Engineers, Programmers, and Black Boxes

Bob Colwell

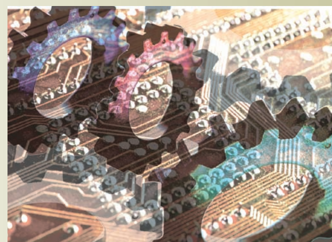
Universities teach engineers all sorts of valuable things. We're taught mathematics—especially calculus, probability, and statistics—all of which are needed to understand physics and circuit analysis. We take courses in system design, control theory, electronics, and fields and waves. But mostly what we're taught, subliminally, is how to think like an engineer.

Behind most of the classes an engineer encounters as an undergraduate is one overriding paradigm: the black box. A black box takes one or more inputs, performs some function on them, and produces one output.

It seems simple, but that fundamental idea has astonishing power. You can build and analyze all engineered systems—and many natural systems, specifically excluding interpersonal relationships—by applying this paradigm carefully and repetitively.

Part of the magic is that the function the black box contains can be arbitrarily complex. It can, in fact, be composed of multiple other functions. And, luckily for us, we can analyze these compound functions just as we analyze their mathematical counterparts.

As part of an audio signal processing chain, a black box can be as simple as a low-pass filter. As part of a communications network, it can be a complicated set of thousands of processors, each with its own local network.



You can build and analyze all engineered systems by applying the black box paradigm.

MARVELS OF COMPLEXITY

Modern microprocessors are marvels of complexity. Way back when, the Intel 4004 had only 2,300 transistors, a number that is not too large for smart humans to keep in their heads. Engineers knew what each transistor did and why it had been placed where it was on the die. The bad news was that they *had* to know; there were no CAD tools back then to help keep track of them all.

But even then, the black box functional decomposition paradigm was essential. At one level of abstraction, a designer could ask whether the drive current from transistor number 451 was sufficient to meet signaling requirements to transistors 517 and 669. If it was, the designer would conceptually leave the transistor level and take the mental elevator that went to the next floor up: logic.

At the logic level, the black boxes had labels like NAND and XOR. The designer's objective at this level was to make sure that the functions selected correctly expressed the design intent from the level above: Should this particular box be a NAND or an AND? There were also subfloors. It's not only possible, it's also a very good idea to aggregate sets of boxes to form more abstract boxes. A set of *D* flip-flops is routinely aggregated into registers in synchronous designs, for example.

Next floor up: the microarchitecture. At this level, the boxes had names like register file, ALU, and bus interface. The designer considered things like bandwidths, queuing depths, and throughput without regard for the gates underlying these functions or the actual flow of electrical currents that was such a concern only a few floors below.

For hardware engineers, there was one more floor: the instruction set architecture. Most computer engineers never design an ISA during their careers—such is the commercial importance of object code compatibility.

For decades now, the prevailing theory has been that to incentivize a buyer to suffer the pain of mass code conversion or obsolescence, any new computational engine that cannot run old code, unchanged, must be at least *N* times faster than anything else available. The trouble with this theory is that it has never been proven to work. At various times in the past 30 years, *N* has arguably reached as high as 5 or 10 (at equivalent economics) without having been found to be compelling.

The x86 architecture is still king. But the latest contender in the ring is IBM's Cell, introduced in February at ISSCC 05. Touted as having impressive computational horsepower, Cell is aimed initially at gaming platforms that may

Published by the IEEE Computer Society

not be as sensitive to the compatibility burden.

Stay tuned—this new battle should play out over the next three years. Maybe computer engineers will get to play out in the sunshine of the top floor after all.

SOFTWARE FOLKS DO IT TOO

The ability to abstract complex things is vital to all of engineering. As with the 4004's transistors, without this ability, engineers would have to mentally retain entire production designs. But the designs have become so complicated that it has been about 25 years since I last saw a designer who could do that. Requiring designers to keep such complex designs in their heads would limit what is achievable, and doing so isn't necessary as long as we wield our black-box abstractions properly.

In the early days of P6 development at Intel, I found it amusing to try to identify various engineers' backgrounds by the way they thought and argued during meetings. My observations went through several phases.

I was intrigued to observe that a group of 10 engineers sitting around a conference room table invariably had a subtle but apparent common mode: They all used the black-box abstraction implicitly and exclusively, as naturally as they used arithmetic or consumed diet Coke. Although these engineers came from different engineering schools, and their degrees ranged from a BS to an MS or a PhD, they implicitly accepted that any discussion would occur in one of two ways—either at one horizontal abstraction layer of the design or explicitly across two or more layers. It was generally quite easy to infer which of those two modes was in play, and all 10 engineers had no difficulty following mode changes as the conversation evolved.

When thinking about this (and yes, I probably should have been paying attention to the technical discussion instead of daydreaming), it occurred to me that the first two years of my undergraduate EE training had sometimes seemed like a military boot camp. In fact, it *was* a boot camp. With the exception of social sciences, humanities, history, and phys. ed., all of our classes were done in exactly this way.

I don't know if we became EEs because we gravitated toward the academic disciplines that seemed most natural to us, or if we just learned to think this way as a by-product of our training. Maybe we just recognized a great paradigm when we saw it and did the obvious by adopting it.

