Using LMC Simulator Assembly Language to Illustrate Major Programming Concepts

William Yurcik
Larry Brumbaugh
Department of Applied Computer Science
Illinois State University
Normal, IL, 61790,USA

Abstract

Examples are given that describe how the Little Man Computer (LMC) Model and its associated assembly language code can be used to illustrate a wide variety of core programming topics including a loader program, relocatable and impure code, array processing, function calls, and multitasking. We share this experience as an example "best practice" for incorporating core programming concepts within a computer engineering course.

Keywords: simulation, computer engineering, assembly language, operating systems, education

Teaching the basic concepts behind computer operation is not an easy task. It is complicated by a wide range of subject domains (electrical engineering, computer science, mathematics, physics), discrepancy between theory and practice, wide ranges in level of abstraction, and the lack of coordinated teaching tools [CASSEL]. There has also been two public academic debates over pedagogy in teaching computer operation: (1) top-down versus bottom-up and (2) increasingly higher levels of abstraction. Top-down argues for teaching the familiar to the unknown while bottom-up argues for teaching the unknown to better understand the familiar. The increasingly higher levels of abstraction argument states that computer hardware is a commodity such that a student does not need to learn details they may never design or directly use but rather they should start their learning with higher level languages (C, C++, or Java as opposed to assembly language).

We take the stance for bottom-up learning with information-hiding only as students learn the lower layers of abstraction based on the following reasoning: students can build upon their own knowledge to hide lower level information only when they have internalized the lower level information to hide. While educators may have themselves internalized lower level computer operation, we feel it would be a mistake to assume students will do the same especially when faced with their sometimes strange ideas of how a computer "really works". We feel that not only computer science majors but all IT students and maybe all undergraduates should have a strong understanding of basic computer operation. In this endeavor, we have been very successful in the approach we now report.

This paper first presents a computer system simulator that implements a hypothetical computer architecture for instructional purposes. The simulator is accessible via the Internet and has been used in a classroom setting for the last two years. The remainder of this paper is organized as follows: Section 3 surveys previous relevant work and Section 4 introduces the pertinent instruction set. Section 5 presents illustrative examples of specific simulator assembly language projects for major programming topics. We close with a summary and future directions in Section 6.

1. THE LMC SIMULATOR

The Little Man Computer (LMC) paradigm was developed by Stuart Madnick and John Donovan both of MIT during the 1960s where it was taught to all MIT undergraduate computer science students. The paradigm has stood the test of time as a conceptual device that helps students understand the basics of computer operation. Irv Englander of Bentley College currently has a popular textbook that continues the LMC tradition [ENGLANDER].

The LMC paradigm consists of a walled mailroom, 100 mailboxes numbered 00 through 99, a calculator, a two-digit location counter, an input basket, and an output basket. Each mailbox is designed to hold a single slip of paper upon which is written a three-digit decimal

number. Note that each mailbox has a unique address and the contents of each mailbox are distinct from the address. The calculator can be used for input/output, temporarily store numbers, and to add and subtract. The two-digit location counter is used to increment the count each time the Little Man executes an instruction. The location counter has a reset located outside of the mailroom. Finally, there is the "Little Man" himself, depicted as a cartoon character, who performs tasks within the walled mailroom. Other than a reset switch for the location counter, the only communication an external user has with the Little Man is via slips of paper with three-digit numbers put into the input basket or retrieved from the output basket.

The analogy between LMC and real computers is not perfect. In a real computer, memory (mailboxes) are separated both physically and functionally from the central processing unit (CPU). In most computers, registers are available to hold data temporarily while it is being processed. Although the LMC paradigm has no registers, the calculator display loosely serves the purpose of an accumulator. Clock timing and interrupts are not part of the LMC paradigm. Lastly, the LMC instruction set is based on the decimal system, not binary as a real computer would be. In spite of these deviations from reality, the use of this simple but powerful model with a more familiar number system allows students to focus on understanding the tasks being performed in executing instructions rather than the sometimes complex details of a specific manufacturer's implementation. This LMC simulator can be found at the following URL: <http://www.acs.ilstu.edu/faculty/javila/lmc/>

