Technology Innovation
(Hardware, Software, and What You Can Learn from Startup Companies)

Bruce Jacob, Ph.D.

Keystone Professor & Director of Computer Engineering,
Electrical & Computer Engineering Dept.
University of Maryland
College Park, Maryland
http://www.ece.umd.edu/~blj/

Formerly
Chief Engineer & System Architect,
Priority Call Management (now uReach Technologies)
Wilmington, Massachusetts

OUTLINE

1. Introduction: To Mandate Creativity
2. A Diversion: Software Guys vs. Hardware Guys

   Characteristics of Software as an Engineered Product

   Characteristics of Hardware as an Engineered Product

   And Ne'er the Twain Shall Meet?
3. The Motivation and Reward of Engineers

   A Page from the Startup Story

   A Structure for Identification and Reward: Venture Capitalism
4. The Value of Design

   A Tale of Three Design Flows

   Emergent Paradigms: Manufacturing as a Service, Design as an End Product

KEY WORDS

Engineering Design, Managing Engineers, Design Principles

ABSTRACT

Innovation is highly sought after: it is extremely valuable, extremely rare, and extremely difficult to do. An executive desiring an innovative company should first realize that innovation is the product of individuals, not organizations. Thus, one obvious approach to creating an innovation-based company is to focus on rewarding innovative individuals—the decision-makers in engineering—in the same manner as typical companies are currently structured to reward decision-makers at the executive level and in sales, i.e. by their contribution to the bottom line.

Technology innovation can take multiple forms, from new products to new ways to make products, and the characteristics of hardware and software systems point to opportunities such as the comprehensive use of CAD tools for embedded-systems design. In the new pace of business, increased competition at the manufacturing level actually enables a company to spend more time, effort, and money on approaches that will lead to innovation.

INTRODUCTION: TO MANDATE CREATIVITY

Paul Graham, co-founder of 1990s startup Viaweb and co-developer of its software, which now powers Yahoo! Stores, motivates the inherent tension and challenge of technology innovation in a typical corporate setting (Graham, 2004):

> Big companies can develop technology. They just can't do it quickly. Their size makes them slow and prevents them from rewarding employees for the extraordinary effort required. So in practice big companies only get to develop technology in fields where large capital requirements prevent startups from competing with them, like microprocessors, power plants, or passenger aircraft. And even in those fields they depend heavily on startups for components and ideas.
>
> In the right kind of business, someone who really devoted himself to work could generate ten or even a hundred times as much wealth as an average employee. A programmer, for example, instead of chugging along maintaining and updating an existing piece of software, could write a whole new piece of software, and with it create a new source of revenue.
>
> Companies are not set up to reward people who want to do this. You can't go to your boss and say, "I'd like to start working ten times as hard, so will you please pay me ten times as much?" For one thing, the official fiction is that you are already working as hard as you can. But a more serious problem is that the company has no way of measuring the value of your work.
>
> A company that could pay all its employees so straightforwardly [as its executives and salesmen, based upon revenue generated] would be enormously successful. Many employees would work

> harder if they could get paid for it. More importantly, such a company would attract people who wanted to work especially hard. It would crush its competitors.
>
> That's the real point of startups. Ideally, you are getting together with a group of other people who also want to work a lot harder, and get paid a lot more, than they would in a big company. And because startups tend to get founded by self-selecting groups of ambitious people who already know one another (at least by reputation), the level of measurement [of individual skill and contribution] is more precise than you get from smallness alone. A startup is not merely ten people, but ten people like you.
>
> [pages 101, 96–97, 99, from the essay "How to Make Wealth"]

Your job as an executive is to figure out how to turn your large company into an innovation machine—to generate new ideas and technology well and relatively often. To anyone who creates, the notion itself should smack of incredible hubris: innovation is the creation of really useful stuff, and so to declare one's company "innovation oriented" is essentially to mandate creativity. However, as any creative person will attest, creativity stubbornly refuses to be mandated, else the phrase "writer's block" would hold no meaning whatsoever. So how, then, could one possibly center one's business model around something as slippery as innovation and retain any hope of staying alive?

Let us get a few facts out onto the table to begin with. It would be good to bear these in mind; they are the obstacles that stand in the way of the executive or manager who desires his organization to innovate. Overcoming these obstacles is the focus of this chapter.

- Technology innovation is ridiculously difficult; it requires extraordinary effort, dedication, time, and focus of attention on the part of extremely talented individuals.
- Technology innovation is driven by *individuals*, not organizations: if you do not have extremely good people, it simply will not happen.
- Innovation usually but not always comes from engineers. Good ideas do come from all sources, but even when an idea originates outside of engineering, it is the engineer who makes the idea work. Therefore the article will focus on understanding and motivating the individual engineer.
- The majority of engineers tend to be good at only one of the following two skills: their job and promoting themselves. Some are good at neither, and it is the rare case in which an individual is good at both. Thus, in almost every single engineering organization, from

the corporate R&D department to the university, there is a rough inverse relationship between people's salaries and their contribution to the organization's bottom line.

- The disparity of salaries and their lack of correlation to skill is quite well known to the engineers themselves. This generates non-vocalized morale problems and causes the extremely talented individuals to leave large companies and seek their fortune elsewhere, typically in startup companies where, despite the 10% rule of thumb for startup success, these individuals still have a far better chance of being rewarded at a level commensurate with their skills.

These may be hard truths to admit, but this is the reality in industry today. By definition, big companies tend not to innovate. The inability of managers to identify and reward talent causes most employees to do above average but far less than spectacular work—and without spectacular contributions, innovation will not occur. The most innovative individuals gravitate toward startups where, unlike the environments in large corporations, they tend to be rewarded in proportion to their accomplishments.

There exists a powerful opportunity to turn this reality to one's advantage: the flip side of the trend is that any large company that *does* learn how to identify, retain, motivate, and reward its best engineers would position itself successfully as an innovator and would most surely dominate its industry. Witness Apple's original emergence and recent re-emergence as a technology-innovation company. Witness Google's rise; their treatment of their engineers is legendary.

So how can an organization become innovative? How can an executive successfully mandate creativity? Though one may not be able to guarantee creativity, there is much an executive can do to foster it, which is certainly an attainable goal and quite possibly "the least one can do" as the person in charge, as this is something of a *sine qua non*—an essential ingredient to the stew. The basics:

- Understand what motivates your engineers, and foster their creativity.
- Support the extraordinary efforts of innovative individuals by rewarding behavior appropriately, in the same manner as a startup company.

It comes down to recognizing and rewarding good design. Good design, the heart of innovation, is infinitely more valuable than money … you cannot *buy* your way into a good design just as you cannot *buy* the ability to innovate. The only way to buy innovation is to purchase an innovative company or hire an accomplished design team, and even then you risk losing all the

innovative individuals you just hired if your corporate environment fails to reward them. Many purchased companies experience so-called "brain drains" as soon as the papers are signed; the purchasing company acquires the innovative company's name and technology but not its innovators.

