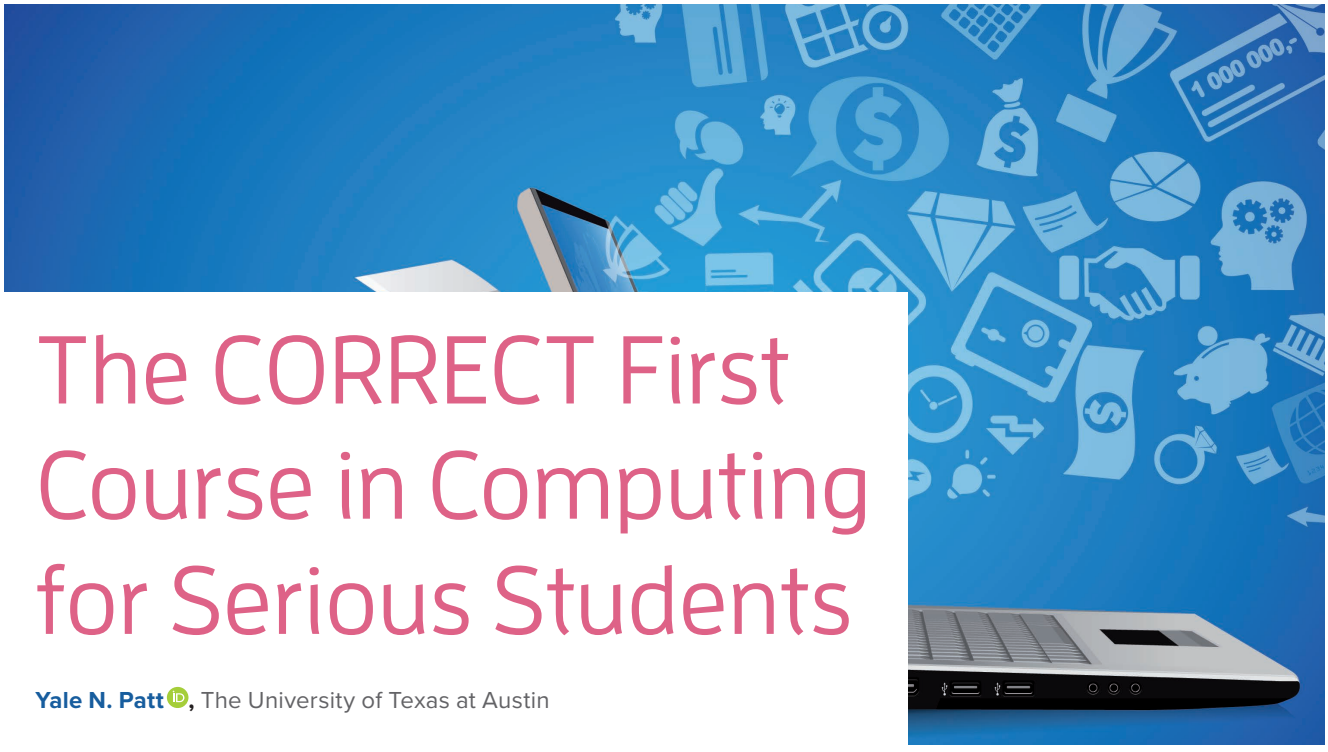


# The CORRECT First Course in Computing for Serious Students

**Yale N. Patt**, The University of Texas at Austin

*This article proposes a bottom-up approach to the first course in computing, in contrast to the conventional top-down approach of teaching programming in a high-level language first, describing details of the course and its benefits.*

Conventional wisdom in computer science and engineering education trumpets "Introduction to Programming in High-Level Language X" as the preferred first course in computing to be taken by every first-year student. And, for every language X, dozens of textbooks are available from which to teach. Unfortunately, students have no foundation regarding how the computer works, so they are reduced to memorizing rather than understanding what is going on. Memorizing is not learning. In 1995, with the support of the Electrical Engineering and Computer Science faculty at the University of Michigan, we embraced what we believe is a better way: the motivated bottom-up approach. Almost 30 years later, I believe more than ever that this is the correct way to introduce first-year students to computing.