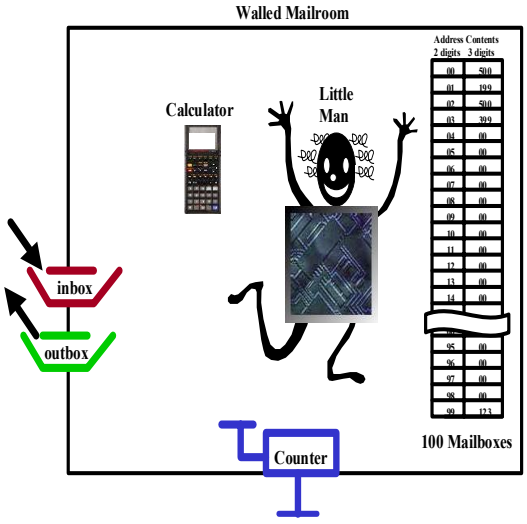


Figure 1. The Little Man Computer Paradigm

2. PREVIOUS WORK

The growing trend toward the use of computer simulators to teach computer operation is documented in [YURCIK/WOLFFE/HOLLIDAY] and [YEHEZKEL/YURCIK/MURRAY]. With the success of the LMC paradigm, LMC simulators have especially proliferated. The different types of LMC simulators are summarized in [YURCIK/OSBORNE]. Despite starting with the same paradigm and the same goals, each implementation is distinct with its own strengths and weaknesses.

The Internet-accessible LMC simulator for which this paper is based is a web-based application implemented in Java embedded within an applet as described in [YURCIK/BRUMBAUGH, YURCIK/VILA/BRUMBAUGH]. The only user requirement is a Java-enabled browser such as Internet Explorer 4.0 (or higher) or Netscape Navigator/Communicator 3.0 (or higher) and LMC can be accessed anywhere via the Internet. User documentation is available via separate web help menus and within the application itself.

3. LMC INSTRUCTION SET

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

Table 1
LMC 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 non-negative	SKP
9	JUMP – goto address	JMP XX

NOTE: XX represents a two-digit mailbox address

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!

Like most assembly language instruction sets, it is difficult to write useful functions in a small number of source code lines. We use this to emphasis the relationship of high-level languages to assembly language in terms of programmability, user friendliness, and efficiency. 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 basic concept that a computer is nothing more than a machine capable of performing simple instructions at very high speed. We discuss instruction set variations as the major difference between types of computers and show examples of different code that performs the same task.

4. ILLUSTRATIVE EXAMPLES

This section describes specific LMC assembly language projects we have used in the classroom to illustrate major programming concepts.

Loader Programs

A Loader program can be written with LMC instructions to load programs from disk into memory. A copy of the basic code to do this can be found in the Solutions Manual that accompanies [Englander]. A very simple Loader copies one program into memory and then the operating system executes the program. Since no assembly is performed, the code loaded into memory must be in 'executable' format, instructions consisting of opcodes and operands (addresses). A delimiter separates the program from any input it processes during execution. A simple loader can be written with 'impure' code. This is code that is modified during execution. The primary STORE instruction is constantly modified by incrementing its address by 1. The Loader program can also be written using pure code. The STORE instruction is replaced by a STORE * (indirect addressing). Here a pointer identifies where an instruction is placed in memory.

A more powerful Loader program can be created that loads multiple programs into memory. When a program finishes execution, control returns to the Loader, which loads the next program. As this version of the Loader copies a program into memory, it examines the instructions and for each HLT instruction a JMP to the initial instruction in the Loader is copied into memory (Figure 2). Like DOS, in this version of the Loader each program is loaded into the same location in memory and only one program can be executing. The Loader becomes (or represents) part of the operating system kernel.

An even more versatile Loader program can switch back and forth between:
a) loading part of one program
b) let another program execute
c) resume loading operations on the first program
d) let another program execute, etc.

This requires that programs be loaded at different locations in memory. An address translation may need to occur for programs not loaded at specific addresses. This is a non-trivial function to perform with the existing version of LMC. The code to do this can be described, but it has not been implemented with only 100 memory locations.

