USING COMPUTER ANIMATIONS IN THE CLASSROOM

George F. Riley
Georgia Institute of Technology, Atlanta, GA. U.S.A.

Keywords: Animations, Classroom teaching, Teaching programming.

Abstract: We present several animation tools that we have used in the classroom to help reinforce some fundamental concepts of computer architecture, computer programming, and computer networking. When presenting such concepts to undergraduate students, the concept of concurrency has been particularly difficult to explain and demonstrate. When two more things are happening at the same time, student often struggle to grasp what this actually means. Additionally, when teaching introductory programming classes, we believe it is important for students to understand how their program is converted into a larger number of simpler steps resulting in assembly language or machine code instructions. Attempts to show these concepts using traditional chalkboard methods or static viewgraphs have not worked well. To help present these concepts in a more understandable way, we have created several computer animations that we have shown to students in the classroom to illustrate visually these fundamental concepts. Further, our animation programs can create movie output files that students can download and view as often as desired.

1 INTRODUCTION

The use of computers and computation has become fundamental to all engineering disciplines. All undergraduate engineering students are required to take *CS1371 - Computing for Engineers* and all undergraduate Electrical and Computer Engineering (ECE) students must take *CS1372 - Program Design for Engineers*. The CS1371 class introduces students to basic programming concepts using the *Matlab* programming language, and gives a solid understanding of simple concepts such as variable declarations, arrays, loops, and subroutines. The second class, CS1372, uses the *C* programming language to introduce more advanced concepts such as memory addressing, pointers, structures, and recursion. We typically teach several thousand students annually in CS1371, and several hundred annually in CS1372. Recently, we decided to introduce the concepts of concurrency and multi–programming in one section of the 1372 class, since multi-core architectures have become pervasive and are expected to continue for the foreseeable future. Additionally, we concluded that explaining "how the computer works" at a lower level (e.g. assembly language versus high–level languages) would be of benefit, particularly when discussing race conditions and interlocking.

When preparing lecture materials for these new topics, we realized that we needed more than just a set of static handout slides, since we need to illustrate the concepts of concurrency (things happening at the same time) and sequential actions (such as assembly language instructions executing one after the other). Thus we undertook to develop a set of computer programs that both simulate the action of some fundamental process, and animate it visually to help student "see" what is happening. Further, we designed these animation programs to be flexible enough to create animations of specific actions based in either command line arguments or data input files. For example, our CPU animator actually reads in an assembly language program (in human readable format) and then executes the specified program. As another example, our animation of *parallel bubble sort* is driven by command line arguments specifying the size of the array to be sorted, and the number of CPU's assigned to sort the array. Finally, our animation programs output a set of individual time–stepped images that can be combined into a single movie file in one of several video formats.

We will present some details regarding several different animation programs we created in subsequent sections. In some cases, individual snapshots of the animation will be shown here with discussion of the animations used.

F. Riley G. (2010).
USING COMPUTER ANIMATIONS IN THE CLASSROOM.
In *Proceedings of the 2nd International Conference on Computer Supported Education*, pages 473-476
DOI: 10.5220/0002806404730476

2 RELATED WORK

The use of simulations and animations to increase understanding of fundamental principles is of course new new and has been used for decades. As far back as the 1970's the Control Data Corporation *PLATO* system (Control Data Corporation and PLATO Learning Inc., 2009) made extensive use of graphics, animation, and simulation to help teach undergraduate and graduate students. However, the use of the PLATO in educational settings was primarily used for "drills" with customized feedback to the users, rather than being used in the classroom. One notable exception was the *CDC Assembly Language Simulator* for PLATO, that allowed users to write, execute, and debug programs both in the CDC CPU language and the CDC PPU language.

More recently, Jerding et. al (Jerding et al., 1997) have discussed using visualization to observe the flow of a given program, primarily to assist in debugging and performance analysis. Earlier, Stasko et. al (Stasko, 1991) designed the *Tango* system, that provides a simple and easy-to-use interface for instructors to create animations of algorithm execution. The *Tango* system produced animations similar to our *parallel bubble sort* animation discussed later.

Carothers et. al (Carothers et al., 1997) created a visualization environment that we used to observe the execution of a parallel and distributed discrete event simulation environment. Their *PvaniM* system was primarily intended to allow a user to observe possible performance bottlenecks in an optimistic distributed simulation, and to compare various middleware approaches to obtain optimal performance.

The commonly used *SPIM* (Larus, 2009) MIPS simulation tool is widely used to help students design, implement, and debug assembly language programs. While the *SPIM* environment does have a comprehensive and friendly graphical user interface, it does not animate the actual execution steps within an instruction (such as computing the effective memory address and fetching operands from memory).

