THE UNDERGRADUATE CURRICULUM IN COMPUTER ARCHITECTURE

The discipline of computer science has greatly expanded in breadth and depth since a joint Computer Society-ACM task force wrote the curriculum guidelines in 1991. New fields such as object-oriented programming, networks, parallel computing, and visualization have broadened the modern curriculum. Moreover, many topics included in the 1991 curriculum now require teaching in greater depth because of recent developments. The developments affecting the computer science curriculum have similarly affected computer architecture courses.

Although definitions of computer architecture vary, the term usually refers to the way the elements of a computer relate to each other. College courses generally treat computer architecture from the programmer's viewpoint—that is, the idealized or abstract view of a computer. A computer's implementation, or organization, is hidden from the programmer, but it would be wrong to entirely divorce architecture and implementation because each exerts a powerful influence on the other. Many courses cover both architecture and organization.

The dramatic growth in the body of knowledge making up computer science has forced academics to justify the inclusion of their particular subdisciplines in the curriculum. Nowhere is this pressure greater than in the teaching of computer architecture. Indeed, some of my colleagues have suggested that computer architecture is no longer relevant to the needs of today's computer scientist, and that its traditional place in the curriculum should be allocated to more important, software-oriented subjects. By contrast, this article makes a case for including a broad-based, introductory computer architecture course early in the student's studies.

Goals and approaches

Computer architecture is not taught in an academic vacuum. Universities are devoted to education, research, and the maintenance of academic standards. Consequently, teaching computer architecture cannot be just a matter of training engineers to design chips. It must also provide students a solid grounding in fundamental concepts to prepare them for a career that may span four decades or more in a rapidly changing industry.

Computer science courses have different

0272-1732/00/$10.00 © 2000 IEEE
A significant choice facing teachers is whether to use a bottom-up or a top-down model.

Aims and characteristics. For example, courses in computer architecture, operating systems, and languages and compilers differ from courses in programming, databases, and object-oriented programming. Students taking the latter courses expect to use the skills they acquire in the classroom as soon as they graduate. However, most students taking computer architecture will probably not become practitioners in that field. Consequently, they need not follow a rigid curriculum as long as they cover certain core concepts.

Variations in institutions and student programs make it difficult to impose a standard computer architecture curriculum. Moreover, it would be impossible to cover all the legitimate topics within an acceptable period (three to five years). Therefore, faculty who construct degree programs must carefully select what they include.

The curriculum must provide an overview of computer architecture and teach students the operation of a typical von Neumann machine. It must highlight the important issues facing today's designers and give students the tools they will need to carry out research. A secondary aim should be to reinforce topics common to other areas of computer science. For example, I use register indirect addressing to reinforce the concept of pointers in C.

A significant choice facing teachers is whether to use a bottom-up or a top-down model. A first course in computer architecture might follow a bottom-up sequence: gates, circuits, buses, CPU, assembly language. This approach lets students take small steps as they build a conceptual model of a computer component by component. Unfortunately, with the bottom-up approach, students don't see their ultimate goal as they progress through the sequence of topics. For example, a student learning how gates work does not appreciate their relevance to a computer's operation.

The top-down approach begins with the CPU, subdivides it into functional blocks such as ALUs and control units, and decomposes these blocks into gate-implemented registers, adders, and buses. This approach provides the "big picture" first, a method well supported by software engineering. However, the top-down approach is probably less intuitive than the bottom-up approach. Both teaching models have their supporters and detractors. In practice, which is more effective depends more on the teacher's skills than on the model's academic merit. In my own courses, I provide a rapid top-down overview and then more detailed bottom-up coverage.

History

The history of computer architecture education is as interesting as the history of the computer industry itself. Academics have had to keep pace with developments in a discipline that is growing at an unprecedented rate. In traditional fields such as mathematics and physics, developments took place in universities, and industry followed suit. In computer architecture, industry has led many developments, and universities have followed commercial trends. As a result, some interesting differences exist between the university and industrial viewpoints of the computer architecture curriculum.

Computer architecture courses fall into six broad historical eras: mainframe, minicomputer, early microprocessor, late microprocessor, RISC (reduced instruction set computer), and post-RISC. The mainframe era predated the microprocessor revolution of the early 1970s. Courses in this era concentrated on large IBM mainframes and similar systems. Students had little direct, practical experience of such computers.

By the mid-1970s, computer science departments were springing up in universities throughout the world. Many departments bought low-cost minicomputers, and courses based on popular machines such as Digital Equipment Corporation's PDP-11 began to appear. In an era when many of today's software tools had not been invented, these courses had to concentrate on assembly language programming.

