

A HARDWARE/SOFTWARE SIMULATOR TO UNIFY COURSES IN THE COMPUTER SCIENCE CURRICULUM

Lubomir Ivanov, John S. Mallozzi
Department of Computer science
Iona College
New Rochelle, NY 10801
jmallozzi@iona.edu, livanov@iona.edu

ABSTRACT

We present the design of a small but efficient computer system, a software simulator with a graphical user interface, and supporting software designed to integrate the educational experience of students taking courses in Computer Organization and Architecture, Operating Systems, and Compiler Design. We demonstrate the essential links among these three areas of Computer Science, and discuss how to integrate the developed system and software into the course material of the above-mentioned courses.

I. INTRODUCTION

Although the college experience is divided into discrete courses, the implied separations between various areas of a subject do not occur in reality. This disparity is one that appears in most areas of the college curriculum. In Computer Science, the courses entitled Computer Organization, Operating Systems, and Compiler Design – three fundamental courses in the curriculum – are affected by this problem. These subject areas are integrally linked to each other, and design decisions in one affect the others significantly. Many examples of this intimate interdependency can be brought forth. For example, the number of CPU registers and their intended use can affect the efficiency of the high-level language compiler and the execution time of the programs compiled with it, while at the same time having an impact on the performance of the computer's operating system. Many of the algorithms used to improve the performance of a computer system and decrease software execution times can be implemented as hardware devices (expensive, but very fast), or as software techniques embedded in the functionality of an optimizing compiler (slower, but less expensive, and sometimes more accurate and larger in scope).

These examples emphasize the strong interdependence between software and hardware, and among the courses in Computer Organization, Compiler Design, and Operating Systems. However, these relationships are difficult to explore in the context of separate courses, taught in isolation from each other, usually by different faculty members. As a result, students never perceive the overall picture, and the close ties that link the various sub-fields of Computer Science together. One obstacle encountered in the effort to achieve continuity among courses in the Computer Science curriculum is the lack of a common basis for the classroom discussions – a computer system around which the course material for all three courses can revolve. The computer architecture in question has to be relatively simple, emphasizing the most essential aspects of computer hardware and software design, but leaving out the

intricacies of today's highly advanced machines. Such a machine can serve as a foundation of the discussion of most topics in Computer Organization, and as a platform on which students can develop a real, working high-level language compiler in the Compiler Design class.

This paper describes the design and implementation of a software-driven computer system simulator, an assembly language, and other supporting software, and their integration into the course material of Computer Organization, Operating Systems, and Compiler Design as a foundation for providing strong continuity among these courses.

II. BACKGROUND

The need for architecture simulators has been recognized for decades. It stems from the need to test and verify the operation of a new design before it is embedded into silicon, as well as from educational needs. Thus, architecture simulators have been used both in the industry and in education for many years, and, depending on their intended use, vary greatly in their scope and functionality. Some simple simulators, used primarily in the educational setting, implement the functionality of a small, hypothetical computer, in which the most important issues and design decisions have been crystallized, while more complex or less essential features are left out. These simulators present the student with a working model of a computer system, which is easy to grasp and use, while avoiding the complexity of the powerful modern architectures. Excellent examples of such simple simulators are CASTLE and PIPPIN, which can be downloaded free off the web. Other, more advanced simulators provide a much larger instruction set, and advanced features such as interrupt handling, pipelining, and micro-code level simulation of instruction execution. An excellent overview of the field of architecture simulation can be found in [1,2].

