEE BREAK
```

```
00 500;IN1
01 211;IN1->11
02 500; IN2
03 212; IN2->12
04 411;IN2-IN1
05 800; IF IN1>IN
06 909;
07 111;IN1
08 412; IN1 - IN2-
09 600;output
10 700; COFFEE E
```

Calculator: 005

```
00  500
01  211
02  500
03  212
04  411
05  800
06  909
07  111
08  412
09  600
10  700
11  004
12  009
13  ---
14  ---
15  ---
```

Hand Counter: 10

Program Status Field: Execution terminated normally

Fig. 4. The Little Man computer simulator.

register indicates error conditions and is set/reset for corresponding overflow/underflow conditions and zero/negative/positive values within the calculator.

It should be observed that LMC is a direct implementation of a von Neumann architecture. The calculator corresponds to the arithmetic logic unit (ALU); mailboxes correspond to memory; the location counter corresponds to the program counter; the I/O corresponds to input/output baskets; and the Little Man himself corresponds to the control unit.

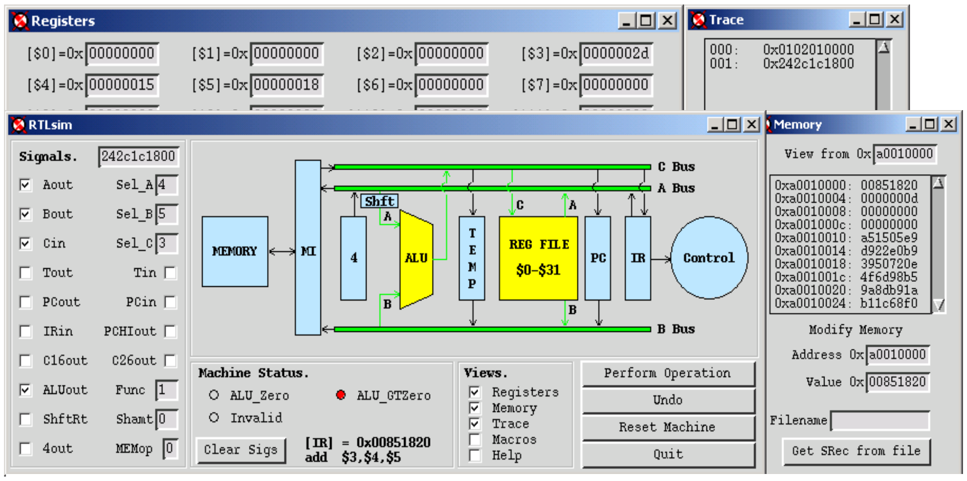
We have found that LMC visualization provides an opportunity to examine the concepts underlying the von Neumann architecture. While instructions and data can be input, a program is not necessarily executed in the same order as the code is input, there must be a place to temporarily store both instructions and data. Without didactic lecturing, students independently learn the concept of memory that can lead to discussion of the memory hierarchy and virtual memory.

Addressing is usually one of the more challenging concepts to impart, but the visualization of addressing modes (including immediate and indirect addressing) has made these concepts intuitive, such that focus can be placed on advanced concepts. An example of an advanced concept is the relationship of the operating system to computer architecture in terms of bootstrapping, loading, time-sharing, and virtual memory.

3.3 RTLSim

RTLSim is a UNIX program that simulates the datapath of a simple nonpipelined MIPs-like processor. When running the simulator, the student (user) acts as the control unit for the datapath by selecting the control signals that will be active in each control step.

Figure 5 shows the RTLSim main window which is comprised of two main components: (1) a visual representation of the datapath and (2) a control signals window. The datapath is made up of a 32-register register file, ALU, memory interface, and other



```
[$0]=0x00000000  [$1]=0x00000000  [$2]=0x00000000  [$3]=0x0000002d
[$4]=0x00000015  [$5]=0x00000018  [$6]=0x00000000  [$7]=0x00000000
```

```
000:    0x0102010000
001:    0x242c1c1800
```

```
Signals.  242c1c1800
☑ Aout      Sel_A 4
☑ Bout      Sel_B 5
☑ Cin       Sel_C 3
☐ Tout      Tin ☐
☐ PCout     PCin ☐
☐ IRin      PCHIout ☐
☐ C16out    C26out ☐
☑ ALUout    Func 1
☐ ShftRt    Shamt 0
☐ 4out      MEMop 0
```

```
Machine Status.
○ ALU_Zero      ● ALU_GTZero
○ Invalid
[IR] = 0x00851820
add  $3,$4,$5
```

```
Views.
☑ Registers
☑ Memory
☑ Trace
☐ Macros
☐ Help
```

```
View from 0x a0010000
0xa0010000:  00851820
0xa0010004:  0000000d
0xa0010008:  00000000
0xa001000c:  00000000
0xa0010010:  a51505e9
0xa0010014:  d922e0b9
0xa0010018:  3950720e
0xa001001c:  4f6d98b5
0xa0010020:  9a8db91a
0xa0010024:  b11c68f0
Modify Memory
Address 0x a0010000
Value 0x 00851820
Filename
```

Fig. 5. RTLSim main window.

registers to store values (including the program counter and current instruction register). Three internal buses are used to connect these components. It is possible to execute most MIPs instructions on the datapath. The control signals section at the left-hand end of the main window can be used by the student to set the values of control signals that are going to be active in the current control step. For example, consider the execution of the MIPs instruction "`add $3, $4, $5`" (i.e. R3 = R4 + R5). Assuming the instruction was fetched into the instruction register (IR), then the settings shown in the control signals window of Figure 5 will execute this instruction.

As control signals are set, a student is given visual feedback on the datapath about what will occur when the control step is executed. For example, if the PCout signal is selected, the colors of the PC register and bus 2 change to show that the PC register is going to output a value onto bus 2. If two components try to output to the same bus concurrently, it will turn red to indicate an illegal operation.

Windows that show the contents of the memory and the register file may also be opened. The simulator can also record a trace of operations performed in each control step, to be used for playback purposes or input to an automated marking system.