The advent of the microprocessor created
a demand for new architecture courses. Courses of the early microprocessor era took a minicomputer approach to microprocessor technology. In fact, some courses were nothing more than existing minicomputer courses rapidly edited to suit the new 8-bit, 8080-vintage microprocessors. Such courses were poorly constructed because the microprocessor had a different use than the minicomputer. Software development tools were still few and far between, and the microprocessor was effectively a sophisticated programmable control element.

By the 1980s, microprocessor systems were well established in the home, industry, and academia. Computer architecture courses often centered on a specific microprocessor and its particular niche. These courses provided an excellent introduction to primitive microprocessor architecture but sometimes ignored general computer science developments. Indeed, some computer scientists regarded the microprocessor as a regressive influence that turned back the clock of computer science. For example, memory management systems had been used in mainframes and minicomputers since the early 1960s, but microprocessor-based courses often completely ignored this aspect of computer architecture.

The hostility shown by some computer scientists to the microprocessor was inevitable. By the 1970s, the mainframe had reached a high degree of sophistication with its virtual memory, advanced operating systems, and long word lengths. Computer scientists regarded the new microprocessor as little more than a low-cost logic element. The 8-bit microprocessor lacked the characteristics they had come to expect of a computer—for example, complex addressing modes and virtual memory. Electrical engineers didn’t share this skepticism and were delighted with a device they could use to build powerful, low-cost controllers. Even more, they were overjoyed to get back the device they had invented, which they felt computer scientists had hijacked.

As time passed, it became possible to put more active devices on larger chips. Microprocessors of the early 1980s provided more registers, complex addressing modes, memory management, and floating-point coprocessors. Architecture courses of this era focused on the 32-bit, CISC (complex instruction set computing) microprocessors of the Motorola 68000 and Intel 80486 generations. At this point, the microprocessor world had caught up with the mainframe world, and advanced CISCs had architectures comparable to earlier, powerful mainframes.

In the late 1980s, interest in microcomputer architecture broadened. Unlike microprocessor research in the 1970s, which concentrated on getting more gates on a chip, research in the 1980s developed on many fronts. Some researchers worked on ultrafast computers using gallium-arsenide technology, others developed RISC architectures, and others constructed complex architectures with sophisticated memory management mechanisms to support powerful operating systems.

Courses developed in the late 1980s began to incorporate the new RISCs. However, the RISC processor soon faded as a major topic in its own right because designers were applying RISC techniques to almost all processors. The pure RISC processor effectively became a thing of the past in the late 1980s.

By the late 1990s, microprocessor developers were more concerned with performance enhancement than instruction set design. Research, articles, and curricula concentrated on such topics as instruction set parallelism, branch prediction mechanisms, speculative execution, and cache memories.

The divergence between educational and industrial practice widened during the nineties. The slogan “Intel Inside” dominated the personal computer world, and virtually all students had access to PCs based on Intel processors or Intel-like processors from AMD. Nevertheless, relatively few computer architecture courses modeled themselves on Intel processors, probably because the Pentium’s architectural complexity made it difficult for students to comprehend.

Instead, many courses used hypothetical
teaching machines, processors such as the M68000, or RISCs such as the MIPS and Sparc processors. I find the M68000 an ideal teaching vehicle because of its architecture (including exception handling) and interface. Figure 1 illustrates the structure of the M68000 processor viewed as a teaching machine.2

An interesting aspect of the development of computer architecture courses is that high-level material moves down in the curriculum in successive years. That is, material regarded as PhD level one year becomes master's level the next year and undergraduate level the year after. For example, serial interface devices, originally a second-level course topic, now appear in introductory courses.

It is instructive to look at the topics covered in depth in particular courses. For example, hardware-related courses in the 1970s concentrated heavily on the fine detail of device characteristics such as fan-in and fan-out in digital circuits. As these courses developed, their focus changed from device to subsystem, and they emphasized the design of complex digital circuits such as fast adders and multipliers. Apart from specialist courses, today’s courses have dropped these details and instead look in depth at cache memory, branch prediction, and speculative execution.

Perspectives on computer architecture

We can see how the notion of computer architecture has evolved by looking at how academics write about it. In 1989, Dasgupta3 called computer architecture the “art, craft, and science of the design of computers.” He viewed computer architecture in terms of three hierarchical levels of abstraction: exoarchitecture, endoarchitecture, and microarchitecture.

Exoarchitecture refers to the higher-level aspects of computer architecture such as data types, instruction formats, instruction set, and addressing modes. Endoarchitecture refers to the computer's internal organization, encompassing the performance of the computer's principal units, component connections, and information flow control. The endoarchitecture describes a processor at the level of its functional units (registers, ALU, control circuits, and so on).