III. ARCHITECTURE AND ORGANIZATION

The abundance of architecture simulators for education notwithstanding, most are aimed solely at aiding the teaching of Computer Organization and Architecture. None of these simulators are intended and (to the best of our knowledge) used outside the Computer Organization course and as an integral part of other Computer Science courses. Moreover, many simulators are not designed with the notion of inter-course continuity in mind, and are often unsuitable for use in any other course but Computer Organization. Thus, when we set out to achieve our goal of unifying Computer Organization and Architecture, Operating Systems, and Compiler Design through the use of a common simulated computer system, we had a number of design features in mind:

- A reduced instruction set with only the most essential addressing modes, which will be convenient for teaching assembly programming, as well as for developing a simple high-level language compiler.
- A set of architectural features that is small enough to avoid the complexities of modern CPUs, yet sufficient to explain the operation of a modern computer system, and to be able to create both a simple compiler and a small but efficient operating system.
- An appropriate, intuitive graphical interface to allow students to explore the features and operation of the machine with minimum guidance from the instructor

- A small, convenient assembly language that is easy to learn and use. We specifically left certain features of the assembly language open to possible programming mistakes so as to enhance the educational experience of our students and point out potential pitfalls.
- A simple, yet functional high-level language such that a compiler for it can be generated during the Compiler Design course.

The organization of the machine involves the following components:

- Registers, R0 through R13 are general purpose registers, with register R0 used implicitly in most data transfers (see instruction set below)
- Register R14 is programmer-inaccessible and used for temporary data storage during swaps.
- Register R15 is the dedicated stack pointer (which is programmer-accessible)
- XAR – External Address Register
- XDI – External Data In Register
- XDO – External Data Out Register
- PC – Program Counter
- IR – Instruction Register
- EMIT – Register for Immediate Values
- PSW – Processor Status Word Register
- 16-bit Integer ALU
- Microprogrammed Control Unit
- Dual internal data bus (16-bit wide)
- The memory space is 4Kx16 and is only 16-bit word addressable
- Memory-mapped IO: addresses 4088 to 4091 are input ports and addresses 4092 to 4095 are output ports

Instruction	RTL Description	Instruction	RTL Description
ADD A, B, C	$A \leftarrow B + C$	SW addr	$M[\text{addr}] \leftarrow R0$
SUB A, B, C	$A \leftarrow B - C$	LWR A	$\text{ACC} \leftarrow M[A]$
MUL A, B, C	$A \leftarrow B * C$	SWR A	$M[A] \leftarrow \text{ACC}$
DIV A, B, C	$A \leftarrow B / C$	MOV A, B	$A \leftarrow B$
SHL A, B, C	$A \leftarrow B \ll C (C > 0), A \leftarrow B \gg C (C < 0)$	LI imm	$R015..12 \leftarrow 0,$ $R011...0 \leftarrow \text{imm}$
AND A, B, C	$A \leftarrow B \& C$	J addr	$\text{PC} \leftarrow \text{addr}$
NOT A, B	$A \leftarrow !B$	BEQZ addr	if $(R0 == 0)$ $\text{PC} \leftarrow \text{addr}$
LW addr	$R0 \leftarrow M[\text{addr}]$	PSW	$R0 \leftarrow \text{PSW}$

Fig.1 (Note: A, B, C are programmer-accessible CPU registers)

The instruction set is given in Fig. 1. It assumes a three-operand, register-register architecture with a uniform 16-bit instruction length and a fixed field instruction format. The opcode field is 4 bits wide, which allows 16 machine instructions. Each register field is 4 bits wide, and the address and immediate fields are 12 bits wide. The limited number of instructions and the need to support the necessary addressing modes forced us to provide some instructions, traditionally present in most machine level instruction sets, at the assembly language level as macros, which are translated to sequences of machine instructions by the assembler. Among these are the OR, PUSH, and POP instructions. Additionally, the HALT assembly instruction is translated by

the assembler into an unconditional jump to address 4095, which the simulator recognizes as a signal to stop further execution.

Many of the architectural and organizational aspects of our machine have been selected with the aim of providing educational examples and exercises that span the three target courses – Computer Organization, Operating Systems, and Compiler Design. The choice of a general purpose register (GPR) architecture provides greater flexibility than a stack or accumulator architecture in terms of designing a simple compiler, while at the same time lowering the instruction count, and, thus decreasing program execution time. The choice of 16 general purpose registers is based on the minimum heuristic requirement for an efficient compiler-based register renaming. At the same time, from the point of view of the operating system designer, this relatively small number of registers guarantees a quick and efficient context switch between processes managed by the Operating System.