RTLSim is used in a second-year computer systems course at the University of Waikato. In the first half of the course the students are taught the relationships among programs written in a high-level programming languages such as C, data representations in assembly and machine language, and finally how the machine code representation can be executed on a processor [Pearson et al. 1999]. RTLSim is used to reinforce concepts learned in the final section on executing machine code on a processor.

Before the introduction of RTLSim to the course, the students were given a paper-based exercise on RTL-level execution of MIPs-like instructions. One of the major problems with this exercise is that the students were not given feedback on the exercise until after it was marked. However, with the introduction of RTLSim students are now given immediate feedback at several levels. First, students are given visual feedback on the datapath as they set the control signals. Second, once students believe they have the necessary signals to execute the control step, they can try it and observe the outcome in the registers and memory. If the outcome is incorrect, the simulator provides undo operations so they can try again. Third, an automated marking system is used. If the exercise is not completed correctly, the marking system generates a set of comments that tells the students where they went wrong so they can try again.

4. BENEFITS AND EVALUATION

The use of simulators has provided a number of additional benefits in terms of (1) financial support; (2) obsolescence; (3) access, and (4) research. First, teaching computer organization at the novice level often leads to examples of machines that either no longer physically exist except in museums, or if the machines do exist, are too expensive to justify solely for educational purposes. One author's experience is that a DEC VAX was maintained at the University of Pittsburgh into the new millennium on the basis of superior educational material developed specifically for teaching VAX architecture and assembly language. The use of simulators frees educators to teach concepts on any machine if a simulator is available, without having expensive support contracts for machines dedicated to educational purposes. A second, closely related, benefit is that obsolescence is no longer a factor, since computer systems can be maintained indefinitely as simulators, and computers that no longer exist can be virtually recreated as simulators. Third, a Web-enabled simulator has ubiquitous access for students and interested parties throughout the world via the Internet. Students can interact with the simulators asynchronously at any time for as long they want. In fact, Web-based simulators allow educators to introduce and use many different types of computer systems that otherwise would only have been discussed in textbooks. Finally, simulators allow students to experiment with well-known computer architectures, as well as the ability to create and test their own computer architectures. Performance tradeoffs and design constraints become real when students attempt to create a novel computer system as a simulator.

Developers are motivated by educators' intuition that visualization can contribute to overcoming their teaching difficulties. Petre et al. [1998] claim that visualization provides "an informative impression of the whole which provides insight into the structure." Unfortunately, the relative scarcity of empirical evaluation in this area is accompanied by virtually no work on effective design claims [Stasko 1998]. The diversity of available simulators due to different pedagogical goals hinders the design of uniform evaluation methods for large populations or comparative research. The effectiveness of each simulator has to be evaluated in its own context according to well-defined individualized criteria. Qualitative research tools may yet answer these specific field research questions in the future. Nevertheless, current anecdotal evidence leads us to believe that simulators are effective teaching tools. There is no doubt that they improve student motivation.

5. CONCLUSIONS

We came to the following conclusions:

- modern processors are optimized for performance and not simplicity;
- simulation is increasingly being used as a tool to support the teaching of computer architecture; and

- when simulators are combined with visualizations, they become an even more effective teaching tool.

As processors have become more complex, they have also become less suited for teaching introductory computer organization and architecture courses. Fortunately, the advent of simulators have been able to fill this role. Simulators also have the advantage of giving students a portal into the internal operation of the CPU that is not possible with real-time hardware. Most of the early simulators were text-based. However, as the power of processors has increased and programming environments have improved, it has become easier to build simulators that employ a graphical interface. Improved graphical user interfaces have made it possible to provide an intuitive view of the internal operation of a computer. The three examples of computer architecture simulators described in this article show how different visualization techniques can be used to aid a student with learning concepts associated with introductory assembler language programming and CPU operation. In EasyCPU, color changes and animations are used in the basic mode to illustrate datapath operations during the execution of a single instruction. Little Man Computer uses color and movement to highlight the interaction between the fetch-execute cycle and memory. In the case of RTLSim, color is used to show a student the effects of control signal settings.

The large variety of computer architecture simulators is the outcome of a wide range of needs, reflecting different target populations, subject matter, and different approaches to teaching among educators. Each simulation has to be evaluated in the context of its unique educational goals. However, developers have a strong common denominator: a belief in the potential of visualization and the determination to improve the teaching process. This common denominator has motivated us to share our experiences, in spite of the fact that we live in far corners of the world and have never met in person. We plan to continue this collaboration with the aim of designing better evaluation methods and formulating appropriate visualization concepts. An appendix is included to provide a more detailed description of each simulator.