The microarchitecture realizes the operations carried out by the endoarchitecture. For example, an endoarchitecture describes what an ALU does, whereas a microarchitecture describes how the ALU does it. An exoarchitecture is the programmer's abstract view of a computer, which is implemented by an endoarchitecture, which in turn is realized by the execution of microprograms running on a microarchitecture. Dasgupta’s view of computer architecture fits in well with that of computer scientists most familiar with mainframes and minicomputers.

Another writer, van de Goor,4 introduced
his computer architecture text by quoting what
Amdahl said in 1964: "The architecture of a
computer system can be defined as its func-
tional appearance to its immediate users." 
Assuming that Amdahl regards the immediate
user as the assembly-language programmer, this
definition of computer architecture corre-
sponds to Dasgupta's exoarchitecture. Build-
ing on Amdahl's definition, van de Goor
described what he thought constituted a good
architecture. He maintained that the funda-
mental characteristics of a good architecture
are consistency, orthogonality, propriety, par-
simony, transparency, and open-endedness.

Consistency is an architecture's freedom from
irregularity—that is, the architecture does
similar things in a similar way. Van de Goor
argued that a system is consistent if partial
knowledge of the system permits you to pre-
dict other things about it. For example, by
knowing how a CPU responds to one type of
exception, you can predict how it responds to
other exceptions. Inconsistency makes it dif-
ficult to write assembly language programs
and to produce efficient compilers. Inconsis-
tency often results from the designer's wish to
provide as many instructions as possible.

Van de Goor subdivided consistency into
three components: orthogonality, propriety,
and parsimony. Consider the set of addressing
modes and the set of computer operations.
These two sets are orthogonal if the instruc-
tions are not related to the addressing modes
and an instruction can take any addressing
mode. For example, the M 68000 provides an
ADD memory, register and an ADD register,
memory instruction, but the only legal
exclusive OR instruction is EOR register, mem-
ory. In this case, addressing modes and instruc-
tions are not orthogonal because the range of
addressing modes is instruction-dependent.

Propriety means that the architecture's fea-
tures are strictly necessary to its function, and
that it has no extraneous features. For exam-
ple, a first-generation 8-bit microprocessor
had an interesting bug. When the micro-
processor executed certain instructions one
after the other, the computer didn't behave as
expected. Programmers had to break up the
sequence by inserting a no-operation instruc-
tion between the offending instructions. This
architecture violates the propriety principle,
because the NOP instruction is not necessary
to the intended application's correct operation
(although it is necessary to the computer's cor-
rect operation).

Parsimony is a special case of propriety; it
means that the architecture includes no instruc-
tions that do not perform useful func-
tions. Such instructions clutter up silicon real
estate, reduce the manufacturing yield, and
increase testing time. The desire to achieve
parsimony was a driving force behind the
development of the RISC architecture.

Transparency means that the implementa-
tion of a low-level function does not affect
the operation of a higher-level facility. That is, the
microarchitecture does not influence the oper-
ation of the endoarchitecture. A transparent
design ensures that implementation deci-
sions— for example, whether to implement a
computer's control unit by hardwiring or by
microprogramming— do not affect the archi-
tecture itself.

Open-endedness means that an architecture
is capable of expansion and development. For
example, a microprocessor might use an emu-
lator trap that forces a call to the operating
system when the processor encounters a new
opcode. Thus, programmers can employ a
new instruction that is not part of the proces-
sor's existing instruction set because the oper-
ating system can emulate it.

Unlike Dasgupta and van de Goor, Kain looked
at computer architecture from the point
of view of the high-level-language programmer.
He concentrated on the naming of data struc-
tures in high-level languages, how the pro-
grammer maps these structures to the target
architecture, and how the architecture access-
es them in physical memory. Earlier, comput-
er architecture had moved from the level of the
building block (endoarchitecture) to the level
seen by the assembly language programmer
(exoarchitecture). Kain pushed it to an even
higher level of abstraction—the high-level lan-
guage. Kain's approach, however, had little last-
ing effect on the undergraduate curriculum.

The rise of the RISC architecture dramatical-
ly changed the direction of computer archi-
tecture education in the 1980s. By
abandoning the so-called complex instruction
architectures, academics and semiconductor
manufacturers rewrote the rules of computer
design. The simplification of real architectures
reinvigorated the architecture curriculum by

MAY-JUNE 2000 17
Students should not regard the computer as just a black box that executes programs by magic.

making teaching much easier. This period saw a growth in the number of texts on RISC processors, although the interest in RISC architecture as such rapidly declined. The introduction of the RISC affected the curriculum by moving the focus of attention to pipelining and instruction set parallelism.