In general, constraints and boundaries are a good thing—they focus the mind.

Microprocessor design teams also have engineers with computer science backgrounds, who may not have gone through an equivalent boot camp. I tried to see if I could spot any of them by watching for less adroitness in following implicit abstraction-layer changes in meetings. I thought I saw a few instances of this, but there's a countervailing effect: CS majors live and breathe abstraction layers, presumably by dint of their heavy exposure to programming languages that demand this skill.

When I began pondering the effect of black-box function-style thinking and programming language abstractions to see if that might distinguish between CS- and EE-trained engineers, I did see a difference. Good hardware engineers have a visceral sense of standing on the ground at all times. They know that in the end, their design will succeed or fail based on how well they have anticipated nature itself: electrons with the same charge they have carried since the birth of the universe, moving at the same speed they always have, obeying physical laws that govern electronic and magnetic interactions along wires, and at all times constrained by thermodynamics.

Even though EEs may spend 95 percent of their time in front of a computer putting CAD tools through their paces (and most of the other 5 percent swearing at those same tools), they have an immovable, unforgettable point of contact with ultimate reality in the back of their minds. Most of the decisions they make can be at least partially evaluated by how they square against natural constraints.

CONSTRAINTS ARE GOOD FOR YOU

You might think such fixed constraints would make design more difficult. Indeed, if you were to interview a design engineer in the middle of a tough morning of wrestling with intransigent design problems, she might well express a desire to throw a constraint or two out the window. Depending on the particular morning, she might even consider jumping out after them.

In general, though, constraints and boundaries are a good thing—they focus the mind. I've come to believe that hardware engineers benefit tremendously from their requisite close ties to nature's own rules.

On the other hand, the CS folks are generally big believers in specifications and writing down the rules by which various modules (black boxes) interact. They have to be—these "rules" are made up. They could be anything.

Assumptions are not just subtly dangerous here, they simply won't work—the possibility space is too large. It's not that every choice a hardware engineer makes is directly governed by nature and thus unambiguous. What functions go where and how they communicate at a protocol level are examples of choices made in a reasonably large space, and there a CS grad's proclivity to document is extremely valuable.

To be sure, some programmers face natural constraints just as real as any the hardware designers see. Real-time code and anything that humans can perceive—video and audio, for example—impose the same kinds of immovable constraints that a die size limit does for a hardware engineer.

I'm not looking for black and white—I'm just wondering if there are shades of gray between EE and CS. My attempt to discern differences between EE and CS grads was simply intended to see if the two camps were distinguishable "in the wild"—to see if that might lead to any useful insights.

Computer science is not generally taught relative to natural laws, other than math itself, which is arguably a special case. I don't know if it should be, or even can be, and it's not my intention to pass a value judgment here.

The CS folks, it seems to me, tend to be very comfortable in a universe bounded only by conventions that they (or programmers like them) have erected in the first place: language restrictions, OS facilities, application architectures, and programming interfaces. The closest they generally come to putting one foot down on the ground is when they consider how their software would run on the hardware they are designing—and that interface is, at least to some extent, negotiable with the EE denizens on the top floor. Absolutes, in the nonnegotiable natural-law sense of what EEs deal with, are unusual to them.

The best engineers I have worked with were equally comfortable with hardware and software, regardless of their educational backgrounds. They had somehow achieved a deep enough understanding of both fields that they could sense and adjust to whatever world view was currently in play at a meeting, without giving up the best attributes of the alternative view.

There is a certain intellectual thrill when you finally break through to a new understanding of something, be it physics or engineering or math—or poetry analysis, for that matter. I always felt that same thrill when I saw someone blithely displaying this kind of intellectual virtuosity.

BOTTOMS UP AND TOPS DOWN

The engineers I know who routinely do this intellectual magic somehow arrived at their profound level of understanding via the random walk of their experiences and education, combined with extraordinary innate intelligence. Can we teach it?

Yale Patt and Sanjay Patel think so. It's a basic tenet of their book, *Introduction to Computing Systems* (McGraw Hill, 2004). On the inside cover, no less a luminary than Donald Knuth says, "People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise, the programs they write will be pretty weird."

There is a certain intellectual thrill when you finally break through to a new understanding of something.

Conversely, people who design computers without a good idea of how programs are written, what makes them easy or hard, and what makes them fail, will in all likelihood conjure up a useless design. I once heard a compiler expert opine that there's a special place in the netherworld for computer designers who create a machine before they know if a compiler can be written for it.

One other data point I'm sure of: Me. I had taken an OS course and several programming language courses and did well at them, but I didn't understand what computer architecture really meant until I had to write assembly code for a PDP-11. My program had to read the front panel switches, do a computation on them, and display the results on the front panel lights.

My first program didn't work reliably, and I spent hours staring at the code, line by line, trying to identify the conceptual bug. I couldn't find it. I finally went back to the lab and stared instead at the machine. Eureka! It suddenly occurred to me that the assignment hadn't actually stated that the switches were debounced, and the PDP-11 documentation didn't say that either. I had simply assumed it.