APPENDIX A.1. EASYCPU SIMULATOR

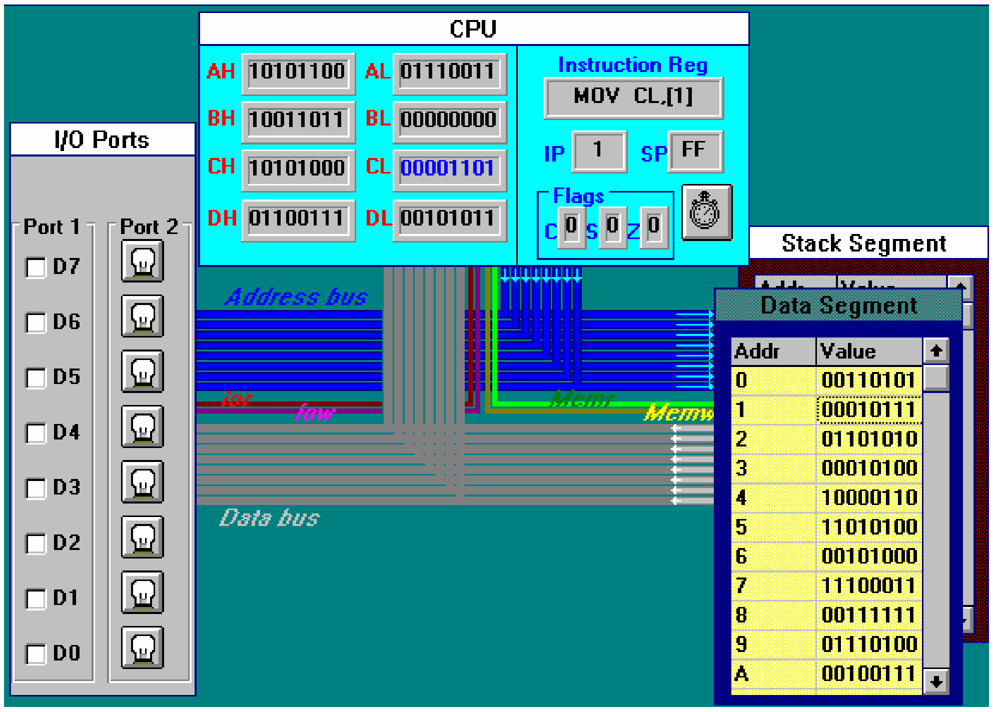
EasyCPU simulates a simplified model of the Intel 80x86 family processor. The EasyCPU instruction set is a subset of the group of instructions used by 80x86 processors. In EasyCPU, the CPU has 8 byte-registers. The CPU is connected to the three-segment memory with an 8-bit data and address bus. When EasyCPU is first started, the main window of the Basic Mode is opened as shown in Figure A.1.

A.1.1 Basic Mode Operation

The main window of the Basic Mode includes the main menu and the work area. In the work area, a simple computer model is displayed, including the following units:

- CPU: central processing unit
- memory: data and stack segments
- input/ouput ports (I/O)

The CPU subwindow displays the main 8-bits registers: AL, AH, BL, BH, CL, CH, DL, DH, flags (Z, S, and C) as well as the IP (Instruction Pointer), IR (Instruction Register), and SP (Stack Pointer). The registers and flags originally contain binary data, but students can select another data format and can also change the contents of a register.



AH	10101100	AL	01110011
BH	10011011	BL	00000000
CH	10101000	CL	00001101
DH	01100111	DL	00101011

Instruction Reg: MOV CL,[1]

IP 1 SP FF

Flags: C 0 S 0 Z 0

Addr	Value
0	00110101
1	00010111
2	01101010
3	00010100
4	10000110
5	11010100
6	00101000
7	11100011
8	00111111
9	01110100
A	00100111

Fig. A.1. EasyCPU Basic Mode window animated during execution of MOV [1], CL.

The input/output unit includes:

- *Input Port*: an array of eight switches, which can be opened or closed by pressing on the depressed part of the button. Reading the input is done by an input instruction in assembly language.
- *Output Port*: an array of eight LEDs. They can be turned on or off by executing an output instruction in assembly language.

The memory unit in Basic Mode contains two segments, as shown in Figure A.1:

1. stack segment: 256 bytes
2. data segment: 256 bytes

It is possible to scroll through the bytes stack by using the scroll arrows and button on the right part of the window. Each byte in a segment has an address (Addr) written in hexadecimal on the left of the byte. In the data segment, students can select another format and can change the contents of a register.

The connections between the three components (CPU, I/O, and memory) can be viewed through the following buses:

- a bidirectional data bus with eight lines, for moving data bytes from the CPU to the memory unit and I/O and vice-versa;
- a unidirectional address bus with eight lines from the CPU to the memory and I/O for operations of moving bytes from place to place; and

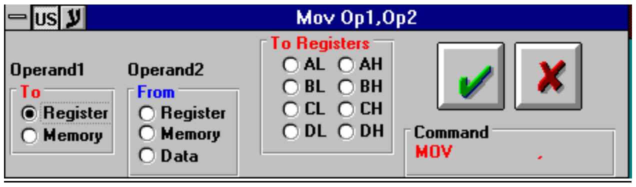
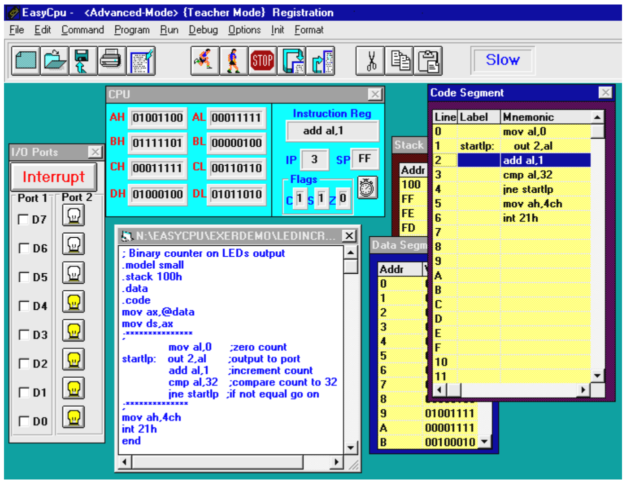


Fig. A.2. Dialog box template for MOV instruction.


EasyCpu - <Advanced-Mode> {Teacher Mode} Registration
File Edit Command Program Run Debug Options Init Format
Slow
CPU
I/O Ports
Interrupt
Port 1 Port 2
D7
D6
D5
D4
D3
D2
D1
D0
Stack
Data Segm

AH 01001100 AL 00011111
BH 01111101 BL 00000100
CH 00011111 CL 00110110
DH 01000100 DL 01011010

Instruction Reg: add al,1
IP 3 SP FF
Flags: C 1 S 1 Z 0

```
; Binary counter on LEDs output
.model small
.stack 100h
.data
.code
mov ax,@data
mov ds,ax
;***************
          mov al,0     ;zero count
startlp:  out 2,al     ;output to port
          add al,1     ;increment count
          cmp al,32    ;compare count to 32
          jne startlp  ;if not equal go on
;**************
mov ah,4ch
int 21h
end
```

Code Segment

Line	Label	Mnemonic
0		mov al,0
1	startlp:	out 2,al
2		add al,1
3		cmp al,32
4		jne startlp
5		mov ah,4ch
6		int 21h
7		
8		
9		
A		
B		
C		
D		
E		
F		
10		
11		

Stack: Addr 100, FF, FE, FD

Data Segment

Addr	V
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	01001111
A	00001111
B	00100010

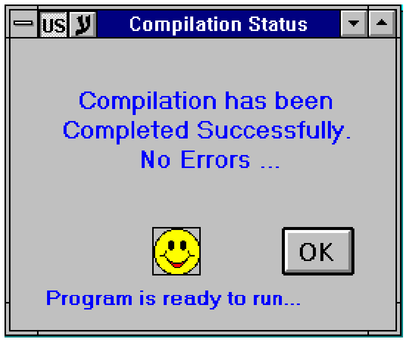
Fig. A.3. EasyCPU program in advanced mode.