In the Fall of 1995, my University of Michigan colleague Prof. Kevin Compton and I set out to introduce computing to first-year students in a very different way than conventional wisdom dictated: "the motivated bottom-up approach."

The idea first surfaced in a faculty curriculum committee meeting chaired by Kevin a year earlier. Faculty moaned and groaned that students who successfully completed the first course did not understand important fundamental concepts. Pointer variables were confusing, and recursion looked like magic. I pointed out that students did not know these things because they were expected to program in that course without understanding how the computer worked. I suggested that if students did not know that memory location 4 containing the value 6 is not the same as memory location 6 containing the value 4, then understanding would be

*Digital Object Identifier 10.1109/MC.2024.3356828*
*Date of current version: 5 April 2024*


0018-9162/24©2024IEEE

EDITORS
**GEORGE HURLBURT** U.S. Federal Service (Retired), USA;
gfhurlburt@gmail.com
**SOREL REISMAN** California State University, USA;
sreisman@computer.org


close to impossible. I offered to work with Kevin to develop an appropriate first course for our freshmen that would build a proper foundation in their first semester at the University of Michigan. I was teaching the junior-level computer organization course at the time, so I asked my students in that class what they thought of my teaching the junior course material to freshmen instead of teaching the Intro to Programming in C. Their response: "Go for it."

So, in the fall of 1995, Kevin and I taught the "new" course for the first time. We started with an overview of computing to motivate what we would teach during the semester. Then we introduced the switch-level behavior of n-type and p-type transistors. The switch-level behavior of transistors is like wall switches that switch lights on and off, something the students had been doing since they were two years old. From there we designed an inverter with one p-type and one n-type transistor. Next, we added more transistors, producing AND, OR, NAND, and NOR gates. Then we used these gates to design MUXes, decoders, one-bit ADDers, and gated latches. With those structures as building blocks, we next designed memory and finite-state machines, continually raising the level of abstraction and building on what they had previously learned. Next came the design of a computer, the LC-3 (little computer #3), because I needed three passes before I was happy with the result. Every step of the way, we built on what they knew, raising the level of abstraction and requiring very little magic.

That was almost 30 years ago. I taught the course four times at the University of Michigan, then moved to Texas, and have taught it at the University of Texas every other fall semester since. We needed a textbook, since there was none on the market, although there were hundreds of *Intro to Programming in High-Level Language X* books devoid of how the computer worked. Kevin did not have time to join me in writing a textbook, so I offered the task to my Ph.D. student Sanjay Patel, who was a teaching assistant in the course the first time we taught it. He eagerly accepted, and shortly thereafter (in 2000), *Introduction to Computing Systems: From Bits and Gates to C and Beyond* was born. The second edition followed in 2004, and the third (with a slightly modified title since we included insights into C++) in 2019.[1] We have had more than 400 adoptions so far, and every year a few more schools jump on board. University of California, Riverside adopted the book before it was even published and has continued to use it. Current adopters include the University of

> Almost 30 years later, I believe more than ever that this is the correct way to introduce first-year students to computing.