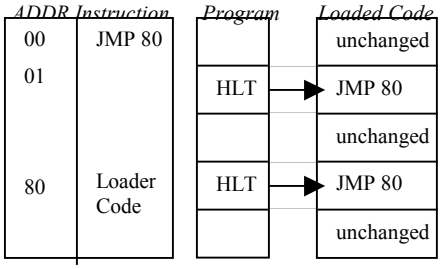


Figure 2. Loader Program

Relocatable Programs and Impure Code

In order to execute a program, the instructions and the data it processes are copied into memory. The data is considered part of the program. When absolute and direct addressing is used, the program is NOT relocatable since addresses identify specific locations in memory. There are several basic problems with non-relocatable code: it requires specific memory locations, prohibits other programs from using those locations, and only one copy of the program can be executing. However, if only relative, immediate and base/displacement addressing are used, the program is automatically relocatable.

There are several basic problems with impure code. It cannot be stored in Read Only Memory. The executing program cannot be restarted in mid state and when execution is finished, it cannot be rerun.

As mentioned with the Loader, impure code can usually be converted to pure code by using indirect addressing. Comparable results can be achieved with base displacement addressing or even relative addressing, in some circumstances.

Arrays

As with most programming languages, using arrays greatly increases the capabilities of the LMC language. Many common array processing functions can be performed using LMC instructions. Three basic parameters are specified for each function: the initial location in the array (Location N in memory), the final location in the array (Location N+M) and the size of an array element. The direction of processing is assumed to begin at Location N and continue through Location N+M.

Typical array processing operations include:

a) Find the largest (or smallest) element in the array
b) Find the sum of the elements in an array
c) Search the array for an element. If it is found, in what location(s)?
d) Sort the elements in an array
e) Merge two sorted arrays into a third array

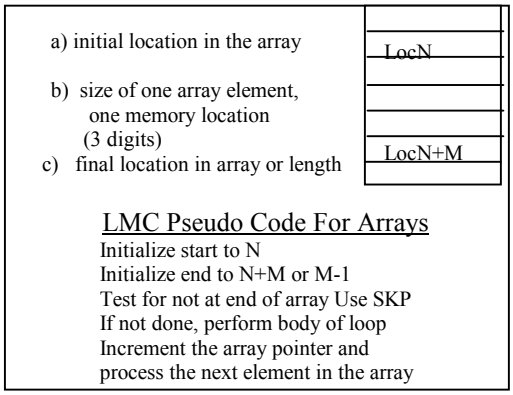


Figure 3. Array Processing

Mailbox	Instruction		Comments
00	LDA 95	N	; this loads "LDA 41"… N=41
01	STA 04		; into address 04 in memory
02	LDA 41		; largest element found so far
03	STA 94		;stored in location 94
04	0		; loads next element in the array
05	SUB 94		; subtract largest element found to this point
06	SKP		; is a new largest number found?
07	JMP 10		
08	ADD 94		; new largest number (new# - old# + old#)
09	STA 94		; and it is stored
10	LOAD 04		; retrieve the instruction in address 04
11	ADD #	1	; increment address in instruction +1
12	STA 04		; instruction has now been changed
13	LDA 93		
14	SUB #1		;decrement the counter
15	STA 93		
16	SKN		; has the end of the array been reached?
17	JMP 04		;not done, continue array processing
18	LDA 94		; processing finished
19	OUT		; display answer
20	HLT		
93	DAT 14	M-1	;final array location identified (15 locations)
94	DAT		;holds largest number found to this point
95	LDA 41		;processed as data/executed as instruction

Figure 4. Array Processing Example: This program determines the largest element in the array that occupies locations 41 to 51 in memory.

Increasing the Size of the LMC Instruction Set

Multiplication and division can be implemented using repeated addition and repeated subtraction respectively. Remainders can also be calculated. The multiplication of M and N can be optimized by adding M to itself N times, where N is the smaller of the two numbers. A comparison is performed prior to the addition.