To be sustainable, technology innovation must be grown locally. To accomplish your goals of becoming *and remaining* a technology-innovation company, you must foster an environment that enables and rewards innovation and good design.

A DIVERSION: SOFTWARE GUYS VS. HARDWARE GUYS

Do not make the mistake of believing software engineers and hardware engineers to be merely interchangeable "engineers" with different job descriptions. The two personalities are quite different; the jobs they do (problems they solve) hardly have a thing in common, though their typical job *descriptions* certainly belie that fact; and it it not clear whether the personalities are molded by the problems they solve, or the personalities of the individuals attract them to solving certain problems.

Either way, if you expect to motivate and reward them, you had better understand them.

Traditionally, software engineers are Computer Scientists and hardware engineers are Electrical Engineers, and so the division begins in the education system and is continued in the workplace. In academia, these topics are taught in separate departments which are usually in separate colleges (i.e., they belong to different organizational & administrative hierarchies), and the different departments never interact. In industry, they are different groups usually housed at opposite ends of the building/campus, and they never interact.

The *reason* they do not interact is often animosity: in both industry and academia (note: I was a software guy in industry then went to academia and evolved into a hardware guy), software guys hate hardware guys, and vice versa. Software people consider hardware people unibrow neanderthals who believe that clubs, sharpened rocks, and C programming are paragons of high-tech. Hardware people consider software people flighty, self-satisfied pansies (think Harvey Korman and Andréas Voutsinas as the bickering French noblemen in *History of the World, Part I*) who complain far out of proportion to the useful work they do.

The funny thing is that both sides are right, but for the wrong reasons. The problem is that both sides are comparing apples to oranges—each evaluates the other against his own standards instead of evaluating the other against *the other's* standards. This is almost always the type of

cognitive disconnect and misunderstanding that occurs when a person dramatically underestimates the value and difficulty of another person's work.

Make no mistake whatsoever: hardware people and software people are both extremely smart; they both solve extremely difficult and valuable problems. But they are apples and oranges; for instance, if you used a software approach to solve a hardware problem, you would get embarrassingly poor results—literally. You would probably be ridiculed (in good humor) by your peers for failing so miserably to address the problem. Similarly, if you used a hardware approach to solve a software problem, you would probably be ridiculed (in good humor) by your peers for failing so miserably to address the problem.

So what is going on here? At a high level, both build extremely complex systems, so it must be the qualities of those systems that creates the divide.

Characteristics of Software as an Engineered Product

First, let us develop a quick understanding of the problem area in which software engineers work: they build extremely large systems of interconnected functions, in which only a relative handful of the functions interact at a given time—meaning at any given point, only a fraction of the system (code) is operative.

Software engineers are held to a standard of correctness that is between 99% and 99.9%—the general rule of thumb is that any given software program has a bug every 100–1000 lines of code, and this is considered an acceptable level of reliability. The truth is that the sheer complexity of these systems, and the richness of the functions and their interactions makes it *extremely* difficult to get a software product even to this level of correctness.

The software engineer's task is inherently creative—the software engineer is tasked to generate new concepts, new features, new behaviors … system-level capabilities that did not exist previously … and then to realize them in code. Often the hardest part is mapping these things, which rarely have words for adequate description, into existing software paradigms that inevitably lack appropriate power of expression.

Characteristics of Hardware as an Engineered Product

Let us extend this understanding to describe what hardware engineers do. Hardware engineers build extremely large systems of interconnected components, in which almost all components interact at a given time—meaning at any given point, almost *all* of the system is operative. If software is a gigantic, complex system of interacting functions, then hardware (e.g.,

computer hardware) is the equivalent of one, single, incredibly enormous function. All of it operates with itself, all the time—so, for example, if **any** of it is broken, the whole thing fails.

If hardware engineers were held up to the same standard of correctness as software engineers (one bug in every 1000 lines of code), then nearly all hardware systems would be completely inoperative. Hardware simply does *not* work if it is only 99.9% correct—it doesn't work if it is 99.999% correct. Semiconductor chips today have over 1 billion parts in them, and a single bug can bring the entire system down, because all of the hardware system is being used, all the time. Forget one bug in a hundred or a thousand; a hardware engineer doesn't sleep well until the bugs are one in a million or better.

And Ne'er the Twain Shall Meet?

So that is the difference: the hardware engineer's problem domain is design reliability, design correctness; the hardware engineer is paid to do it *right*. The software engineer's problem domain is functionality; he is paid to do something *cool* and implement it passably well. The one is an engineer, a scientist; the other is an artist—*cf.* Hackers & Painters (Graham, 2004).

Both require smarts in enormous quantities, both contribute to a company's bottom line, both are equal in value. The successful executive will manage to get each side of engineering to understand they will each benefit individually working together. The key is to recognize that both classes of individuals are extremely competitive, mostly with themselves (i.e., internally driven), and they thrive on solving difficult problems. Both hardware engineers and software engineers consider it far more personally rewarding to solve an "impossible" problem than to do just about anything else. It is mountain-climbing for the techie: a guy who scales a mountain everyone else said was impossible to climb returns with an enormous feather in his cap. Ditto with engineers solving problems that nobody else could. All it takes to get a really good engineer to work on a problem is for him to see how difficult it is: technical challenges to engineers are catnip to cats or bug-zappers to mosquitos.

Importantly, technical challenges also happen to be significantly more engrossing to a hardware/software engineer than bashing the software/hardware department.

Bottom line: these guys will work together famously if the problem is hard and the reward is significant. Innovation at the technical level is by definition a hard problem, so figuring out how to motivate ones engineers is not the hard part. The real issue for the executive is that of reward —how do you convince your engineering staff that it is worth their while to go above and beyond

on a daily basis? How do you convince the exceptionally talented that it is worth their while to work for you instead of themselves?

Now that the question (a two-sentence re-statement of the chapter's Introduction) is put that way, it almost answers itself: you convince talented engineers to work exceptionally hard for you by structuring it so that they *are* working for themselves.