- a control bus containing four unidirectional lines: MemR and MemW for control of reading and writing to memory. IOR and IOW for control of reading and writing to the I/O unit.

The Basic Mode is focused on helping novice students learn the syntax of individual instructions. Students can choose to simulate an instruction from the following subsets: data transfers, logic operations, arithmetic operations, I/O, stack, and control. Then a dialog box opens to help students build the instructions and learn the syntax.

In Figure A.2, the dialog box required to build a “MOV Op1, OP2” instruction is shown displaying all the alternatives for copying two operands: Op2 to Op1. The “MOV” instructions have two operands. This copies data from place to place. There are 92 different types of MOV-like instructions, depending on operands, types, and addressing modes. To build the required instructions, students have to specify the source and target operand types (register, memory, or data), the addressing mode, and the registers to use. The dialog box displays successive templates to guide the students and provide the available alternatives according to previous student choices. The following dialog box displays one of the templates needed to choose the target register for Opd1.

When the building of the instruction is complete, an animation of the instruction execution is displayed on the screen. The execution is a sequence of small operations of



Compilation has been Completed Successfully. No Errors ...

Program is ready to run...

Fig. A.4. EasyCPU assembler success window.

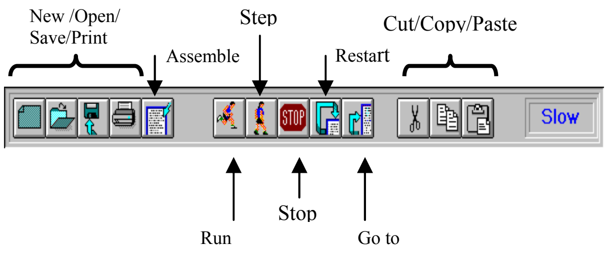


Fig. A.5. EasyCPU programming toolbar.

copying or processing data. For example: during the execution of the "MOV [1], CL" instruction, the CPU writes the content of register CL to the memory data byte addressed by 1, as we can see in Figure A.1. Students have control over the rate of execution through the processor clock. It enables students to control and observe the timing of the dataflow among the CPU, the memory, and the I/O ports.

Students can choose the format of the data displayed. The data in the registers and memory can be presented in different formats using the Format menu: Binary, Hexadecimal, Decimal (also a signed decimal mode). Students can change the contents of each cell by clicking on. A yellow box appears in which a new value can be edited in any of the three modes: binary, hexadecimal, or decimal (by adding the letters b, h, or d, respectively, after the number).

A.1.2 Advanced Mode

The Advanced Mode is dedicated to students with prior knowledge of assembly language programming. The main window in Advanced Mode (see Figure A.3) includes: the main menu, a toolbar offering a graphic menu, and a work area. In the work area, the following components of the simulated model are displayed: CPU; memory (data, stack, and code segments); and input/output ports.

EasyCPU provides students with a development environment for creating its own programs. In this mode, students are able to edit a new program or open examples to assemble, link, or execute. If the assembly syntax is correct, a message is displayed, as shown is Figure A.4.

Table B.1. Little Man Computer Instruction Set

Opcode	Description	Mnemonic
1	LOAD contents of mailbox address into calculator	LDA XX
2	STORE contents of calculator into mailbox address	STA XX
3	ADD contents of mailbox address to calculator	ADD XX
4	SUBtract mailbox address contents from calculator	SUB XX
500	INPUT value from inbox into calculator	IN
600	OUTPUT value from calculator into outbox	OUT
700	HALT - LMC stops (coffee break)	HLT
	SKIP	
800	SKN - skip next line if calculator value is negative	SKN
801	SKZ - skip next line if calculator value is zero	SKZ
802	SKP - skip next line if calculator is positive	SKP
9	JUMP – go to address	JMP XX

Note: XX represents a two-digit mailbox address

To verify the correctness of their programs, students need, through the debugging tools, to keep track of the execution of the programs and the data processing (see the toolbar in Figure A.5).

Step-by-step execution enables students to follow the sequential execution of the instructions, as they are dynamically animated. At any phase of the running process, students are able to change data in CPU and memory. Figure A.3 shows a screenshot of a program sample running in Advanced Mode. The program outputs a binary count to port 2. In conclusion, the EasyCPU development environment was adapted to provide tools to fulfill students' cognitive needs, in parallel with the visualization of the processes taking place within a computer.

APPENDIX B.1. LITTLE MAN COMPUTER SIMULATOR

The Internet Explorer is preferred to run this application. Double-click on default.htm. The application has built-in help menus and indicates a "starting point."

Table B.1 defines the LMC instruction set; the nine instructions below are sufficient to perform any computer program.

When working with students, we emphasize two things about the LMC instruction set:

1. Although any program can theoretically be implemented in LMC assembly source code, the actual implementation may be extremely complex.
2. Expanded instruction sets on modern computers do not change the fundamental operations of the computer!

As in most assembly language instruction sets, it is difficult to write useful functions in a small number of source code lines. We say this to emphasize the relationship of high-level language to assembly language in terms of programmability, user friendliness, and efficiency. Computer instruction sets on real computers are more sophisticated and flexible, providing additional instructions that make programming easier. However, these additional instructions do not change the fundamental operations of the computer. Students learn the fundamental concept that a computer is nothing more than a machine capable of performing simple instructions at high speed. Later in the course, we discuss instruction set variations as the major conceptual difference between processor types, and then show specific examples of a different code that performs the same task.

