A MULTILEVEL SIMULATOR AT THE REGISTER TRANSFER LEVEL FOR USE IN AN INTRODUCTORY MACHINE ORGANIZATION CLASS

by Dale Skrien
Colby College, Waterville, ME 04901 USA
djskrien@colby.edu
and
John Hosack
University of the South Pacific, Suva, Fiji Islands
hosack_j@usp.ac.nz

Abstract

This paper discusses a multilevel simulator and its use in the classroom. A new simulator is presented that allows the user to dynamically change the simulated hardware at run time. We also discuss an incremental series of projects that we have successfully used with the simulator. These projects give the student hands-on experience with the advantages and disadvantages of several architectures.

Introduction

We have used the hardware simulator STARTLE [1] for the past 8 years in the freshmen/sophomore-level machine organization class. Several interesting programming projects have been developed that use STARTLE.

Because of the ready availability of high-quality graphics displays on relatively low-cost micro-computers, such as Apple's Macintosh, a new simulator, based on the ideas in STARTLE, has been created that utilizes the graphical interface of the Macintosh to allow the students to spend more time understanding what the simulator is doing rather than understanding how to run the simulator.

Objective

In our machine organization and assembly language course, we want the students to learn about different architectures and the advantages and disadvantages of each. We do not wish to restrict ourselves to a particular machine, since that would narrow the student's experience and perceptions of machine architecture. In addition, the usage of a real machine would require the students to spend a lot of time learning peripheral details (of a machine they will probably never program at the machine level). For similar reasons, we also want to stress machine organization rather than assembly language programming, which is essentially symbolic machine language.

Meeting our objective

Because we can not afford to provide a lab full of hardware containing the different architectures, especially due to the time the students would have to spend learning how to run each machine, several years ago we searched for simulators. We encountered many of the same problems as Tangorra [2] with existing simulators. We were not happy with simulators that only simulated simple accumulator-based machines with a small fixed set of machine instructions and few addressing modes. We adopted STARTLE because of the way it allowed the users to design the architecture they wanted to simulate at the register transfer level, including the registers, flipflops, i/o channels, and stores, as well as all the micro-instructions and machine instructions. Using STARTLE, we could require the students to develop an architecture themselves and then create and run machine-language programs on it. To accomplish this, we developed a series of lab exercises to gradually

introduce the students to more and more complex machines.

Laboratory projects

The series of projects we have used should give an idea of how such a simulator might be used in any classroom.

In the first project, the students are given a machine description and a sample machine language program for a simple one-accumulator machine, called the Wombat1, with only direct addressing and no stack operations. We felt it was very important that the students be given the first machine so as to avoid the difficulty of creating a machine completely from scratch. In line with our gradual, incremental approach, the students are initially asked to modify the sample program for the Wombat1. After familiarizing themselves with the Wombat1, they are asked to write a program that reads in an arbitrarily long list of positive integers, followed by a negative integer sentinel, and then prints out the positive integers in reverse order. The fact that this can't be done on the Wombat1 without writing self-modifying code helps the students to quickly understand the need for either indirect addressing or the use of a stack.

The second project asks the students to upgrade the Wombat1 to the Wombat2, which contains a hardware stack and the operations of push and pop, which must be defined using micro-instructions. This project requires the students to begin to familiarize themselves with the multilevel concepts, working at the register-transfer level. They are then asked to redo the previous program on reversing a list using the enhanced machine.