THE MOTIVATION AND REWARD OF ENGINEERS

The story of Apple's *Graphing Calculator* application provides good insight: the software was developed by two contractors whose projects were terminated prematurely but who nonetheless remained for months afterward, unpaid, sneaking into the facility to finish (Avitzur, 2004). Here is some insight into the engineers' motivation:

> Why did Greg and I do something so ludicrous as sneaking into an eight-billion-dollar corporation to do volunteer work? Apple was having financial troubles then, so we joked that we were volunteering for a nonprofit organization. In reality, our motivation was complex. Partly, the PowerPC was an awesome machine, and we wanted to show off what could be done with it; in the *Spinal Tap* idiom, we said, "OK, this one goes to eleven." Partly, we were thinking of the storytelling value. Partly, it was a macho computer guy thing—we had never shipped a million copies of software before. Mostly, Greg and I felt that creating quality educational software was a public service.
>
> I view the events as an experiment in subverting power structures. I had none of the traditional power over others that is inherent to the structure of corporations and bureaucracies. I had neither budget nor headcount. I answered to no one, and no one had to do anything I asked. Dozens of people collaborated spontaneously, motivated by loyalty, friendship, or the love of craftsmanship. We were hackers, creating something for the sheer joy of making it work.

The story might well be read by non-engineering types with a kind of horror; I don't know. It is certainly read by engineers as a modern *Robin Hood*, as an example of heroes to honor and emulate. Engineers want to create superbly beautiful things; the less encumbered by bureaucracy, the better. As the story suggests, engineers will work ridiculous hours and go to absurd lengths to ensure that what they create is enviably good design; all they need is the proper motivation—and in this instance, the work itself was its own motivation.

A Page from the Startup Story

Additional insight comes from the rise and fall of tech startups. Consider the normal life-cycle of a company that starts out as a technology innovator and, in this case, succeeds. Normally, startups begin as a small group of like-minded individuals who solve a problem that other people want solved enough to pay for the solution. This is innovation. Startups by definition must innovate, else they fail to thrive. Innovation is the one thing that allows them to compete with established companies.

At the outset, all tasks are handled by the individuals in this small group, but at some point the startup becomes successful enough to warrant additional employees to handle the non-innovative tasks considered mundane by the innovators but that are nonetheless essential: answering the phones, manufacturing/packaging product, taking orders, handling customer service, providing quality assurance, maintaining and/or refining the existing product line, etc. The company's focus, as measured by the number of man-hours spent doing the various tasks, shifts from innovating to staying profitable. As soon as there are more people in the company spending more time doing anything other than innovating, the company has changed, and the shift is palpable to anyone there since the beginning.

Meanwhile the original innovators often do one of two things: they remain focused on innovation either by hiding themselves in their offices and developing the next-generation product, or by leaving the company to start up another.

This is merely Startup 101, the life-cycle of nearly all high-tech startup companies; anyone who has worked at a successful startup recognizes the story. Historically, the life-cycle of the typical innovation-based company includes a slow-down in innovation and a resultant brain drain, but it is exactly this historical trend that one must overcome to remain innovative as an organization.

How is an executive to reverse this trend? Primarily by maintaining the innovative atmosphere associated with the origins of the company. However simple this may sound, there is no obvious mechanism; companies the world over are scrambling for a successful recipe.

Many companies hire outside people; they bring in hired guns to do the really innovative work for a new design. This can prove successful, but it can also backfire if the existing staff feels passed over. In addition, it is just as sensitive to whimsy as relying upon your own staff for innovation if you have not instituted a culture and/or environment conducive to innovation: the contractor's success or failure is effectively out of your control.

It would be more prudent to maximize the probability of success by creating the right environment, whether contracted help is used or not. One approach would be to emulate the risk/reward structure of a startup environment directly, in a manner similar to "skunkworks" projects, combining the competition structure of skunkworks with the reward structure of startups. Such a reward structure, if implemented correctly, would appeal to the top engineers and is likely to attract top engineers from other firms as well.

The fundamental problem is identification: who is good? In a large engineering organization, the answer is non-trivial. To retain good engineers, a company must reward substance, not decoration. Nothing destroys R&D morale faster than good engineers knowing they are paid less than other engineers who don't contribute to the bottom line. However, identifying substance is difficult, especially for non-technical managers and executives. Among other things, as mentioned earlier, engineers tend to be good at only one of the following two skills: their job and promoting themselves. Really good engineers are often quiet and undervalued by their company, so the question is how to identify those individuals. One answer is to let them do it themselves; the denizens of R&D know perfectly well who is good and who is not—let them identify the cream of the crop through their actions. This is a page taken directly from the startup story, where innovation comes from self-selected groups of extremely competent engineers.

A Structure for Identification and Reward: Venture Capitalism

Envision a skunkworks-type competition within R&D for the next-generation design wherein the design teams are self-selected, and the winning team is given a (small but significant) direct share of the product's revenue. This emulates the startup environment very closely—by definition a team must innovate to succeed, and, all else being equal, the team that does the best job is rewarded in proportion to their efforts and skills. The reward is tied directly to revenue generated by the team, which mirrors the reward structure for salesmen and executives, who get paid to produce results. This form of incentive (tying one's reward to the revenue generated) can be extremely effective at motivating people's best efforts, so it surprising that, outside of the startup arena, it is rarely used to motivate innovation in engineering.

A few of the details in this arrangement are understated but quite important:

- The teams must be small. This is for two reasons; first, the productivity of a team is roughly inversely proportional to the team's size (Brooks, 1995). Second, the share of revenue will be split among the entire team, so the larger the team, the smaller the

individual reward. In the limiting case, one could reward the entire R&D department for a design success, but that would do little to motivate the most talented.

- The teams must be self-selected. This is the solution to the identification problem posed earlier: non-technical managers are nowhere near as adept at identifying design talent as are the engineers themselves. Faced with a non-trivial challenge, engineers will want none other than the best on their teams; all else would be dead wood slowing the team down.
- The reward must be real. For quite a while, top executives and salesmen have been paid staggering and highly publicized bonuses for their results (and, in many cases, even their failures). It is hard to get past the surreal juxtaposition of handing a $25 million bonus to an executive for his efforts and handing a congratulatory plaque to an engineering team for their efforts, if the engineering efforts affect the company's bottom line just as significantly as the executive's (for example, by developing a new product and thus a new source of revenue). This incredible disparity of reward is one of the primary reasons the exceptionally talented take matters into their own hands by leaving the corporate environment to start something up on their own.
- The reward must be tied to the product, not the company in general. The typical reward in high tech takes the form of stock options. While this is an appropriate motivator for new hires (like welcoming someone into your family), as a reward for technical innovation on a particular product, it becomes watered down with the size and scope of the company. Give an engineer a piece of the company, and the engineer will work hard, in a vague sense, to ensure the long-term success of the company. Promise an engineer a piece of his product's revenue, and the engineer will work hard on the design and development of that product to ensure its financial success. The carrot dangled dictates the resulting behavior.

This identification/reward structure emulates the competitive engineering environment of the startup industry and, given an appropriate level of reward, would likely attract and retain the same uber-talented engineers as the high-tech startup industry. Interestingly, the self-selective creative aspect mirrors the environment at Disney under Bob Iger, where a six-fold increase in revenue resulted from supporting director-driven movie creation over management-driven movie creation (Economist, 19 April 2008). From an engineer's perspective, this type of identification/reward structure is ideal: it rewards talent and demands of a design team creativity married with

competence—in all likelihood, for a design to succeed in this environment, the idea must be innovative, and it must work. There is no better glove to throw down on an engineer's desk.