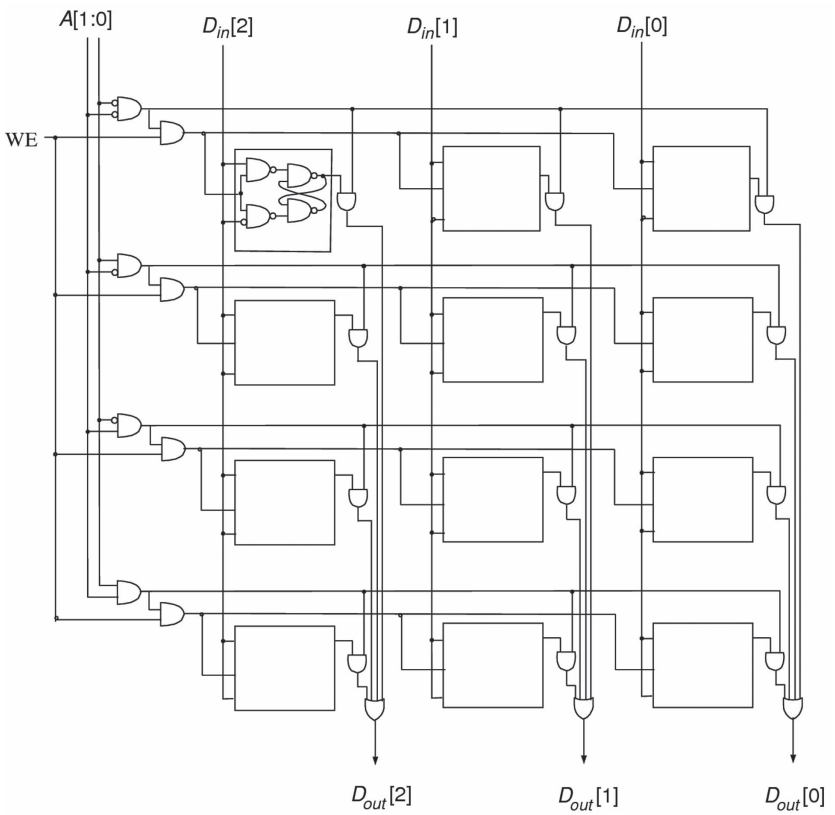


**FIGURE 1.** A $2^2$ x 3 bit memory which students understand by the end of the second week of the semester.

Texas, University of Illinois, University of Wisconsin, Georgia Tech, North Carolina State University, University of Pennsylvania, Shanghai Jiao Tong University, Zhejiang University, and the Chinese University of Science and Technology in Hefei, among others. The general sentiment has been that students love the book because they understand, and teachers love it because the students understand.

## TOP-DOWN VERSUS BOTTOM-UP

Why does conventional wisdom in football dictate bottom-up? That is, football starts with drills before the first scrimmage. Karate starts with mastering basic moves before one engages in combat. Learning to play the piano also begins with mastering basic skills. However, Intro to Computing starts at the top. Why?

I have a few conjectures. Perhaps it is due to a fundamental misunderstanding about learning versus design. Design should be top-down; otherwise, you end up with the old design plus a few new wrinkles. I very much embrace top-down design as correct design. However, design comes after one understands the components. Learning top-down means learning before one understands the components. That usually ends up as memorizing, not learning.

Other reasons I have heard supporting top-down learning suggest a desire to create glitz to attract students to the major. I have found that serious students do not need glitz to be wooed to the major. Another argument for top-down is that companies need JAVA programmers. However, the students are freshmen and will not graduate for another four years. By then, want ads will probably no longer cry for JAVA programmers. Also, the current thrust is for object-oriented, as a productivity enhancer, wherein there is no time to understand the details of computer architecture, operating systems, and compilers. Without that understanding, programming is reduced to memorizing, not understanding.