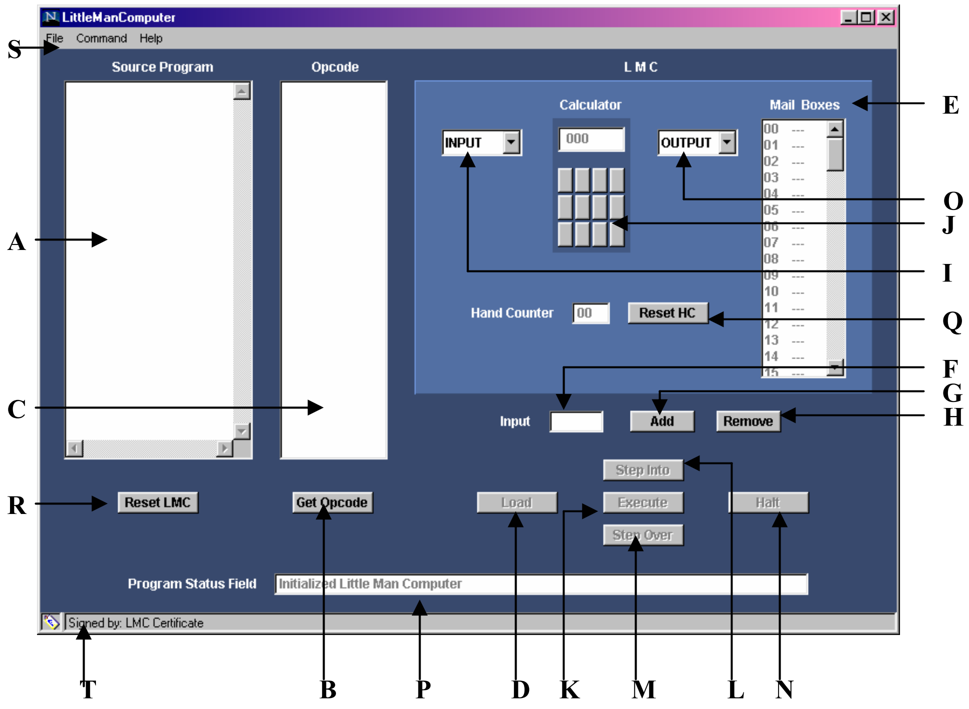


Fig. B.1. Little Man Computer GUI with fields identified.

A: editor to write source program or load existing file
B: assembler, converts mnemonics to machine code
C: if syntax is correct the assembled machine code is here
D: loader, loads machine code into mailboxes (memory)
E: mailboxes, 100 from 00 to 99
F: input box, input data by typing here
G: click here to load input data to input box
H: click here to remove highlighted data from input box
I: input Box, contains data entered from F and G
J: calculator, user cannot interact with calculator buttons
K: execution: burst mode
L: execution: step-into mode
M: execution: step-over mode
N: halt, safety to stop burst mode execution
O: output box
P: program status field, contains flags and error messages
Q: reset location counter (or hand counter)
R: clear, clears all fields except source program in A
S: menu bar for various operations not shown here, including loading and saving of files, setting breakpoints, help, etc.
T: security,, indicates certificate and author certificate

The hardware and software requirements for LMC are generally available on any PC running Windows. LMC works on any Java-enabled Web browser, although it was only tested with Microsoft Internet Explorer and Netscape Navigator. LMC was developed

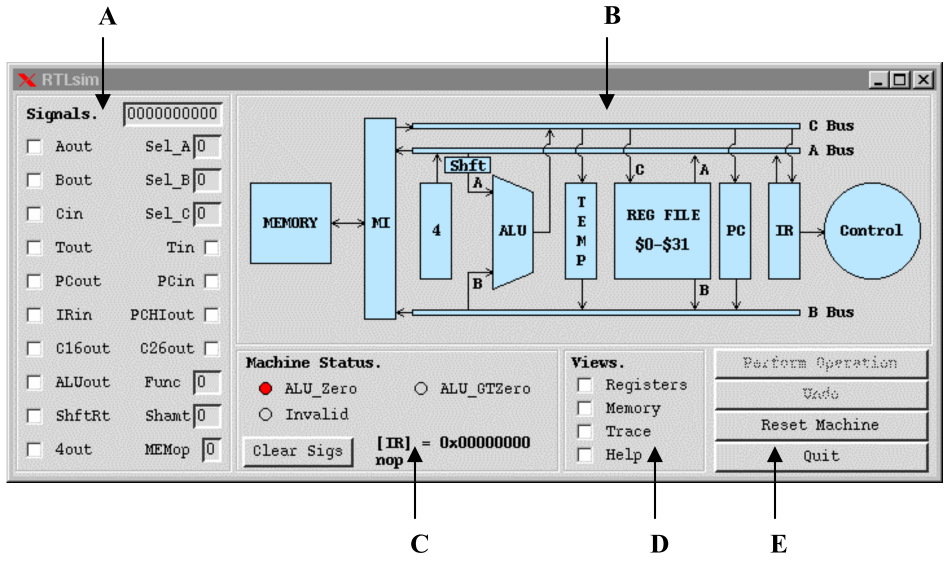


Fig. C.1. RTLSim main window

A: signals sub-window
B: data path sub-window
C: status sub-window
D: views sub-window
E: buttons sun-window

using Java JDK1.2 and is embedded in an applet to provide ubiquitous access to users over the Internet without the need for the local installation of JDK1.2. Another advantage of this approach is that the applet is loaded in a separate window, allowing users to run the application while viewing the HTML documentation in another window. To enable control of the LMC simulation, multiple threads were implemented: one thread for executing the program and a second thread listening for a user interrupt to halt the program.