If it strikes the reader as if the corporate entity essentially becomes a high-tech VC firm funding its own R&D staff, that is exactly what is proposed. Similar precedents exist in high-tech today, for instance Microsoft, Google, and Facebook spurring third-party innovation directly via millions in developer seed funding (Mills, 2008; Google, 2008; Farber, 2007). The only difference is that these examples show companies funding *external* innovation rather than *internal* innovation, which benefits the company indirectly rather than directly. Exploring an internally directed scenario would be worthwhile: the costs would be lower (payment is not speculative but upon success), the benefits more readily observed and quantified.

Moreover, this is arguably one of the *only* feasible solutions to the problem of identifying, retaining, motivating, and rewarding the industry's best designers. As Brooks states, design is an individual process: "although many fine, useful software systems have been designed by committees and built by multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers." (Brooks, 1987) Brooks goes further, and though he speaks of software design, the sentiment is just as applicable to hardware design and embedded systems design:

> I think the most important single effort we can mount is to develop ways to grow great designers. No software organization can ignore this challenge. Good managers, scarce though they be, are no scarcer than good designers. Great designers and great managers are both very rare. Most organizations spend considerable effort in finding and cultivating the management prospects; I know of none that spends equal effort in finding and developing the great designers upon whom the technical excellence of the products will ultimately depend.
>
> [Brooks 1987, reprinted in Brooks 1995]

As Graham suggests in the chapter's opening quote above, were a company to adopt such tactics, it would most likely attract engineers who wanted to work especially hard. It would crush its competitors.

THE VALUE OF DESIGN

At this point we have addressed the issues of identifying and rewarding the company's most talented and innovative engineers, but the original problem is not yet solved. The last issue is to reward the correct designer or design team—to recognize "good" design. The failure of many

companies to capitalize on their own innovations (numerous historical examples spring to mind, including graphical user interfaces, laser printing, computer networks) suggests that, of the set, this is probably the most difficult problem to solve.

Clearly, every industry, and every different product within that industry, will have its own set of metrics for success—qualities that make one design better than others—so it is impossible to be comprehensive here. In addition, most managers and executives already believe themselves good judges of design. Graham discusses some characteristics common to good designs, and at the very least his list (which should be read slowly) will provoke thought:

> Good design is simple.
> Good design is timeless.
> Good design solves the right problem.
> Good design is suggestive.
> Good design is often slightly funny.
> Good design is hard.
> Good design looks easy.
> Good design uses symmetry.
> Good design resembles nature.
> Good design is redesign.
> Good design can copy.
> Good design is often strange.
> Good design happens in chunks.
> Good design is often daring.
>
> [from the essay "Taste for Makers," which discusses each in detail, Graham 2004]

Among other things, good design is not the same as choosing something safe. A safe technology choice is typically a mediocre design chosen not to inspire but rather to avoid failure. Committees typically choose safe designs. Innovators typically choose good designs. The executive who asks for innovation must be mindful of what he wishes for, on multiple levels:

- A request for innovation is a tacit acceptance of risk.
- Old habits die hard: engineers accustomed to choosing safe designs, and management that has historically rewarded safe design choices, will continue down their well-trodden paths until led elsewhere.

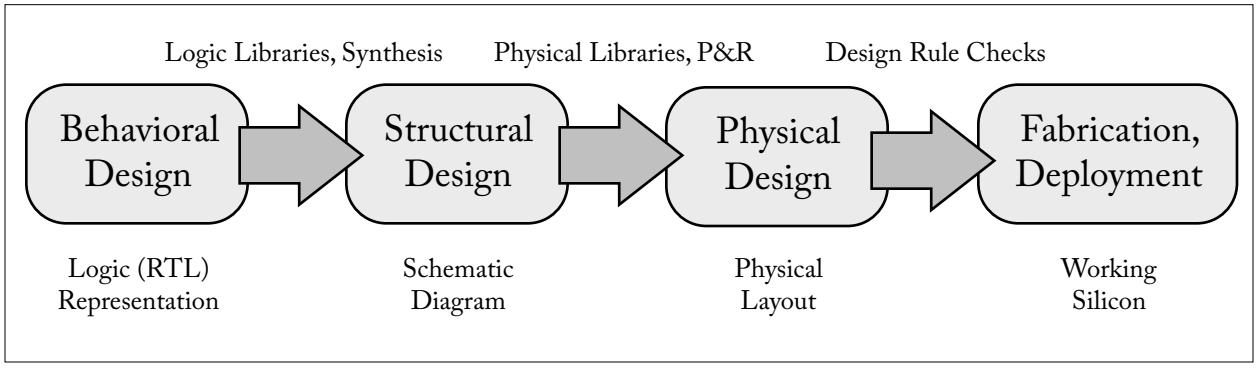

Figure 1: VLSI design flow.

- "Industry best practice" is not. Best, that is. It is by definition the technological state of the art, which in high tech is merely the industry-wide status quo, as any advances are quickly adopted by all; more importantly, it is what the innovator is attempting to beat.

A Tale of Three Design Flows

Both hardware and software are extremely powerful technologies, but both leave much to be desired, creating a significant opportunity for innovation. In particular, each can learn from the other—there is plenty of room for software to be more reliable and for hardware to be more exotic. The engineering challenge is to do this without sacrificing the beneficial characteristics.

This section describes the typical development processes in semiconductor design, embedded systems design, and software design, with the goal of giving the non-technical manager an idea of what is going on. The discussion leans toward what is arguably the easiest problem to solve: that of improving embedded systems' reliability by importing principles of semiconductor design. The example illustrates that not all innovation produces new and better things; often it is just as valuable to produce new and better techniques.

VLSI Design. To begin with, consider VLSI design, the creation of semiconductor parts. Jan du Preez, at the time the President of Infineon Technologies North America, stated quite flatly that "semiconductor design is possibly *the* most complex thing that humans do." (du Preez, 2002) It is also relatively expensive: a mask set for a cutting-edge process technology typically runs in the millions of dollars. Any design revision requires new masks, potentially a full set, so this is not a technology conducive to an iterative design-build-test-redesign development cycle. Designers do not build a chip to test their designs; they build the chip only once they are certain it will work. A design must work the first time around or, worst case, the second time around—more than that, and the project is scrapped and/or the company goes out of business.

```
module fibonacci(clk, rst_l, out_w);

   input          clk, rst_l;
   output   [7:0] out_w;

   reg      [7:0] src1, out;
   wire     [7:0] out_w = out;

   always @(posedge clk)
   begin
      if(!rst_l)
      begin
         src1 <= 1'd0;
         out <= 1'd1;
      end
   else
      begin
         src1 <= out_w;
         out <= src1 + out_w;
      end
   end

endmodule
```

Figure 2: Example behavioral design.