The preceding is only a small sampling of a large number of computer–based simulations that have been frequently used to provide detailed insight into computer and computer program operation. We have developed simulations and animations that are specifically designed to help students understand fundamental concepts in computer architecture, algorithms, and networking.

3 ANIMATION DESCRIPTIONS

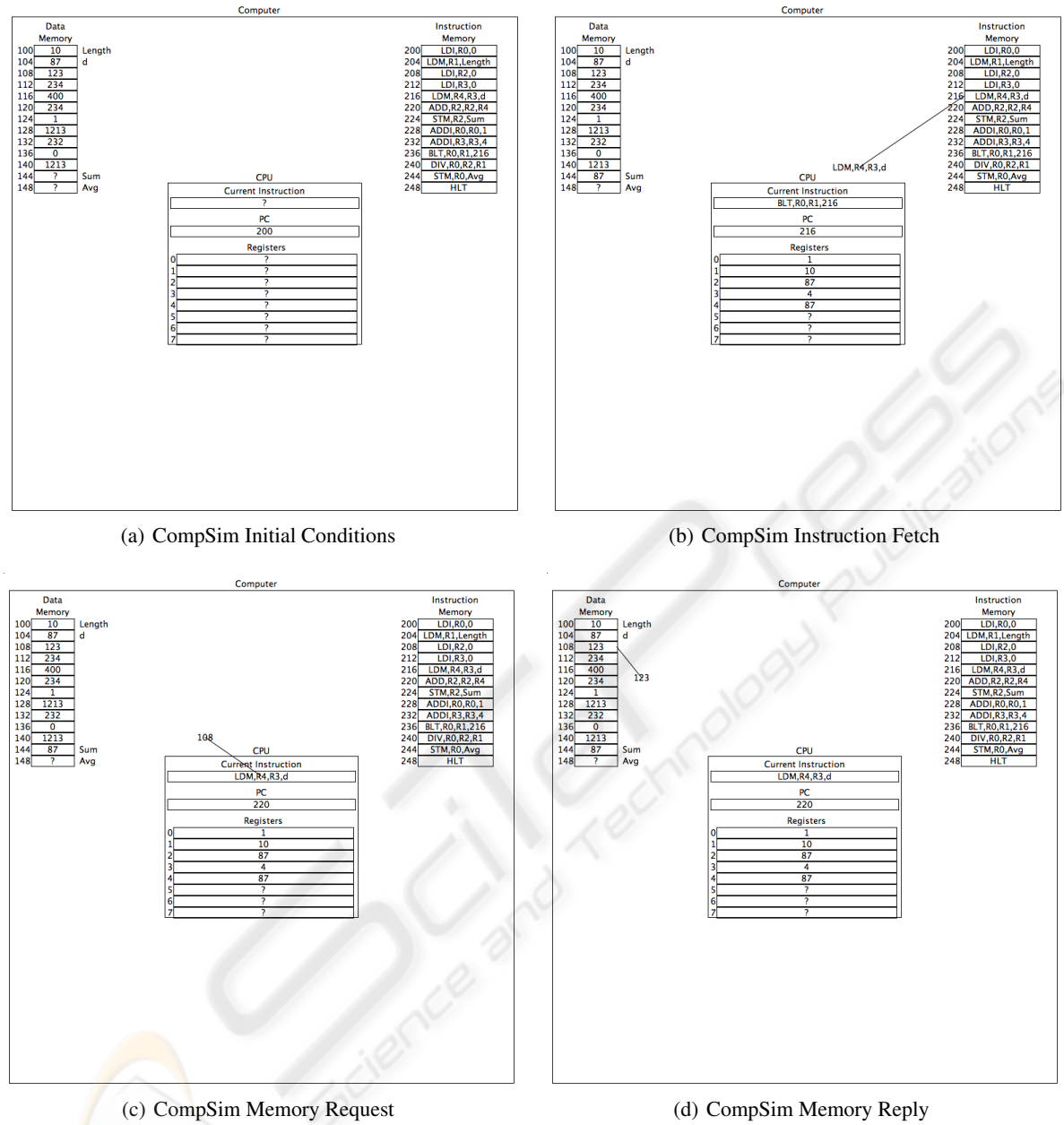
This section describes in detail several of the simulation and animation tools we have created recently. We have focused our design and features specifically for use by classroom instructors during the lectures, to give students additional visual clues for the individual concept being presented.

3.1 CPU Animations

Given the current popularity of multi–core architectures, such as the Intel Core 2 Duo, we decided that an introduction to parallel computation was needed very early in the Electrical Engineering curriculum. In order to provide a deep understanding of the issues in parallel programs, such as race conditions, interlocks, deadlocks and livelocks, we decided to introduce students to lower–level assembly language in the first weeks of CS1372. Our objective was not to teach the students any individual assembly language such as Intel i386, but rather teach the fundamental concepts of memory accesses, register to register operations, and branching instructions. Without this detailed understanding of what is happening "under the covers", it is hard for students to realize the consequences of seemingly innocuous instructions like "`i = i + 1`" in a multi–processing environment. In order to illustrate these assembly language principles, we developed the `CompSim` program.