Applets have restricted access to computer file systems on most browsers. Thus, to enable the saving and loading of source code, users have to install a signed certificate. Signed applets have full functionality, while unsigned applets may still execute in a restrictive "sandbox" mode, preventing hard disk access and Web site connections. Signing certificates for Microsoft Internet Explorer and Netscape Navigator are handled in different ways: complete documentation for type of each browser is provided online with step-by-step instructions.

The following are the general principles used to design the user interface to LMC:

- an intuitive graphical user interface (GUI);
- online help should be available at all times;
- standardized error messages with explanations is provided;
- white user-modifiable fields, gray unchangeable fields;
- equally-sized buttons that are enabled only when required;

- no more than five colors on the screen; and
- screen resolution must be 800 X 600 (commonly available).

Our intent is to simultaneously provide students with a visualization of all components within a computer architecture, an ability to observe the fetch-execute cycle during program execution, and a highly interactive problem-solving environment with flexible input/output demands. Figure B.1 is a screen shot of LMC. 0.

APPENDIX C.1. RTLSIM SIMULATOR

RTLSim uses a GUI interface and, when it is first started, a single window is opened as shown in Figure C.1. The main function of this window is to enable the user to set each of the signals for the current control set in the simulation. For the choice boxes (such as `IRin` and `ALUout`), a tick in a box means the signal is activated for the current control step, otherwise the signal is not activated. There are also six text boxes (e.g., `Sel_A`, `Func,` and `Shamt`) where decimal values can be entered. In addition, there is a clear button in the status subwindow that can be used to clear all of the signals in the signal's subwindow.

As a user selects and deselects signals in the signal's subwindow, the components and buses in this window change color, indicating that the component or bus is involved in the current control step. A bus is green for a valid data transfer, whereas it turns red for an invalid transfer (i.e., two components trying to output a value onto the bus at the same time). In the case of the components, yellow means the component is involved in the transfer and blue means the component is not involved. The main window contains information that the control unit can use to make decisions about the next control step.

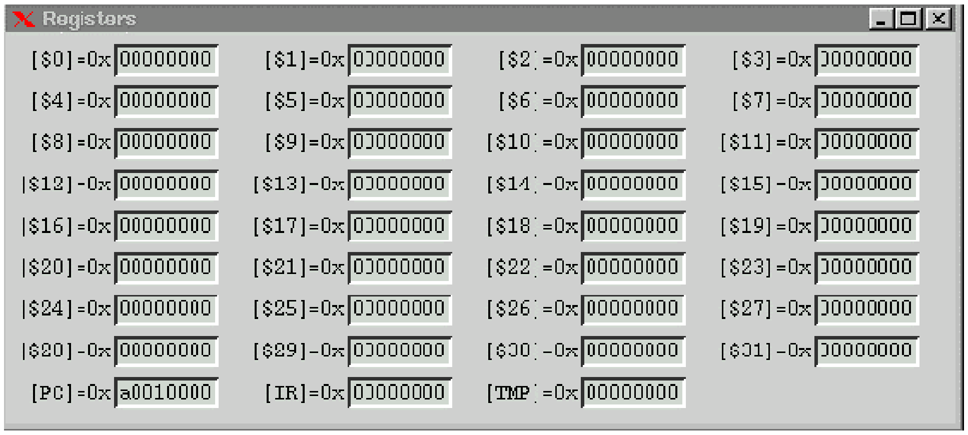
The read only signals shown in the machine status subwindow are as follows:

- `ALU_Zero:` If this signal is active (the circle to the left of the signal name is red), then it means the output of the ALU is equal to zero.
- `ALU_GTZero:` If this signal is active, then it means that the current result being output by the CPU is greater that zero.
- `Invalid:` If this signal is active, then it means that two components are trying to output to the bus at the same time, i.e., the current control step is trying to perform an invalid operation.
- `[IR]:` Shows the contents of the instruction register. The MIPs instruction directly underneath this is a disassembled version of the contents of the instruction register (current instruction being executed).

In the machine status subwindow there is also a `Clear Signals` button that can be used to clear all of the signals in the signals subwindow.

The view subwindow allows us to open other windows to see the status of different parts of the architecture. If a check box has a tick in it, then it means that the window is open. The other windows that can opened are the following:

- `Registers` window allows users to examine and edit the contents of all registers in the simulated architecture.
- `Memory` window allows the user to view and edit the contents of the simulated machines memory.
- `Trace` window allows users to see the trace that is being generated while control steps are being performed. Functions in this window also allow the user to



Registers

```
[$0]=0x00000000    [$1]=0x00000000    [$2]=0x00000000    [$3]=0x00000000
[$4]=0x00000000    [$5]=0x00000000    [$6]=0x00000000    [$7]=0x00000000
[$8]=0x00000000    [$9]=0x00000000    [$10]=0x00000000   [$11]=0x00000000
[$12]=0x00000000   [$13]=0x00000000   [$14]=0x00000000   [$15]=0x00000000
[$16]=0x00000000   [$17]=0x00000000   [$18]=0x00000000   [$19]=0x00000000
[$20]=0x00000000   [$21]=0x00000000   [$22]=0x00000000   [$23]=0x00000000
[$24]=0x00000000   [$25]=0x00000000   [$26]=0x00000000   [$27]=0x00000000
[$28]=0x00000000   [$29]=0x00000000   [$30]=0x00000000   [$31]=0x00000000
[PC]=0xa0010000    [IR]=0x00000000    [TMP]=0x00000000
```

Fig. C.2. RTLSim registers window.