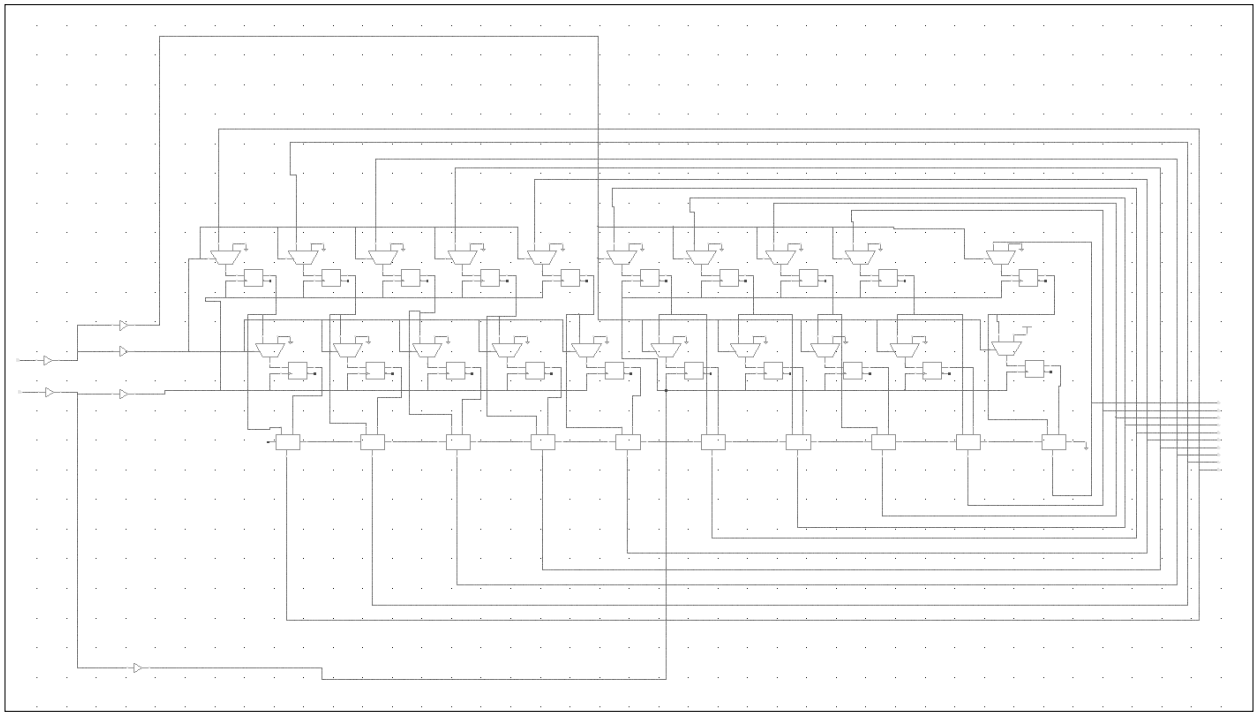

Figure 3: Example structural design.

The methodology for VLSI design enables such tight tolerances on correctness. The design flow is characterized by strict design rules; the development tools enable a verifiable physical design, meaning that one can verify at the design stage, using CAD tools, whether or not the physical implementation will work. One need not build a chip to verify the chip's design.

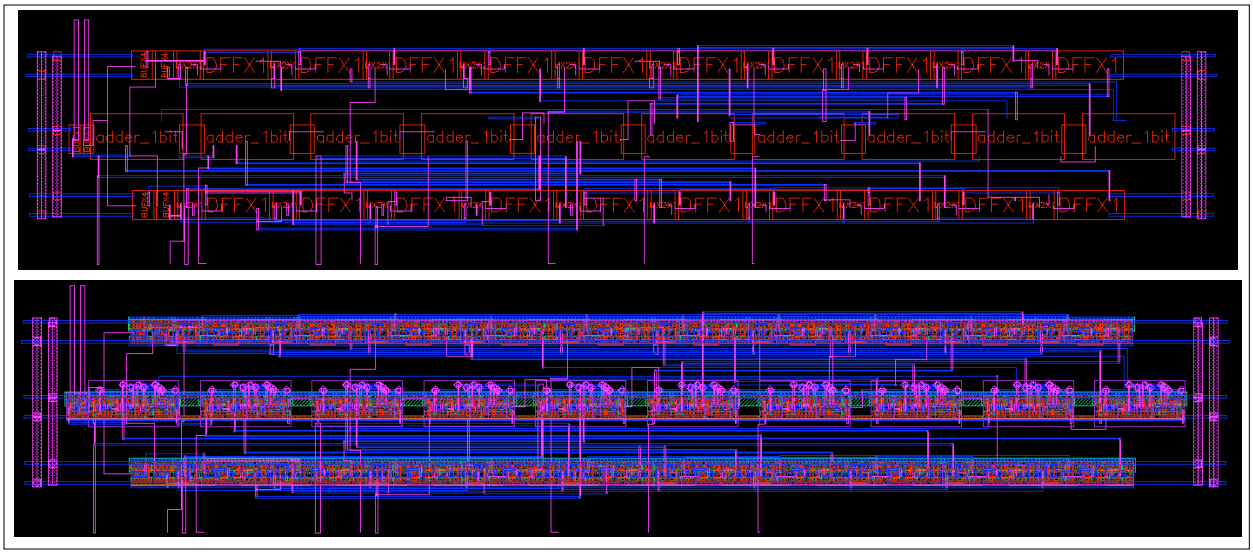

Figure 4: Example physical layouts, one identifying the larger circuit structures such as adders and D flip-flops (top), the other showing detail down to the transistor level (bottom).

A typical design flow is illustrated in Figure 1. The engineer begins with a very high-level representation of the final chip: a behavioral design which looks very much like a piece of software. This specification indicates what the chip is supposed to do, how it is supposed to behave, as opposed to what circuits to use. It is analogous to a blueprint for a building, which typically ignores such implementation details as size of nails, chemical composition of brick & mortar, species of wood, etc. The specification is a logic representation, often called an "RTL" specification, for "register-transfer level," indicating that it specifies what data and commands are transferred and processed, at what time, and stored in what registers.

Figure 2 illustrates an example behavioral design: a simple sequential state machine that produces the first few numbers in the Fibonacci sequence. The code is Verilog, a hardware description language (HDL) with a C-like syntax. The specifics of the design matter less than the fact that what will ultimately be realized in hardware begins in software, specified at a level that is human-readable and human-debuggable (a software engineer who has never built hardware could figure out how it works). As Brooks asserts (1987), the hardest part of designing a product is "arriving at a complete and consistent specification, and much of the essence of building [the product] is in fact the debugging of the specification." Hardware design thus begins with a behavioral design, a form of specification that is far easier to develop and debug than is hardware itself.