IV. THE SIMULATOR

The simulator opens with a “file open” dialog, allowing the selection of either an assembly source file or a file containing assembled code. In the first case, the assembler is called before proceeding (with appropriate interactions in the event of an assembly error). The assembled code is then displayed in the main simulator window (Fig. 2).

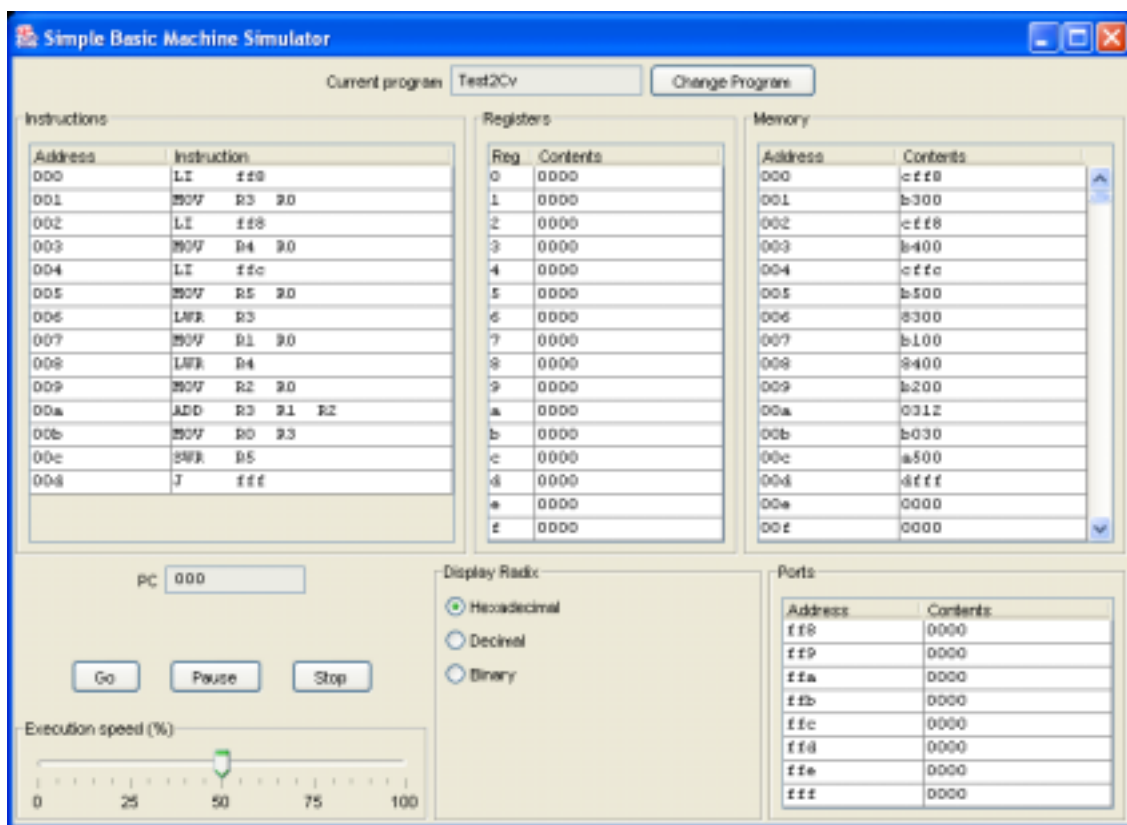


Fig.2 Main Simulator Window

The name of the current program is displayed at the top center of the screen, providing the option to load a new program. The center area is divided into three windows. On the left is a source code rendering of a window into the (preprocessed) instructions and the addresses at which they are stored. In the middle window are the registers and their contents. And on the right is a rendering of a window into the contents of all of memory. In the figure, instructions are stored beginning at address

000 and the memory window also starts at that address. Therefore, the contents of the latter (cff8) represent the coding of the instruction (LI ff8) in the former. The same is true for all address up to 00d. If a machine address containing an instruction is overwritten, the contents of that address will no longer represent the corresponding instruction, and this will be visible to the student. The instruction (left) window is accompanied by a display of the current Program Counter register (PC) value. The user may start (or restart), pause, and stop the simulation, and may also control the speed of execution using the slider at the lower left corner of the screen. These features allow both execution and simple debugging using the simulator. Below the memory contents (right) display is a list of the Input/Output ports. Since these are memory mapped, the same contents will be shown if the memory display is scrolled to the port addresses. Finally, in the lower middle part of the screen are radio buttons controlling whether the numbers are displayed in hexadecimal (the default), decimal, or binary.