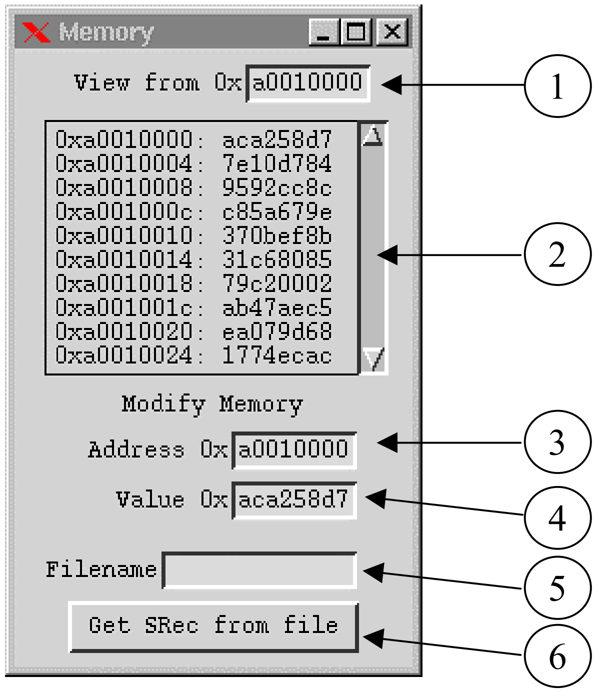
save these traces to disk and load previously recorded traces so that they can be replayed in the simulator.

- `Help` window provides an elementary help facility. At this stage, the only real help available is a listing for the encoding of the memory and ALU functions in the signals window.
- `Macros` window provides a simple macro facility that allows single control steps (that are going to be reused) to be recorded so that they can be reused at a later time.

This buttons subwindow contains the following button selections:

- `Perform Operation:` When this button is clicked, the current set of signals defined in the Signals window is taken and used to perform a control step. As a result of executing a control step, the necessary values will be updated in the register and Memory windows. Also, a control word encoding the operation performed in the control step is added to the end of the list in the Trace window.
- `Undo:` This button allows the last control step(s) to be undone. Also, if we are replaying a trace in the trace window, the `undo` button can be used to move to previous control steps.
- `Reset: Machine:` Clicking on this button resets the machine and clears the contents of all the registers except the program counter, which is initially set to `0xa0010000`. Also note that the contents of the Trace window and Memory windows are cleared.
- `Quit:` Clicking on this button causes the RTLSim program to finish.

A screen dump of the registers window is shown in Figure C.2. This window displays the current contents of each of the registers in the simulated machine. The contents of this window are updated at the end of every control step. It is also possible to change the values of these registers manually by clicking on and editing the register value we want to change. Note that any changes that we make to register values in this window are not recorded in the `Trace` window.



```
0xa0010000: aca258d7
0xa0010004: 7e10d784
0xa0010008: 9592cc8c
0xa001000c: c85a679e
0xa0010010: 370bef8b
0xa0010014: 31c68085
0xa0010018: 79c20002
0xa001001c: ab47aec5
0xa0010020: ea079d68
0xa0010024: 1774ecac
```

Fig. C.3. RTLSim memory window.

The memory window can be used to display and change the contents of the simulated machines' memory. As seen in Figure C.3, the memory window is made up of a number of components. The functionality of each of the components follows:

1. `View from:` This editable text box specifies the first memory address whose contents are displayed in the window directly below it. Editing this text box will cause the starting address in the window to change to its new value when we click anywhere in the window, apart from the text box.
2. This part of the window shows a range of memory addresses (on the left-hand side) and the contents of those memory locations. Note that the contents of memory cannot be edited in this window.
3. `Address:` This text box allows the address of a memory location to be entered so that the contents of the memory location can be changed in the text box directly below it.
4. `Value:` Entering a value in this text box will cause the contents of the memory location, specified in the text box directly above, to be changed to the value in the text box when the mouse is clicked outside the text box.
5. `Filename:` The text box allows the name of an `srecord` file to be entered, so that the `srecord` file can be loaded into the simulated machines memory when `Get SRec` from file button is clicked on.
6. `Get SRec from file:` When this file is clicked on, the `srecord` file specified in the Filename text box will be loaded into the simulated machine's

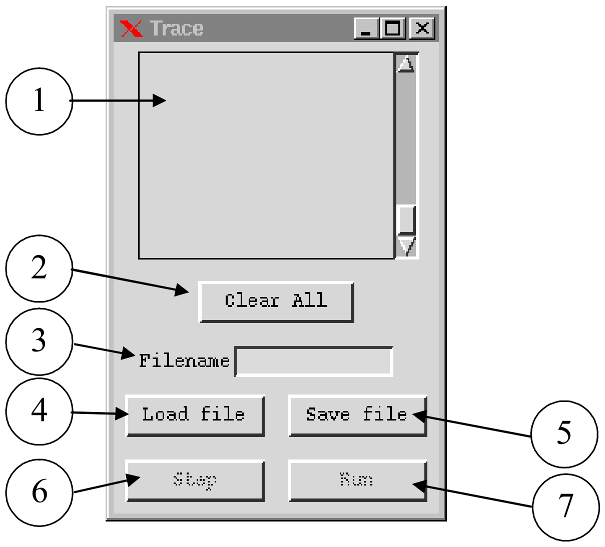
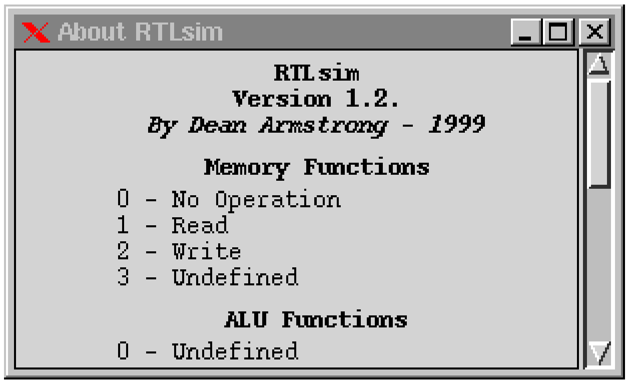


Fig. C.4. RTLSim trace window.

memory. If the `srecord` file specifies an initial address (set by the `asm` command if any of the files used to generate the `srecord` file contain the global label main), the program counter register (PC) will be changed to the initial address.

The trace window is used to record and play back traces of control steps performed by the RTLSim simulator; its snapshot is shown in Figure C.4. The trace window contains a number of components:

1. This box shows a hexadecimal coded version of the control steps performed so far if the simulator is being used to record a new trace, or it shows a previously recorded trace.