Since we were not interested in any one particular machine language, we decided the simplest and easiest approach was to define our own assembly language. We included the usual arithmetic instructions, load and store instructions (with both "load from immediate" and "load from memory"), and the usual conditional and unconditional branching instructions. Even though we only supported 20 instructions, they are sufficiently capable to design and implement any arbitrarily complex program. The key feature of `CompSim` was of course the animation output.

First we will discuss and show the *Single CPU* simulation supported by `CompSim`. A snapshot of the initial state of `CompSim` is shown in figure 1(a). Shown clearly are the CPU components (Instruction Register, Program Counter, and the eight Registers), the data memory, and the instruction memory. The question mark characters shown in several of the registers indicates an *uninitialized* value. This particular "program" calculates the sum and average of an array of numbers, with the length of the array found in a specific memory location. Next, figure 1(b) shows the instruction fetch cycle, which reads the next instruction at the address specified by the `PC` register,



(a) CompSim Initial Conditions

(b) CompSim Instruction Fetch

(c) CompSim Memory Request

(d) CompSim Memory Reply

Figure 1: CompSim Single CPU Animations.

address 216 in this case. Figure 1(c) shows the animation of the execution of the `LDM,R4,R3,d` instruction. This particular instruction reads the memory location specified by the address `d` (104 in this example) plus the value found in register `R3` (4 in this case). Thus the memory address 108 is requested. Not shown of course is the animation, which animates the value 108 leaving the CPU and arriving at the data memory unit. Finally, figure 1(d) shows the value 123 being read from memory and being loaded in register `R4`. The `CompSim` animation will continue reading instructions and executing the instruction, including of course the looping instructions, until the program terminates with the `HLT` instruction at address 248.

As we mentioned, one of the primary reasons for using `CompSim` in the classroom was to illustrate the concept of "things happening at the same time" that is needed to understand and create parallel programs. To this end, the `CompSim` animation supports the simultaneous simulation of two CPUs. There is no techni-

cal reason why `CompSim` could not simulate and animate a larger number of processors, but the animation and screen real estate would be too cumbersome to be meaningful.

We have also use a similar approach to show the operation of a *multi-core* system, with two CPU's operating simultaneously and independently. Our animations show clearly that we have two distinct and separate Central Processing Units, each with the own and independent program counter (PC), Current Instruction register, and the set of eight general–purpose registers.

One particular animation shows the two CPUs executing independent programs with disjoint memory footprints for each. We have several other example programs, including a true *multi–threaded* program where both CPU's are executing from the same instruction memory and have potential access to the same data memory locations. We believe this helps with understanding the problems of race conditions and mutual exclusion that are fundamental to multi–core programming.

Another item of interest for the `CompSim` program is that it models the Intel addressing approach of individual byte addresses, even though virtually all memory accesses are on 4–byte boundaries. The memory addresses shown in `CompSim` advance by 4 bytes for each consecutive memory address, as is done in Intel architectures.

4 CONCLUSIONS

We have presented details of several animations that we used in a classroom environment when teaching introductory "C" programming to freshmen at Georgia Tech. These animations were created in an attempt to increase student's understanding of both the low–level fundamental operation of computer hardware, as well as understanding how algorithms can operate "in parallel" to produce more timely results. We used these animations in the classroom in a recent section of our CS1372 class. Further, we posted movie files of several of the animations for students to download and observe as needed.

Unfortunately we only have anecdotal evidence of the effectiveness of this approach. We intentionally did not pursue approval from our Institutional Review Board (IRB), nor did we compare performance of these students to a possible control group in other sections. The reason was simply that we were not confident that we could create the necessary software to produce the animations on a timely basis. Informal feedback from students indicated a good appreciation and understanding of the issues presented, but of course such evidence is by no means conclusive.

REFERENCES

Carothers, C. D., Topol, B., Fujimoto, R. M., Stasko, J. T., and Sunderam, V. (1997). Visualizing parallel simulations in network computing environments: a case study. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 110–117, Washington, DC, USA. IEEE Computer Society.

Control Data Corporation and PLATO Learning Inc. (2009). The PLATO system. http://www.plato.com.

Jerding, D. F., Stasko, J. T., and Ball, T. (1997). Visualizing interactions in program executions. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 360–370, New York, NY, USA. ACM.

Larus, J. (2009). A MIPS32 simualtor. Software on-line: http://pages.cs.wisc.edu/ larus/spim.html. Microsoft Research.

Stasko, J. T. (1991). Using direct manipulation to build algorithm animations by demonstration. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 307–314, New York, NY, USA. ACM.