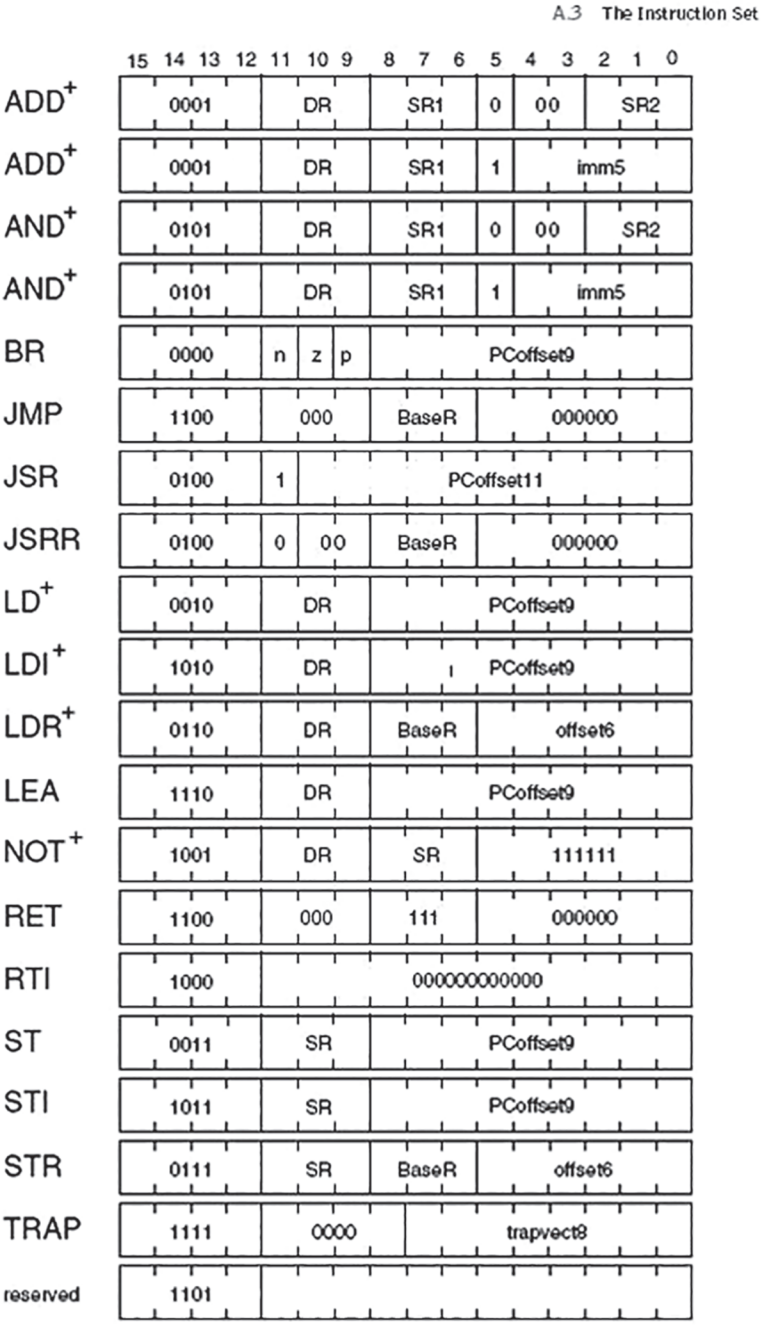


**FIGURE 2.** The LC-3 ISA. Note: + indicates instructions that modify condition codes.

## MY MANTRA

My mantra for teaching is very straightforward. Start with what they know: wall switches, not quantum mechanics. Build on what they know, continually raising the level of abstraction, memorizing as little as necessary, and trying very hard not to introduce magic. Figure 1 shows the design of a memory after they have designed a latch, a decoder, and a MUX; 2^2 locations, with three bits stored at each location, Not 4GB or whatever is contained in students' laptops today. Students master Figure 1

before the end of the second week of the semester.

## THE LC-3

Choose a computer model that is simple enough that students can grok it without being overwhelmed yet rich enough that they can do useful work. Figure 2 shows the entire instruction set of the LC-3. Just 15 opcodes! Figure 3 shows the data path necessary to implement the LC-3 and Figure 4 shows the state machine for the LC-3. In deciding on the LC-3, I rejected the obvious alternatives. I did not adopt the x86 because I felt it was too complicated for the student's first instruction set architecture (ISA). I rejected using a subset of a real ISA because ISAs are produced by companies wanting to make money, and I have found that such ISAs always come with quirks that create an economic advantage at the expense of extra (and unnecessary for learning) complexity of the instruction set.

I chose the LC-3 because I had the luxury of being able to ignore economic considerations and design for pedagogical clarity. Even with this simpler instruction set, we could still get the desired WOW effect. For example, first-year students have designed game programs with this ISA. An essential part of the course is the LC-3 simulator/debugger, which students use to easily debug their programs by themselves without needing help from the instructor or teaching assistant.