The third project requires the students to add call and return machine

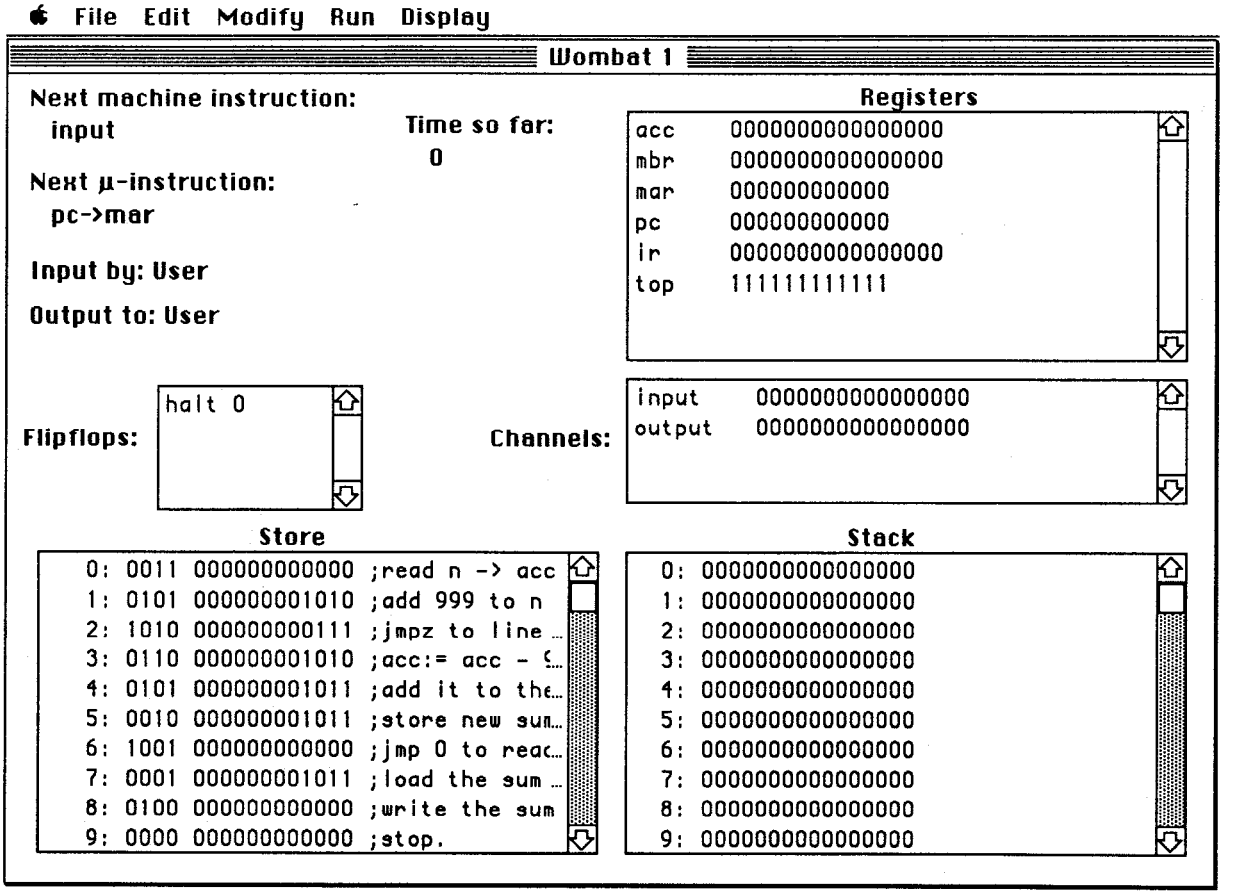

Figure 1.

instructions, creating the Wombat3, so that subprograms can be written and run.

They are then asked to write programs using subprograms, including recursive subprograms.

Finally, the fourth project asks the students to create a much more realistic machine, the Wombat4, with direct, indirect, immediate, and stack addressing modes. Programming assignments for this machine include the writing of subprograms that pass parameters by reference rather than just by value.

STARTLE only accepts programs written in machine language, and so does not allow for assembly language programming. However, the second author has written an assembler for the Wombat3 machine, and so the students are given the opportunity to write some simple assembly language programs, have them assembled into machine language, and then run these programs on the Wombat3 using STARTLE.

As students proceed through the project phases they also write a simple interpreter in Pascal for each Wombat. This interpreter reads the machine language file and stores the instructions in an array simulating memory. Then the basic machine cycle is followed: fetch instruction, increment program counter, decode instruction, and execute instruction.

The students gain a great deal from writing an interpreter. They improve their programming ability, especially in constructing well-documented and organized programs easily upgraded from one Wombat version to the next, and their implementation of the machine cycle provides reinforcement of the concepts.