The behavioral design is run through a CAD tool that synthesizes a structural design by replacing various functions with their circuit-level equivalents taken from various logic libraries

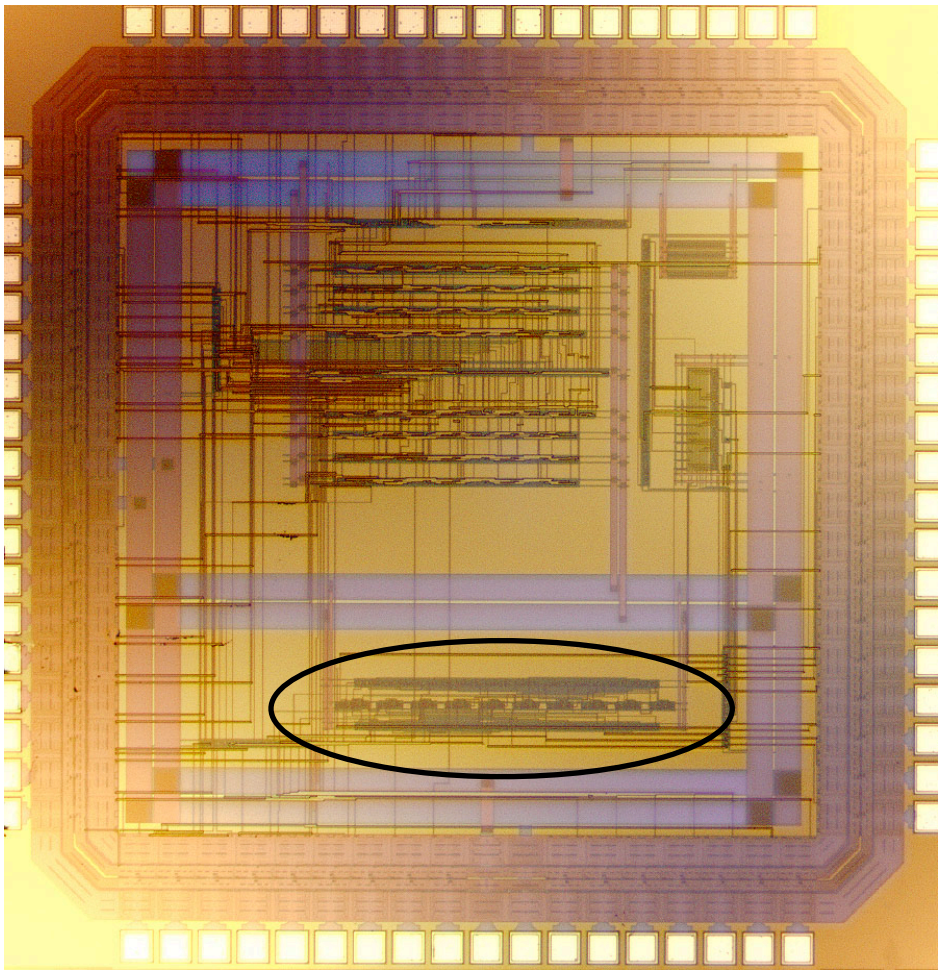

Figure 5: Fabricated VLSI die with the example circuit from Figures 1–4 indicated.

(e.g., replacing each instance of the '+' operator with an n-bit wide adder, each instance of the '<<' operator with an n-bit shifter, etc.). The format of the structural design is a schematic diagram, e.g., the schematic in Figure 3, which illustrates the netlist (circuit) produced from the behavioral code in Figure 2. The synthesis tools can be parameterized to choose between circuit implementations that are characterized as fast and those that are characterized as small, but that is generally the limit to their abilities. Their main benefit is the saving of valuable engineering time: the tools provide a reasonable first cut at a low level design, one which the engineer will further optimize by hand.

The optimized structural design is run through another CAD tool that replaces the logic-level structures with equivalent physical layouts taken from a library, places those structures on the chip, and routes the signals that connect them. What is produced is a design that looks on the computer screen exactly like the chip that will be fabricated: a vast Mondrian on black. Figure 4

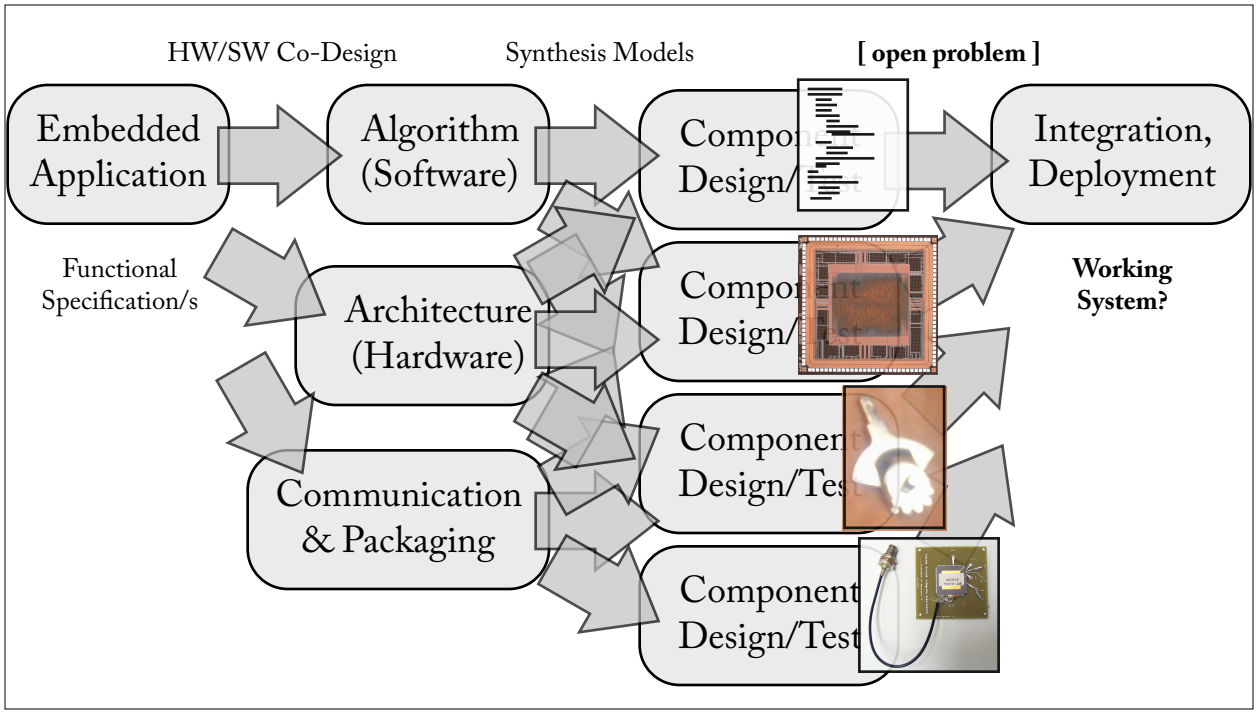

Figure 6: Embedded-systems design flow.

illustrates the layout produced from the netlist shown in Figure 3. The figure presents two layouts, one showing more detail than the other. The bottom layout, showing the transistor-level detail, is exactly what the design will look like when fabricated in silicon.