Finally, there is room to augment the ISA in later courses. Among the things we left out (and left for future courses) as unnecessary in the first course are endianness, unnecessary data types, unnecessary addressing modes, a multiply instruction (a procedure can compute the product of two numbers), a shift instruction, a carry condition code, and an ADDC instruction. One important insight of our approach is that one can use our simple set of instructions to write many practical procedures, which is a hallmark of computers.

## THE BENEFITS OF THE BOTTOM-UP APPROACH

There are many benefits of this approach. We eliminate magic, minimize memorization, and maximize understanding. Students build on what they have learned earlier in the course. They debug the programs they write by themselves because they can. An extra benefit is that this approach produces a broad base foundation. Students are exposed to and use data representations, logic gates and structures, finite-state machines, basic computer organization, assembly language, assemblers, physical input/output, and

> The general sentiment has been that students love the book because they understand, and teachers love it because the students understand.

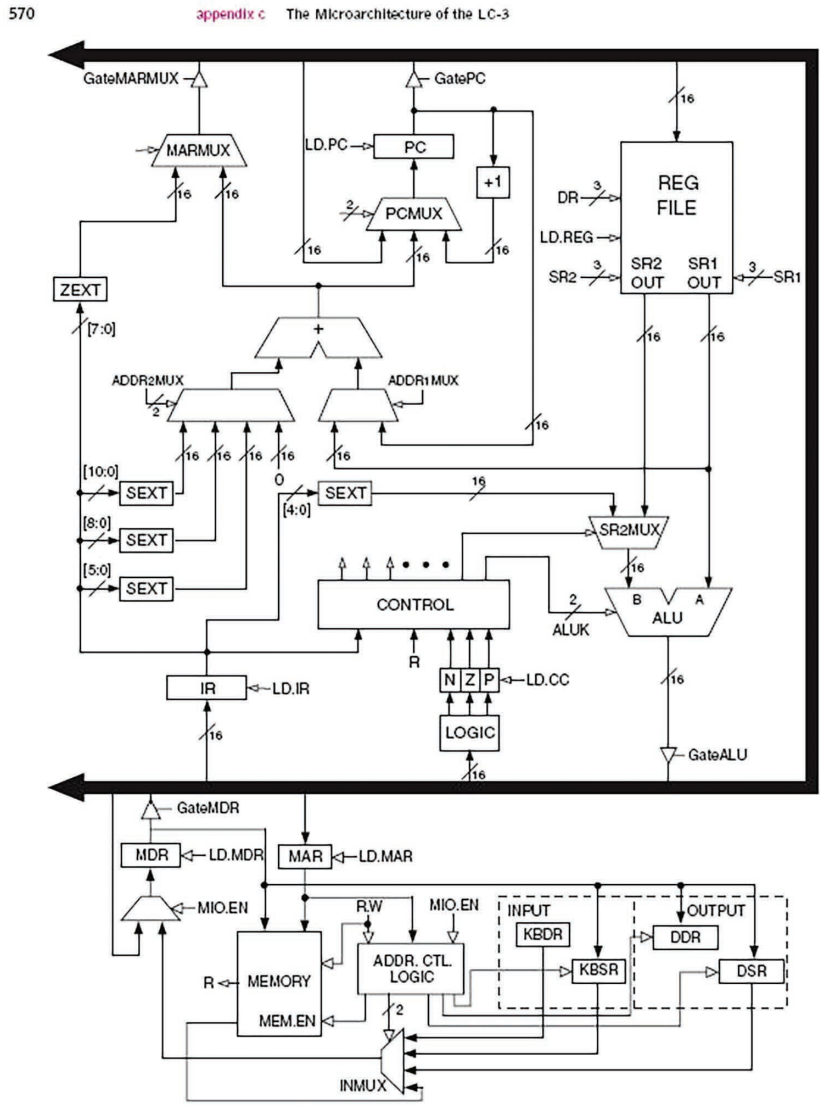


FIGURE 3. The LC-3 data path.