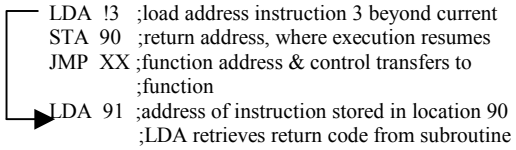
A corresponding optimization is not apparent with division. However, repeated addition can also be used with division. To divide 14 by 3, add 3 to itself until the sum exceeds 14. Here 3+3+3+3+3=15. One less 3 is the answer, here 4. The remainder is 14-12=2.

Function/Subroutine Calls

A fundamental programming concept is the function or subroutine call. It contains five important components: These include: (1) the actual transfer of control, (2) the

return from the function to the next statement following the call, (3) passing parameters between the calling and called routines, and (4) a return code generated by the called function. Note: (3) and (4) are part of (5), a general save area. These basic application and system programming concepts are implemented using LMC code.

The code to transfer control and then resume is at the next instruction is:



The final lines executed in the function should be:

```
;this comment is the first line in the function
;executable function statements
;next to last statement in function –
; return code has been calculated
STA 91  ;return code is in Location 91
JMP *90  ;return to the address stored in location 90
```

The above code illustrates several of these points. Here it is assumed that location 90 (or more commonly a register) holds the location of an instruction where execution is to resume when the function finishes. Ordinarily it is the next instruction. Likewise, the function uses location 91 to return a code to the calling routine identifying success or failure. It is important that both programs understand the role of locations 90 and 91 in memory and not use them for any other purpose. A general save area and parameter passing can use a designated area in memory agreed upon by the calling program, the function and (perhaps) the operating system.

Multi-Tasking

A multi-tasking component of the operating system (consisting of the Loader and any additional code) can be created that executes several programs concurrently. The operating system needs to be resident in memory. The operating system needs to remember the next instruction it is to execute. These need to be stored somewhere along with the data the program is using. Some of the programming specifics are included with the Functions topic above. For simplicity, the unit of time allocated to each executing program is two instructions (or one if there is an abend or the program ends normally). The concept of a thread could be introduced here as one program executes in several different places.

5. SUMMARY

In this paper we have presented examples of how a web-based computer simulator can be used to convey major fundamental computer engineering concepts. We are convinced by class performance and feedback that interactive visualization of computer architecture and assembly language is a powerful tool to help students understand both low-level computer operation as well as higher-level language programming constructs.

6. ACKNOWLEDGEMENTS

The authors would like to thank the following students who were instrumental in developing the ideas of this paper: (1) Rahul Gedupudi who programmed the latest version of LMC as a Masters Project and continues to maintain and provide upgrades and (2) the students within the ACS 254 classes at Illinois State University from Fall 1999 to Spring 2001 who learned along with us the value of active learning using interactive simulation.

7. REFERENCES

Cassel, Lillian (Boots), Deepak Kumar et. al., (to appear). “Distributed Expertise for Teaching Computer Organization and Architecture,” ITiCSE Working Group Report July 2000.

Englander, Irv, 2000, The Architecture of Computer Hardware and Systems Software 2nd edition, John Wiley & Sons, Inc.

Yehezkel, Cecile, William Yurcik, and Murray Pearson, January 2001, "Teaching Computer Architecture with a Computer-Aided Learning Environment: State of the Art Simulators," 2001 Intl. Conf. on Simulation and Multimedia in Engineering Education (ICSEE 2001), Soc. for Computer Simulation (SCS).

Yurcik, William and Larry Brumbaugh, February 2001, “A Web-Based Little Man Computer Simulator**”,** 32nd Tech. Sym. of Computer Science Education (SIGCSE), pp. 204-208.

Yurcik, William and Hugh Osborne, December 2001, “A Crowd of Little Man Computers: Visual Computer Simulator Teaching Tools,” Winter Simulation Conference (WSC).

Yurcik, William, Joaquin Vila, and Larry Brumbaugh, August 2000, “An Interactive Web-Based Simulation of a General Computer Architecture”, IEEE Intl. Conf. on Engineering & Computer Education (ICECE 2000), São Paulo Brazil.

Yurcik, William, Greg Wolffe, and Mark Holliday, July 2001, “A Survey of Simulators Used in Computer Organization / Architecture Courses”, Summer Computer Sim. Conf. (SCSC), Soc. for Computer Simulation (SCS).