Before the design can be fabricated, it must pass a number of tests (e.g., electrical checks, design rule checks, etc.) that ensure the part as built will faithfully reproduce the design. These tests ensure that each level of design (behavioral, structural, physical) corresponds exactly to the other, that whatever physical connections are implied at one level of design are implemented in another, and that all electrical connections existing at the physical level are anticipated by the higher-level designs. The design rules, if followed, further guarantee no unanticipated interactions between components. For example, wires spaced too closely might short out, creating a physical connection where none was intended; transistors placed too near one another can influence each other similarly. The end result of all the testing is a reasonable guarantee that, when the part is fabricated, any mistakes found in the implementation will be the result of a faulty specification (i.e., behavioral design) and not the fault of fabrication. Each level of the design (behavioral, structural, physical) can be tested thoroughly with CAD tools, and as each successive design approaches more closely the final physical form, so, too, the tests that are applied to the designs mimic more closely the tests one would perform on an actual chip, the more realistic and

convincing the results, and the more confident the engineer that the design will work when fabricated.

Figure 5 shows a photo-micrograph of the fabricated die, with the example layout circled. The fabricated part was fully functional.

Note that the reason digital VLSI design is verifiable is because of its limited palette: a digital designer can use wires and transistors, and that is all. This was considered extremely limiting when the concept was introduced in the late 1970s by Carver Mead and Lynn Conway (Mead & Conway, 1979), as designers had been comfortable designing digital functions using all manners of devices, processing steps, and process technologies that were incompatible with each other and all essentially analog in terms of their analysis. The introduction of a limited palette coupled with strict design rules enabled the analysis and verification of designs to be relegated to CAD tools, thereby enabling significantly more sophisticated designs. Previously, designers could put only a handful of devices on one chip—not due to space limitations but instead due to reliability. It was only after the introduction of VLSI design rules that the exponential growth in the semiconductor industry began.

Embedded Systems Design. To compare with the VLSI design flow, the design flow for embedded systems is shown in Figure 6. An embedded application is specified at a functional level (what the application does, as opposed to how it does it), using methods that range from the informal to the formal: e.g., prose descriptions, block diagrams, pseudocode, state machines, mathematical expressions, MatLab code, UML diagrams.

The first step is to partition the high-level application into the various blocks that will implement it, for instance hardware and software. An entire field of study is devoted to this partitioning problem, called hardware/software co-design, in which the partitioning is automated. An additional component that is often overlooked is the physical packaging of the components and mechanisms that allow the components to interact (e.g., circuit boards, multi-chip modules, 3D stacking, computer enclosures, switches, wired and wireless networks and their protocols, etc.). Often the choice of packaging and communication mechanisms dictates whether desired functions can be implemented.

After choosing the components to constitute the application, engineers build and test each non-COTS component. For instance, software modules must be written, ASICs developed and fabricated (using the VLSI design flow described earlier), actuators and sensors developed, metal/wood/plastic parts machined and assembled, circuit boards manufactured and assembled.

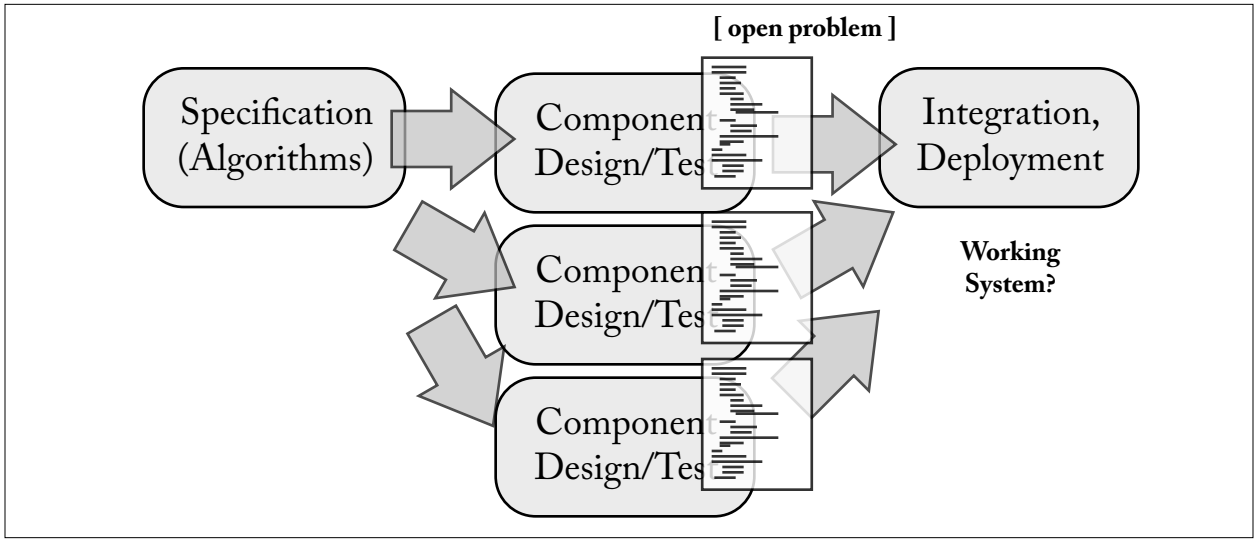

Figure 7: Software design flow.

Synthesis models for some of these components exist—meaning that, given a high-level specification, a CAD tool can produce a low-level design for the desired part (as in the VLSI steps before)—but these synthesis models are not nearly as well developed as those for semiconductor design, so much of this level of development is done by hand.

An important fact is that these individual components are designed, built, and tested in isolation from each other. This is fairly representative—e.g., in the design of automobile control networks it is not unusual for each component (powertrain, spark-plug ignition, anti-lock brake system, stability augmentation system) to be designed and built by a separate group, division, or even company whose sole focus is that control system.

When the components are ready, they are integrated into a complete system, the first testable prototype. In many cases, this is the first time a full system test can be performed, so it is no surprise that it is an open question whether or not the system will indeed work when turned on. Whereas the VLSI design flow is characterized by strict design rules that guarantee a degree of system-level verification, the embedded-systems design flow is characterized by a complete lack of design rules and the use of ad hoc methods (at best) for system-level verification.