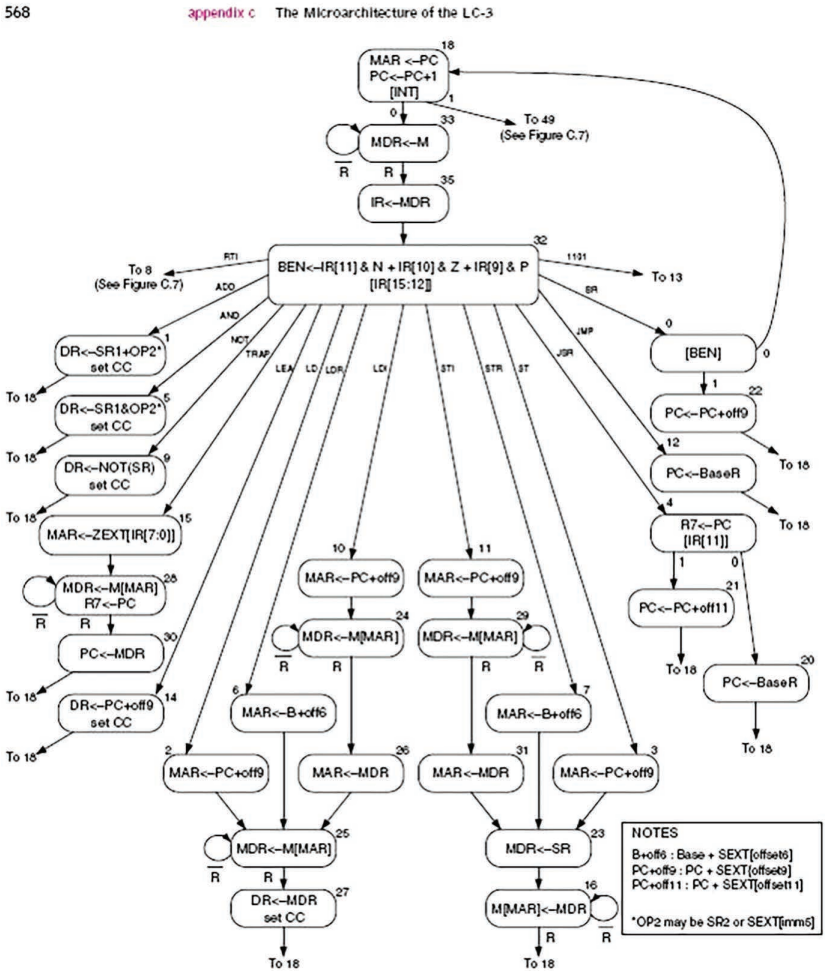


**FIGURE 4.** The LC-3 state machine.

the interaction of application software with the operating system.

## SOME OBSERVATIONS

I have noticed a few trends because of this course. Students exposed to the LC-3 in the freshman course are eager to expand it in the senior course. Some professors augment the compiler course with a C to LC-3 compiler. Others design a small operating system that can run on a slightly modified LC-3, or implement the LC-3 in their VLSI course. Concerning freshmen, they can handle serious meat, they do not need glitz, and they revel in deeper understanding.

About education: engineering education is about design. There is no substitute for the student designing something that does not work, then debugging it, fixing it, and seeing a working product that the student created. Furthermore, abstractions are great after the student understands what is being abstracted. Finally, after almost 60 years in the trenches, I believe a simple concrete model makes a big difference. We can think about beautiful abstractions, but we are not freshmen seeing this for the first time. Ergo, my LC-3 ISA and the motivated bottom-up approach.

## REFERENCE

1. Y. N. Patt and S. J. Patel, *Introduction to Computing Systems: From Bits and Bytes to C/C++ and Beyond*, 3rd ed. New York, NY, USA: McGraw-Hill, 2000.

**YALE N. PATT** is a professor of electrical and computer engineering, Ernest Cockrell, Jr. Centennial Chair in Engineering, and distinguished university teaching professor at The University of Texas at Austin, Austin, TX 78712 USA. Contact him at patt@ece.utexas.edu.