Both input and output are accomplished using dialogs. For the given program, input occurs by placing the desired value into an input port. In the sample program shown in Figure 2, the console input port is mapped to address ff8. Instruction 000 places this immediate value into R0 (by default). The following instruction moves this value to R3. The instruction “LWR R3” at address 006 requests movement of data from the memory location pointed to by register R3 (M[R3]) to the accumulator (R0). Since M[R3] is the memory-mapped console input port, this instruction causes the simulator to open a dialog requesting input (Fig.3).

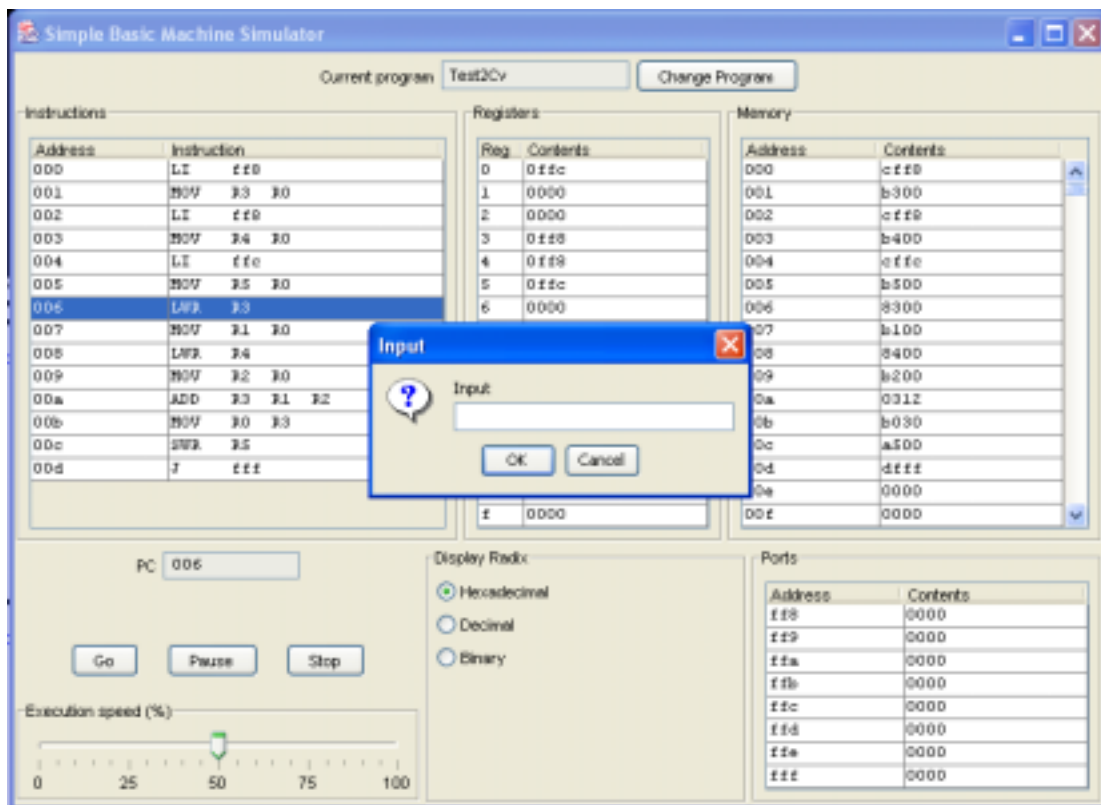


Fig.3 Requesting Input through an Input Port

Data entered into the Input dialog will appear in the port display, in the memory display, and – by virtue of the LWR instruction – in R0. Figure 4 shows the state of the machine following input of 125 (007d) and the copying of the data from R0 into R1 (instruction 007). The memory window has been scrolled down to show address

ff8, which is mapped to the first output port and whose contents have now been copied into both R0 and R1.