2. `Clear All`: Clicking on this button clears the contents of the trace window so that a fresh trace can be recorded.
3. `Filename`: This text box allows a filename to be entered so that a trace can be loaded from disk using the `Load File` button or saved to disk using the `Save File` button.
4. `Load File`: Loads the file specified in the `Filename` textbox into the trace list at the top of the window. This trace can then be replayed through the simulator using the `step` and `run` buttons at the bottom of the window.
5. `Save File`: Clicking on this button will save the current trace list at the top of the window into the file specified in the `filename` text box. The file created can then be reloaded into the simulator or used as input to the Web-based marking program.

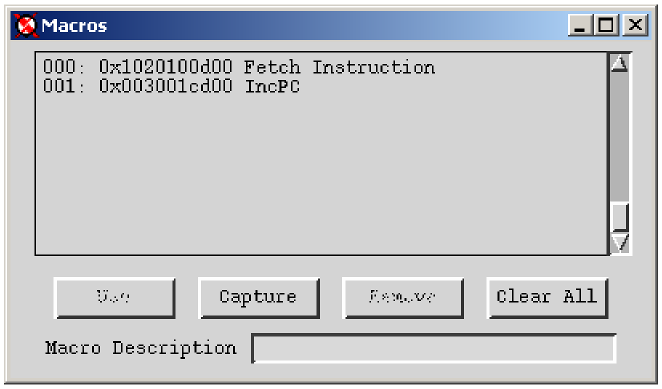


RTLsim
Version 1.2.
By Dean Armstrong - 1999

Memory Functions
0 - No Operation
1 - Read
2 - Write
3 - Undefined

ALU Functions
0 - Undefined

Fig. C.5. RTLSim help window.



```
000: 0x1020100d00 Fetch Instruction
001: 0x003001cd00 IncPC
```

Fig. C.6. RTLSim macro window.

6. `Step:` Unless a trace file has been loaded from disk, this button will normally be grayed out so that it cannot be clicked on. However, when a trace file has been loaded from disk, clicking on this button will cause the step highlighted in the trace list at the top of the window to be executed. The highlighted instruction will be advanced to the next one in the list so the process can be repeated. After the last instruction has been executed, the mode of the simulator will change so that we can perform new control steps that will be added to the end of the list.
   Also, while single- stepping through control steps in the trace list, click on the `undo` button so that the last control step(s) can be undone.
7. `Run:` Once a trace file has been loaded, the whole trace can be replayed by clicking on this button. Once a trace has been replayed, it is possible to add new control steps to its end by executing control steps in the normal fashion.

The help system for the simulator is sparse at present. However, the help window does contain a list of the decimal codes for the memory and ALU signals in the signals window; Figure C.5 shows a screen snapshot of the help window.

RTLSim contains a simple macro facility that allows single control steps to be recorded so that they can be replayed at a later time. For example, the control step can be used to fetch an instruction multiple times. Rather than having to set up the signals every time it is to be used, the macro window can be used to record the signals used in the control step the first time it is used and then, when it is required again, the macro window can be used to replay it. When we want to record a control step, we set up all the signals required for the control step in the signals subwindow of the main window and then click on the capture button in the macro window. This should cause a new line to be added to the macros window (see Figure C.6). The macro can then be named by typing the macro name into the "Macro Description" text box of the macros window. At a later time, when the macro is to be reused, click on the required macro and then click on the "use" button. This should cause all of the signals defined in the macro to be set up in the signal subwindow of the main simulation window.

REFERENCES

CASSEL, L., KUMAR, D., BOLDING, K., DAVIESW, J., HOLLIDAY, M., IMPAGLIAZZOO, J., WOLFFE, G., AND YURCIK, W. 2001. Distributed expertise for teaching computer organization and architecture. *ACM SIGCSE Bull. 33,* 2, 111-126.

HANRAHAN, P., LEVOY, M., AND ROSENBLUM, M. 1996. Visualizing computer systems. http://graphics.stanford.edu/courses/cs348c-96-fall/syllabus.html.

FAYZULLIN, M. 2002. *How to Write a Computer Emulator*. http://www.komkon.org/fms/EMUL8/HOWTO.html.

FISHWICK, P. 2000. Modeling the world. *IEEE Potentials* (March), 6-10.

FOLEY, J. 1998. Foreword. In *Software Visualization: Programming as a Multimedia Experience*, J. STASKO, J. DOMINGUE, M.H. BROWN, AND B.A. PRICE, Eds. MIT Press, Cambridge, MA, xii-xiii.

LOUI, M.C. 1988. The case for assembly language. *IEEE Trans. Education 31,* 3, 160-164.

PEARSON, M.W., MCGREGOR, A.J., AND HOLMES, G. 1999. Teaching computer systems to majors: A MIPS based solution. *IEEE Comput. Soc. Architecture Tech. Committee Newsl*., 22-24.

PETRE, M., BLACKWELL, A., AND GREEN, T. 1998. Cognitive questions in software visualization In *Software Visualization: Programming as a Multimedia Experience*, J. STASKO, J. DOMINGUE, M.H. BROWN, AND B.A. PRICE, Eds. MIT Press, Cambridge, MA, 453-480.

SIMEONOVV, S., SCHEINDER, M. 1995. MISM: An improved microcode simulator. *ACM SIGCSE Bull.27,* 2, 13-17.

STASKO, J. 1998. Empirically assessing algorithm animations as learning aids In *Software Visualization: Programming as a Multimedia Experience*, J. STASKO, J. DOMINGUE, M.H. BROWN, AND B.A. PRICE, Eds. MIT Press, Cambridge, MA, 419-438.

YURCIK, W., VILA, J., AND BRUMBAUGH, L. 2000. An interactive Web-based Simulation of a general computer architecture, In *Proceedings of the IEEE International Conference on Engineering and Computer Education*.

Received November 2001; revised January 2002; accepted February 2002