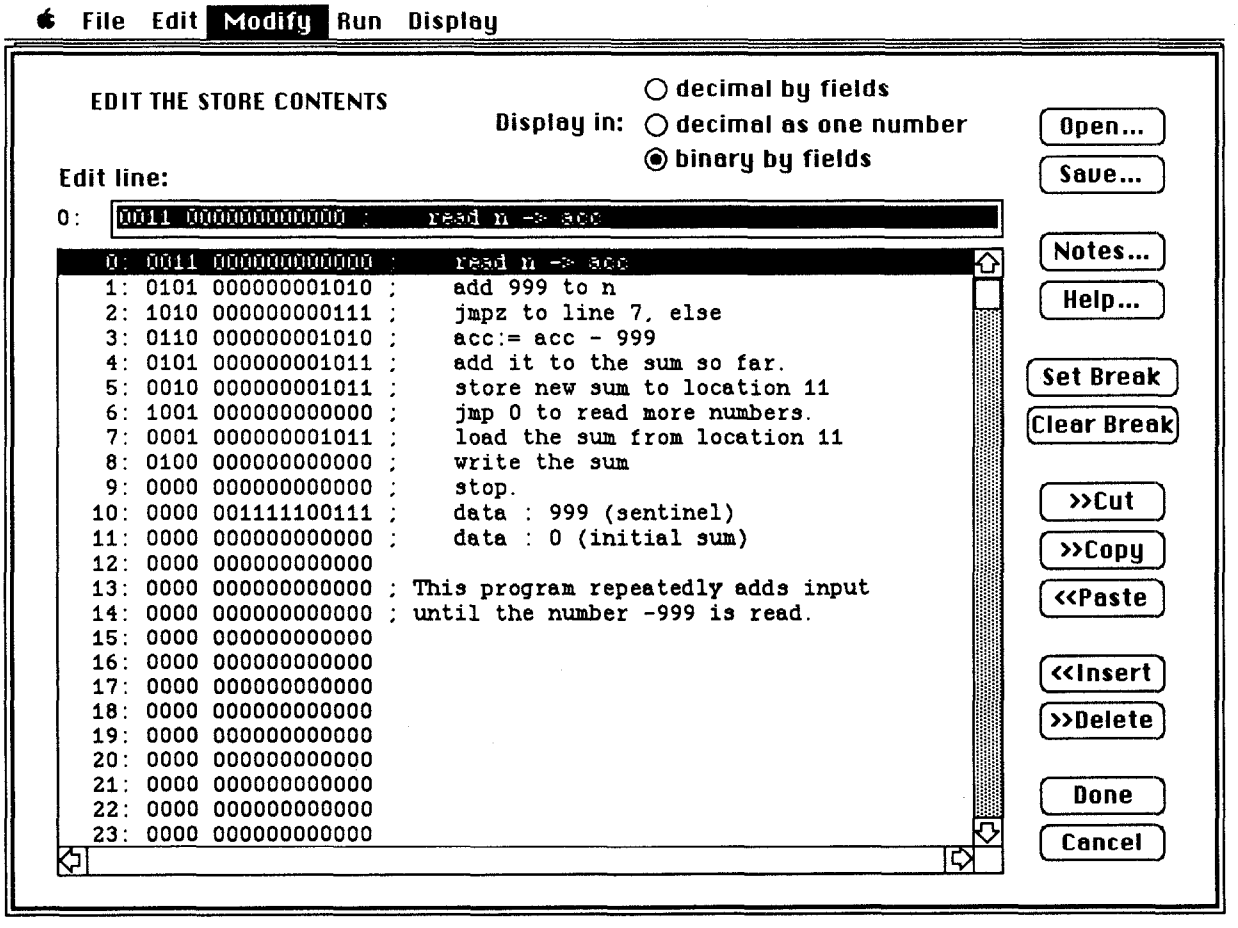

Figure 2.

The new simulator

These exercises have worked very successfully for us. The students have felt they learned a lot through the use of the simulator. However, STARTLE was written in Pascal with purely textual i/o. This has meant that too much time has to be spent teaching the students how to run the simulator; in particular, they had to be instructed in the proper format of the input and output for the simulator, which included many values whose meaning was not immediately clear. There seemed to be some frustration with this that detracted from the purpose of the course. However, with the purchase of a lab of Macintosh IIcx's, Colby obtained the hardware to do much more in the way of providing the students with a user-friendly environment.

For this reason, a project was started to create a new simulator, based on the ideas in STARTLE, that fully utilized the Macintosh interface to allow the students to concentrate on the simulation rather than the simulator.

The result, called CPU SIM, has a main display (see Figure 1) showing the registers, flipflops, channels, the main store and stack of the simulated machine, and their contents (in decimal or binary form). It also displays the current machine instruction, micro-instruction, and the time the simulated machine has used up to this point.

After a hypothetical machine has been created or loaded by the user (including the hardware description, micro-instructions, and machine-instructions), and after initial values have been loaded (see Figure 2 to see how the contents of main memory are loaded and edited), the simulator can start running the program loaded in main memory. It starts executing the instruction at the address specified in the program-counter register. The simulator can run the program until it halts and then display the final values in all memory devices, or several break points can be set, halting the machine at these points for inspection. The user can also step through the program, one machine instruction or one micro-instruction at a time. Finally, the user can backup one machine instruction at a time. This flexibility allows the user to easily find any point at which the program did not perform as expected.

At any time during the execution or before, the user can modify the contents of any of the components. Furthermore, the user can modify the actual simulated hardware, such as changing the widths of registers or adding new machine or micro-instructions.