The simulator is coded in Java. This language choice was prompted by the ease with which a GUI application can be developed and the possibility to implement the simulator as a cross-platform application. The source code is available to students interested in the implementation.

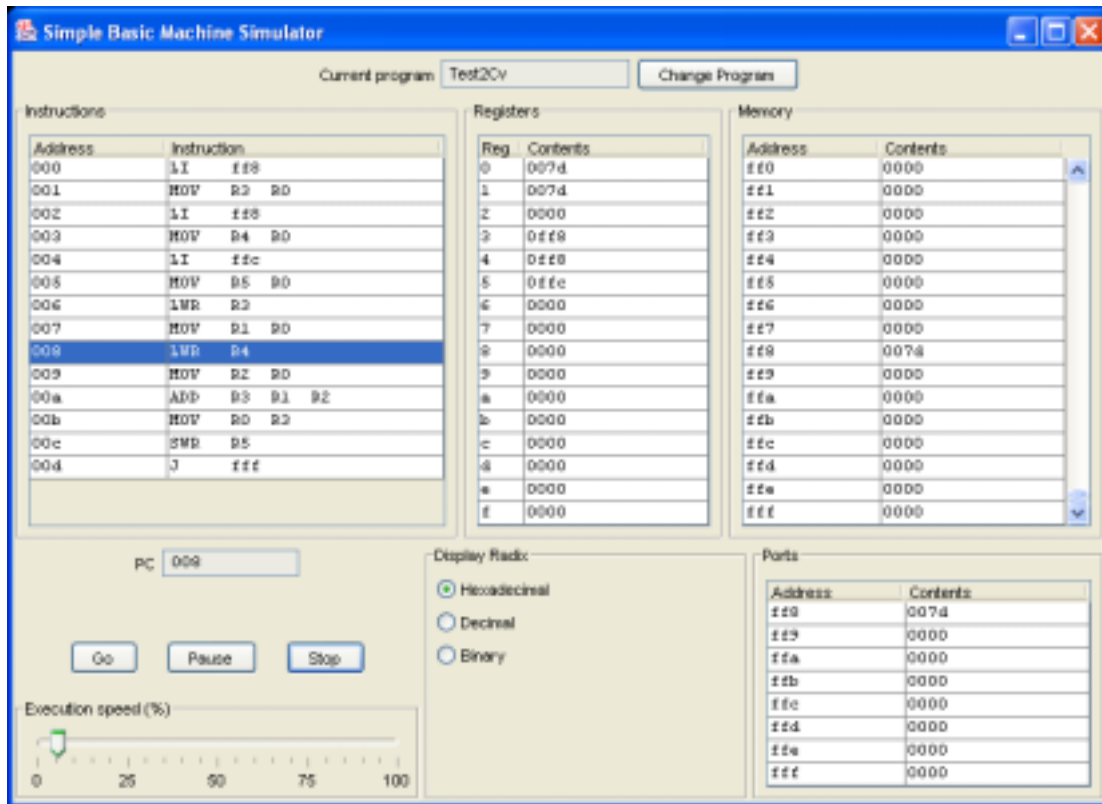


Fig.4 Following Input and Copy

V. ASSEMBLY LANGUAGE

The assembly language provides mnemonics for the machine instructions in Fig.1, but adds the following:

- PUSH instruction
- POP instruction
- OR instruction
- NEG instruction
- HALT instruction
- DW (define word) directive
- DUP (duplicate) directive
- END directive
- STACK directive

The PUSH, POP, OR, NEG and HALT instructions are implemented as macros that substitute appropriate machine language sequences in place of the above mentioned instructions. PUSH and POP do not check for underflow or overflow automatically, leaving this task to the student. The stack is defined with the STACK directive, which specifies the starting address of the stack and the maximum stack size. By default the stack is set to the address just above the IO ports (address 4087), and grows towards lower memory. No error checking is performed with regard to the

stack overwriting the program's code and data areas. We believe that encountering this error will be a valuable educational experience for the students.