The most significant event in computer architecture education during the 1990s was the publication of Hennessy and Patterson's Computer Architecture: A Quantitative Approach. Although intended for graduate courses, this book has become the first standard text in the subject and has strongly influenced courses at all levels. Earlier texts often concentrated on computer structure and assembly language and had little to say about performance (throughput), which was reserved for research papers in journals and conference proceedings. In contrast, Hennessy and Patterson stressed the relationship between computer architecture and computer performance. The authors have since complemented their classic text with an undergraduate text, Computer Organization and Design.

If Hennessy and Patterson's text had concentrated only on computer performance, it would have become a minor footnote in the history of architecture texts. However, the authors are two of the most influential, authoritative people in the field. The high level of detail, thorough background information, and helpful "Fallacies and Pitfalls" at section ends have made the text the first choice for many computer architecture teachers. Indeed, one reviewer said, "The only time a professor would recommend another text to his students would be if he wrote the text himself."

Although no serious challenge to Hennessy and Patterson's view of computer architecture has arisen, the nature of textbooks may be changing. In 1998, Shriver and Smith published a book that may prove to be a significant step in the development of computer architecture courses. Shriver and Smith do not redefine computer architecture, but they repackage it. Their book covers the development of a specific device, AMD's K6 3D processor. The authors go into considerable detail about the processor and its interface to the PC. They particularly examine the decisions made by the chip's designers.

What makes Shriver and Smith's book especially interesting is its multimedia approach. The book is part of a comprehensive package that also includes a CD containing background reading, video clips of engineers discussing the processor, a hypertext version of the book, and logic/CPU simulators. Shriver and Smith will probably not have the same impact that Hennessy and Patterson had on the definition of the subject, but they may have an equal impact on the way it is taught.

Should computer architecture remain in the curriculum?

As I have mentioned, the growth of computer science has put pressure on course designers to remove old material to make room for the new. Some have suggested that architecture is a prime candidate for pruning. However, the computer lies at the heart of computer science—without it, computer science would be a branch of theoretical mathematics. Students should not regard the computer as just a black box that executes programs by magic. In addition to this philosophical consideration, there are practical reasons for computer architecture's continuing relevance in today's world.

Computer architecture has important implications for the computer professional. Suppose a graduate enters the industry and is asked to select the most cost-effective computer for use throughout a large organization. Understanding how the elements of a computer contribute to its overall performance is vital: Is it better to spend $50 on doubling the cache size or $100 on increasing the clock speed by 50 MHz?

We cannot divorce computer architecture entirely from software. Most processors are at work not in PCs or workstations but in embedded applications. Designers of multiprocessors and real-time systems must understand funda-
mental architectural concepts and the limitations of commercially available processors. In developing an automobile's electronic ignition system, programmers might write the code in C. But they might have to debug the system with a logic analyzer that displays the relationship between interrupt requests from engine sensors and the machine-level code.

Another reason for teaching computer architecture is that it incorporates a wealth of important concepts that appear in other areas of the computer science curriculum. For example, studying how the computer provides architectural support for high-level languages reinforces aspects of the languages and compilers curriculum. In my course, I show how a microprocessor implements local workspace and parameter passing at the machine level. This reinforces the notions of passing by value and passing by reference in high-level languages. Teaching bus design and arbitration introduces the important topic of fairness versus priority, which is also found in courses on operating systems. Similarly, understanding cache coherence protocols for multiprocessors allows developers to design better cache management mechanisms for Web browsers.

Finally, a computer architecture course offers a starting point for postgraduate study and possible employment at the hardware-oriented end of computing such as data storage system design.

Breadth and depth

Computer architecture education is not just what is taught, but also how it is taught. Traditionally, computer science has been taught in the same way as foreign languages: first the basics of grammar and vocabulary (programming and data structures) and then the use of the language (applications such as AI or graphics).

Some regard this sequence as detrimental because students must digest a large body of information before they appreciate how it all fits together. With this in mind, the 1991 curriculum task force recommended a new approach, “breadth before depth.” This approach would replace the conventional introduction to programming with a more broadly based course that introduces topics such as artificial intelligence and graphics at a much earlier stage.

The same consideration applies to the computer architecture curriculum. Students who continue on to careers in computer design are well served by current architecture courses, which heavily emphasize digital design and instruction set architecture. But the fraction of students that will ever be directly involved in computer design is declining. Computer science is expanding to include new areas such as netcentric computing, multimedia, and visualization. Indeed, several universities in the UK now provide both undergraduate and postgraduate degree programs in these areas. Students in such programs don't see the point of studying computer architecture, and pressure is growing to drop it from the curriculum.