The hypothetical machine

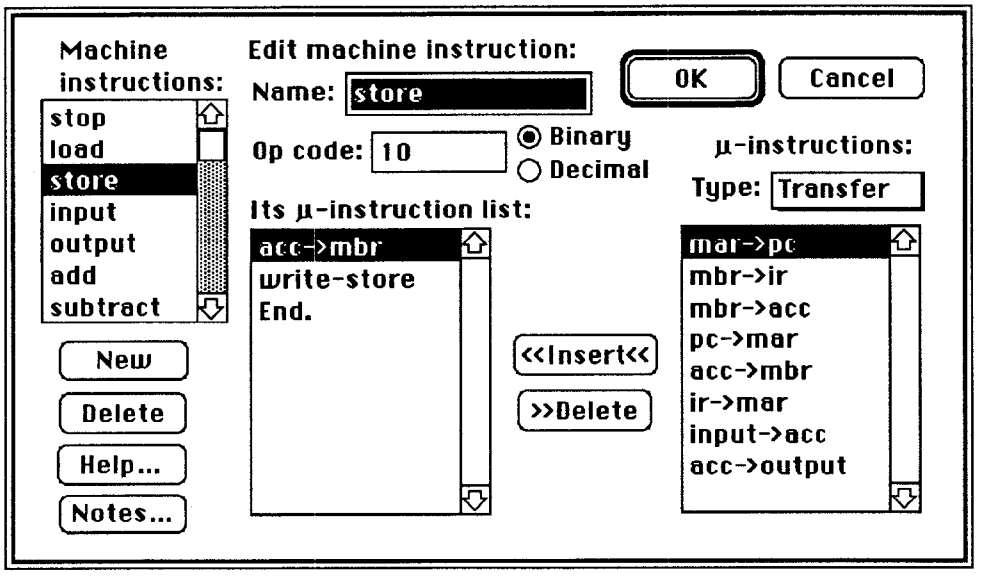

Figure 3.

descriptions created for the simulator can be saved as text files for reloading and execution or modification at a later time. Similarly, the machine language programs to be run on the hypothetical machines and the input data can be saved in separate text files. The output can be directed to a window on the screen or saved as a text file.

The user can create or modify a simulated machine very easily. By selecting appropriate menu items, windows will appear allowing the user to modify or create new hardware components, or change or create new machine or micro-instructions. As an example, Figure 3 shows the window in which machine instructions are edited. CPU SIM was designed so that the user would have to do as little typing as possible, thus avoiding most of the usual typing and syntax errors.

Requirements

CPU SIM was designed using Allegro Common LISP, and runs on any Macintosh with 2.5 megabytes of RAM and a 13-inch (or larger) monitor. The RAM is necessary because of the overhead needed to support a LISP program: the compiled code for CPU SIM is a little over 1 megabyte. A large monitor is needed since several of the CPU SIM windows are too large for the standard 9-inch Macintosh screen. Plans are under way to revise CPU SIM so that it will use less memory and a smaller monitor.

The text files in which the hypothetical machine descriptions are saved consist of actual LISP code utilizing Allegro's Object LISP extensions to the language. Therefore, these files could be modified by the user with any word processor, but could easily result in an unreadable file if the result is not proper LISP code. In any case, it is much easier for the user to modify the machine description using CPU SIM, so this should rarely be necessary.

The machine language program and data files just consist of integers and comments and so allow easy modification by the user at any time with any text editor.

The standard version of STARTLE is written in Pascal and has been ported to UNIX and to MS-DOS (using Turbo Pascal).

Limitations

CPU SIM does not display the contents of its components in HEX or octal because it seemed sufficient to have it display both decimal (to help the students create their input and output) and in binary so that the actual bits can be seen.

The only representation mode allowed is two's complement. The only input and output data the simulator will accept is integers in base 10.

The standard version of STARTLE reads and displays all data in decimal integers, but it does allow the choice of two's complement, one's complement, or signed magnitude for internal representation of data.

Conclusion

A very versatile multilevel simulator is valuable for use in the classroom. It allows the students to use and create many different hardware configurations for comparison. It also allows the students the opportunity to experiment on their own in creating or improving existing simulated machines. We have found it to be an essential component of our course since it gives students hands-on experience with a multilevel approach to machine organization.

References

[1] J.M. Kerridge and N. Willis, "A Simulator for Teaching Computer Architecture," ACM SIGCSE Bulletin, Vol. 12, No. 2, July 1980, pp. 65-71.

[2] F. Tangorra, "The Role of the Computer Architecture Simulator in the Laboratory," ACM SIGCSE Bulletin, Vol. 22, No. 2, June 1990, pp. 5-10.