The inclusion of the NEG instruction is prompted by the fact that LI instruction allows only unsigned 12-bit immediate values. Using all 12 bits for representing unsigned magnitudes is necessary in view of the 12-bit memory addresses of the simulated machine, and the 4-bit opcode and the fixed 16-bit instruction length do not allow for an additional sign-bit. Therefore, the loading of negative numbers is done by first loading an unsigned 12-bit immediate, then using the NEG instruction to convert that to two's complement.

The DW directive can be used to declare variables, tables, and strings. For example, "A DW 6" declares a variable A and initializes it to the integer 6. The statement "A DW ?" declares, but does not initialize variable A. To declare tables, the DUP directive is needed: "T DW 10 DUP 0" declares a table of 10 elements, each initialized to 0. Strings are not null-terminated, and are declared as follows: "STR DW 'Hello'".

The END directive indicates the physical end of the source code, and, at the same time, is used to provide the program's entry point, i.e. the address of the first instruction to be executed.

The assembler works in two stages. During the first stage the source file is read, the macros are processed, some error checking is performed, and the symbol table containing the addresses of all symbolic labels is prepared. During the second pass, the machine instructions are assembled along with some additional error checking, and then written to a machine executable file. At the beginning of the executable file, the assembler includes an RST table, which specifies the relocatable fields (the R section), the starting (entry) point of the program (the S section), followed by the actual code (the T (text) section).

VI. HIGH-LEVEL LANGUAGE

A succession of simple but reasonably complete languages is used in the Compiler Design course. The first of these has only facilities to declare variables of simple data types and to define functions. The first few productions in the language syntax (using EBNF) are shown below (Students are also given a fairly detailed operational description of the language semantics). Control structures are Ada-like. Functions may be declared or defined, as in C++ or Ada.

```
<program> → program identifier is
           [global {<var dec>} end global]
           [functions {<fn dec>} end functions]
           begin <statement list> end [identifier]
<var dec> → identifier {,identifier} as
           [constant] <type> [initially <value>];
<type>    → <array> | <built in>
<built in> → integer | real | boolean | char | string
<array> → array[ intlit ] of <built in>
```

Although fairly simple, this is expressive enough and provides sufficient functionality to give a reasonably true-to-life compiling experience. Students compile the language into intermediate code (quadruples) and are provided with – and eventually themselves implement parts of – a translator of the quadruples into the assembly language of our machine.

VII. USING THE SIMULATOR IN THE CLASSROOM

1. Computer Organization and Architecture

Naturally, the Computer Organization and Architecture course offers the instructor numerous opportunities to use the simulator in lectures and discussions. The simulated machine can be used as the fundamental organization, on the basis of which the internal components of a typical CPU (e.g. general and special purpose registers, ALU, control) can be discussed, along with design decisions regarding component interconnection and control. The issues of program execution, the fetch-decode-execute instruction cycle, and microprogramming can be easily and visually explored in the context of the machine simulator. The instructor may use the selected instruction set of the simulated machine as a starting point for the discussion of addressing modes and architecture types (stack, accumulator, general-purpose register, etc.), and link the material further to the issues of creating a two-pass assembler and a compiler for a high-level language. In particular, it is instructive to discuss how the choice of specific addressing modes enables or facilitates the implementation of programming language constructs such as loops and decision statements, and how the features of a high level programming language must be taken into consideration when deciding on the specific addressing modes, and instruction formats to be included in the instruction set of the machine being designed.

2. Operating Systems

Most Operating Systems courses require that the students implement some aspects of a simulated operating system, such as process management, memory management, and a command interpreter in a series of course projects. In many cases, no specific reference to a particular architecture is given, so the simulation does not really illustrate to students the intricate interaction between the operating system and the underlying hardware. Using our simulated machine as a hardware platform, students get the opportunity to write a small, but functional OS kernel for a “real” machine. Thus, they get to experience first hand the various issues involved in designing an operating system for a specific hardware platform: swapping register values between the process control blocks in memory and the CPU registers during a context switch, transferring control from the operating system to a user process and back to the operating system, memory allocation and deallocation, etc.