But all students taking computer-related degree programs need some knowledge of computer architecture. Even students whose goals are far removed from computer design should appreciate a computer system's functional components, their characteristics, their performance, and their interactions. Consequently, a first-level, breadth-before-depth course covering the entire computer system should be mandatory for all students in the computing disciplines (see sidebar, next page).

Such a course combines elements of the traditional digital logic course with the introductory architecture course. Its goal is that students gain an appreciation of the computer system from gate level to system level. Students should take the course in the freshman year. To compensate for the lack of depth, students can take an optional laboratory course and higher-level architecture courses.

Computer architecture or computer systems architecture?

Curiously, some computer architecture courses cover the architecture and organization of the processor but make little reference to buses, memory systems, and high-perfor-
Computer technology was once driven by the paperless-office revolution with its demand for low-cost mass storage, sufficient processing power to rapidly recompose large documents, and low-cost printers. Today, the driving force is the multimedia revolution with its insatiable demand for pure processing power, high bandwidth, low latency, and massive storage capacity. This trend has led to important commercial developments in computer architecture—for example, Intel’s MMX instruction set, which provides simple SIMD (single-instruction, multiple-data) parallelism and saturated arithmetic processing.

However, areas other than computer architecture are also feeling the demands of multimedia. Hard disks provide a continuous stream of data, and design techniques for low latency are becoming more important than those for high-speed retrieval. Computer users can tolerate a degraded picture much better than a picture with even the shortest discontinuities. Such demands require efficient track-seeking algorithms, data buffering, and high-speed, real-time error correction and detection algorithms. Similarly, because of thermal effects, today's high areal data densities require frequent recalibration of tracking mechanisms. Some modern disk drives include the AI world's so-called smart technologies, which predict disk failure before it occurs. These developments are as worthy of inclusion in the architecture curriculum as CPU developments.

Some of the most impressive advances in recent years have been in computer graphics.
More and more of the burden of image processing is moving onto graphics chips. Similarly, digital signal processing, once the province of the electrical engineer, is edging into mainstream computing as multimedia systems provide support for audio processing.

Are these technologies reflected in today's computer architecture courses? Generally, they are not. Some educators might argue that these developments are not relevant to the core curriculum and are best dealt with elsewhere. Some might regard the modern graphics chip as no more than a CPU with a microprogrammed instruction set and a large dose of parallelism. But I would answer that the contemporary computer architecture (and organization) course should place greater emphasis on the whole computer system and the new technologies that support multimedia. Some educators might say that the curriculum is already full and that nothing can be omitted. However, many current computer architecture texts emphasize techniques, such as branch prediction, that represent just one small element of the field. Texts and courses could de-emphasize some of these topics to make room for systems-oriented material.

Changes in the classroom

The advent of the personal computer in the 1980s gave students real systems to work with, and new compilers and assemblers made programming relatively easy. By the 1990s, computers could simulate processors, and students could investigate computer architecture by observing the data flow in a machine being simulated. Ever since, undergraduates have used simulators to understand the fundamentals of computer architecture. Postgraduates use them to experiment with computer design. Simulation considerably helps the teaching of computer architecture by enabling students to visualize how a computer operates and to interact with it.

**Reference**

Figure 2 shows a typical logic simulator, which lets students drag and drop gates onto a work area to construct and test circuits. Simulators help teachers cover more material in a course because students can absorb difficult concepts faster, particularly simulators with good visual interfaces. Moreover, with public domain simulators, students can work from home.

The Internet is already having an impact on the way I teach computer architecture. In the past, I might have asked students to describe how cache memories work. Today, I can ask them to conduct research on the Internet to write a comparative report on cache strategies adopted by several major manufacturers. Modern software enables me to construct Web-based graphical animations to illustrate complex topics that students often have difficulty understanding.

Today, opportunities for engaging students in computer architecture education have never been better. Simulators and Web-based tools have created an immensely rich teaching environment. Architectural innovations such as branch prediction and speculative execution (which fascinate my students) and many of the hardware-based advances in multimedia systems have greatly enriched the subject matter.

At the same time, however, computer architecture education faces a dilemma. It must serve two masters. It must serve the greater computing community by providing a broad introduction for students who require only an overview of today’s high-performance personal computers and workstations. Equally important, it must also serve the design community by providing a springboard for students who are going to design tomorrow’s computers.

**References**


**Alan Clements** welcomes comments. His biography and contact information appear in this issue’s Guest Editor’s Introduction.