Mechanical switches are constructed so that flipping the switch causes an internal metal plate to quickly move from one position to a new one where it now physically touches a stationary metal plate. Upon hitting the stationary plate, the moving metal repeatedly bounces up and down until it eventually settles and touches permanently.

Even at the glacial clock rates of the 1970s, the CPU had plenty of time to sample a switch's electrical state during the bounces. Debouncing them in software was just a matter of inserting a delay loop between switch transition detection and logical state identification.

Without an understanding of both the hardware and the software, I'd still be sitting in front of that PDP-11, metaphorically speaking.

There are always tradeoffs. The horizontally stratified way we teach computer systems today makes it difficult for students to see how ideas at one level map onto problems at another.

EVEN GOOD ABSTRACTIONS CAN HURT

If you really want to snow a student under, try teaching computer system design from application to OS to logic to circuits to silicon physics as a series of vertical slices.

In some ways, I think this problem was fundamental to Intel's failed 432 chips from the early 1980s—they were "capability-based" object-oriented systems in which one global, overriding paradigm was present. The system was designed from one point of view, and to understand it you had to adopt that point of view. To wit: Everything—and I do mean everything—was an object in those systems. In some ways, it was the ultimate attempt to systematically apply the black-box paradigm to an entire computer system.

An object in a 432 system was an abstract entity with intrinsic capabilities and extrinsic features. Every object

was protected by default against unauthorized access. If one object (your program, say) wanted access to another (a database, perhaps) your program object had to first prove its bona fides, which hardware would check at runtime. At a software level, this kind of system had been experimented with before, and it does have many appealing features, especially in today's world of runaway viruses, Trojans, worms and spam.

But the 432 went a step further and made even the hardware an object. This meant that the OS could directly look up the CPU's features as just another object, and it could manipulate that object in exactly the same way as a software object.

This was a powerful way of viewing a computing system, but it ran directly contrary to how computer systems are taught. It made the 432 system incomprehensible to most people at first glance. There would be no second glance: Various design errors and a poor match between its Ada compiler and the microarchitecture made the system almost unusably slow. The 432 passed into history rather quickly.

If the design errors had been avoided, would the 432 have taken hold in the design community? All things considered, I don't think so: It had the wrong target in the first place. The 432 was intended to address a perceived looming software production gap. The common prediction of the late 1970s was that software was too hard to produce, it would essentially stop the industry in its tracks, and whatever hardware changes were needed to address that gap were therefore justified.

With a few decades of hindsight, we can now see that the industry simply careened onward and somehow never quite fell into this feared abyss. Perhaps we all just lowered our expectations of quality to "fix" the software gap. Or maybe Bell Labs' gambit of seeding universities in the 1970s with C and Unix paid off with enough programmers in the 1980s. Whatever the reason, the pool of people ready to dive into Ada and the 432's new mindset was too small.

New paradigms are important. Our world views make it possible for us to be effective in an industry or academic environment, but they also place blinders on us.

In the end, I concluded that it wasn't a matter of identifying which world view is best—EE or CS. The best thing to do is to realize that both have important observations and intuitions to offer and to make sure the differences are valued and not derided.

Society at large should go and do likewise. ■

Bob Colwell *was Intel's chief IA32 architect through the Pentium II, III, and 4 microprocessors. He is now an independent consultant. Contact him at bob.colwell@comcast.net.*

The 30th IEEE Conference on
Local Computer Networks (LCN)
Sydney, Australia – November 15-17, 2005
Call for Papers

http://www.ieeelcn.org

The IEEE LCN conference is one of the premier conferences on the leading edge of practical computer networking. LCN is a highly interactive conference that enables an effective interchange of results and ideas among researchers, users, and product developers. We are targeting embedded networks, wireless networks, ubiquitous computing, heterogeneous networks and security as well as management aspects surrounding them. We encourage you to submit original papers describing research results or practical solutions. Paper topics include, but are not limited to:

• *Embedded networks* • *Wearable networks* • *Wireless networks* • *Mobility management*
• *Networks to the home* • *High-speed networks* • *Optical networks* • *Ubiquitous computing*
• *Quality-of-Service* • *Network security/reliability* • *Adaptive applications* • *Overlay networks*

Authors are invited to submit full or short papers for presentation at the conference. Full papers (maximum of 8 camera-ready pages) should present novel perspectives within the general scope of the conference. Short papers are an opportunity to present preliminary or interim results and are limited to 2 camera-ready pages in length. All papers must include title, complete contact information for all authors, abstract, and a maximum of 5 keywords on the cover page.

Papers must be submitted electronically. Manuscript submission instructions are available at the LCN web page at http://www.ieeelcn.org. Paper submission deadline is **May 10, 2005** and notification of acceptance is **July 28, 2005**.

General Chair:
Burkhard Stiller
University of Zürich, and
ETH Zurich, Switzerland
stiller@tik.ee.ethz.ch

Program Chair:
Hossam Hassanein
Queen's University
Canada
hossam@cs.queensu.ca

Program Co-Chair:
Marcel Waldvogel
University of Konstanz
Germany
marcel.waldvogel@uni-konstanz.de