Particularly instructive is the aspect of this project involving IO transfer and control. Students explore the issues of handshaking and synchronization in the process of creating IO routines that control the transfer of data through the IO ports of the hardware system. Nowhere else is the intimate interdependence between the hardware and the operating system software revealed better than in this part of the course project. In addition, the IO routines developed in the Operating Systems course, can be used during the following semester when students take Compiler Design, thus, once again emphasizing the mutual interdependence among Computer Organization and Architecture, Operating Systems, and Compilers.

One of the difficulties in designing and implementing such an operating system, however, is the limited amount of main memory available – only 4Kb. This however, is an excellent opportunity to discuss the issues of design efficiency, and the space vs. time complexity tradeoffs. In essence, only the most important parts of the operating system are kept in memory, with IO routines and other less often needed components only brought in when necessary. Additionally, it’s important to realize that, since this is a simple simulated system, only very few small user processes will need to share the memory space, so the majority of the space is available to the operating system. One additional issue this brings up is protecting the operating system code and data structures. Since our hardware simulator does not provide a relocation register or dual (system/user) mode of operation, it is entirely possible that a user program can overwrite some critical parts of the operating system code and data structures in

memory. This is an interesting, educational experiment, and demonstrates to students the need to partition memory into a kernel space and user space. Thus, this experiment could be a prelude to a more in-depth discussion of memory partitioning in real systems, and the importance of hardware support for the safe and efficient operation of the system.

3. Compiler Design

The high-level language described earlier is, over the course of the semester, translated into an intermediate code (quadruples). The simulator comes into play to allow students to execute their code once it has been translated. (As noted, a translator of quadruples into the assembly code is provided). After translation, the student may invoke the simulator to watch the translated program run (and probably learn something about old-fashioned debugging techniques). Near the end of the course, we discuss the translator in some detail, and in this very concrete but simple context, the question of code optimization. There is also ample opportunity to discuss such matters as machine design issues in the context of language design, the value of callable operating system routines to supplant such pleasantries as direct generation of I/O instructions, etc.

VII. CONCLUSIONS

In this paper we presented a computer system simulator and the supporting software aimed at unifying the three fundamental courses in the Computer Science curriculum – Computer Organization and Architecture, Operating Systems, and Compiler Design. Our approach to the design of a machine and its simulator involves a significant tradeoff – the use of a very simple architecture in all three of the above mentioned courses. However, this choice does not preclude the discussion of more complex architectures. Rather, it provides an easily accessible starting point for further in-depth discussions of more complex issues, and offers significant educational experience for students, extending over several courses, using a concrete architecture in multiple settings. We believe that, through this experience, students are able to better appreciate the interconnections among the areas involved, in a way that mere references within individual courses does not bring forth.

The presented material is work in progress: the authors are continually improving the simulator, its graphical user interface, and the supporting software by adding new features relevant to the educational experience of our students. We intend to eventually expand the simulation to microcode level, so that students can observe the step-by-step fetching, decoding, and execution of individual instructions as it occurs inside the CPU.

We hope that integrating the simulation environment into the course material of Computer Organization and Architecture, Operating Systems, and Compiler Design will enhance the educational experience of our students, and, by revealing the close relationship among the subject areas – between software and hardware in general – will better prepare them for the demands of today's dynamic computer industry.

REFERENCES

- [1] G. Wolffe, W. Yurcik, H. Osborne, M. Holliday "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", proc. of SIGCSE 2002, Northern Kentucky, USA, Feb/March 2002.
- [2] W. Yurcik, G. Wolffe, M. Holliday "A Survey of Simulators Used in Computer Organization/Architecture Courses", proc. of the Summer Computer Simulation Conference (SCSC), Orlando, FL, USA, July 2001