

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