This is an obvious candidate for technology innovation. The integration of complex heterogeneous components is a stumbling block for system design: one cannot guarantee the correctness of a system by verifying components in isolation—the entire system must be verified as a whole. Yet component-level design and verification is typical practice in building modern-day embedded systems; system-level testing is frequently done only when the already-built

components are assembled into a working system. The practice leaves latent design bugs that are too subtle to be uncovered by physical tests and that manifest themselves after deployment. For example, the software loops in an automobile control system are usually designed and tested in isolation, but a recent trend in the industry is to time-share these loops on a single microprocessor. In this scenario, the implicit assumption of independence is no longer valid, and, now that there can be direct interaction between the loops, unintended (and untested) behaviors can arise. Another example: a recently proposed redesign of the Blackhawk helicopter replaced the existing electrical wiring of the controller-area network with a fibre-optic channel. Though the new optical network increased bandwidth tremendously, the increase in packet latency that was required by the optical network-interface hardware made it impossible to design a stability and control augmentation system that would meet the desired specifications.

The good news is an increased trend of building embedded systems in a manner similar to semiconductor devices, i.e. entirely in software—for example Fiat's rapid development of the 500 (Economist, 26 April 2008). Such a CAD-oriented method is far from standard in most industries, and so any company that can adopt this technique stands to improve its time to market and reduce design bugs discovered late in the development cycle.

Software Design. The third design flow, that of software development, is shown in Figure 7. The diagram is the software subset of the embedded-systems design flow, and, like embedded systems in general, software design is largely done by hand: the components are designed and developed by hand, and they are tested in isolation before being integrated into the main codebase. This presents problems for components developed by different people or different organizations, as it heightens the importance of correctly and unambiguously specifying the interfaces between components. Witness the loss of the 1999 Mars probe (Oberg, 1999), in which the technical failure was due to a mis-match of measurement units used by different development teams in different countries.

What makes improving software development challenging is that, unlike hardware development, software is not particularly amenable to CAD-tool support (Brooks, 1987). Thus, improving the reliability of hardware systems and embedded systems is the relatively low hanging fruit, and improving the reliability of software is still a long-term goal and a heavily researched area.

Emergent Paradigms: Manufacturing as a Service, Design as an End Product

Recent trends would seem a boon for companies that want to focus more attention on technology innovation; they enable a company to spend less capital on infrastructure and less attention on manufacturing. Rather than exploiting these trends merely to cut costs, an organization could instead spend the freed capital and attention on R&D and quality assurance.

In every era, manufacturing has been pushed to the fringes of society, away from living and communal spaces. Factories have been moved away from dense urban areas; manufacturing has been exported to the third world. At the same time, design has never been pushed away; design has never been exported, except to the detriment of the exporter. Design is the core intellectual exercise that can define a company, an industry, a culture, a nation. Assuming you consider yourself an innovator, if you give design over to a third party, you have given away your reason for being—anything else you bring to the table can be bought; all else but design is a commodity.

The interesting result is manufacturing as a service, a phenomenon increasing in both visibility and popularity. Traditionally, capital expenses are required to enter a manufacturing industry: before you can build widgets, you must first build a factory that can build widgets. One consequence of the Internet and the international competition it has enabled (Friedman, 2005) is the number of plants offering custom manufacturing as a retail service. For instance, circuit-board manufacturers accept customer designs uploaded via the web; they manufacture the boards and perform assembly as well—i.e., given bills of materials they will return a fabbed and completely populated board and can obtain all COTS parts directly from distributors. Some of the world's largest semiconductor companies, such as TSMC, Taiwan Semiconductor Manufacturing Company, focus on building other people's designs, not their own. Similar services are available for CNC routing, plastics, metal working, final assembly, etc.

Due to this phenomenon, the traditional capital-expense barrier to entry no longer exists: one can "manufacture" a new product line that is manufactured entirely by third parties. Certainly the per-unit costs are higher, but the capital startup cost is gone; that means the cost exposure of testing the waters with a new product, even in a new industry, is effectively nil (it is only an issue for experiments that fail so miserably and so publicly that negative PR hurts the company's other products).

This now enables a focus on design, as an end product in and of itself: anything your organization can design, someone out there can and will manufacture for you, and the net has not only enabled this but also simplified tremendously the search for willing manufacturers.

The obvious conclusion is that this empowers startup companies to compete in industries that were previously out of reach. Numerous examples can be found in the semiconductor industry now that design firms need not spend the several billion dollars required to build a fab: Silicon Valley is bursting with companies offering intellectual property, rather than hard goods, as their value added.

The less obvious conclusion is that it also enables larger companies to try the same trick: to explore new products or even new industries outside the company's core area that would have been deemed too risky had the startup costs included capital expenses. With deeper pockets, a large company should be able to cast a wider net than a startup—in effect, to spread the risk and increase the likelihood of success by emulating the effect of several startups in several different areas or by trying several different approaches in the same area.

Again, if this sounds like the approach that venture capitalists take, it is exactly that, and the logic behind the behavior is exactly the same. Many innovators, after starting up one or more successful high-tech companies, move into the VC industry; their nest egg, capital earned from the sale of their startups, is more likely to grow if it is put into more than one basket.

The lesson to take away is simple but powerful. Innovation is like throwing darts, in that each shot may or may not hit the mark, even for an accomplished thrower. The most reliable way to make a bullseye is to take multiple attempts. The most reliable way to make money in the startup industry, given a fixed amount of attention, time, and energy, is not to start up a company but to fund startup companies. Similarly, an executive that seeks innovation will maximize his chances of success by becoming a de facto VC—by treating his engineering staff as a collection of would-be startup companies.

REFERENCES

Ron Avitzur. "The *Graphing Calculator* Story." 2004.
http://www.pacifict.com/Story

Frederick Brooks. *The Mythical Man-Month*. Addison-Wesley. 1995.

Frederick Brooks. "No silver bullet—essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19. April 1987.

Peter Burrows. "The seed of Apple's innovation," *BusinessWeek*, Voices of the Innovators. October 12, 2004.
http://www.businessweek.com/bwdaily/dnflash/oct2004/nf20041012_4018_db083.htm

Jan du Preez. Personal communication. 2002.

Economist. "Lessons from Apple: What other companies can learn from California's master of innovation," *The Economist*. (on-line from the print edition) June 7, 2007.
http://www.economist.com/opinion/displaystory.cfm?story_id=9302662

Economist. "Disney: Magic restored," *The Economist*. April 19, 2008.

Economist. "Fiat: Rebirth of a carmaker," *The Economist*. April 26, 2008.

Dan Farber."Facebook investors start $10 million fund for developers," *Between the Lines (ZDNet Blogs)*. September 17th, 2007.
http://blogs.zdnet.com/BTL/?p=6286

Thomas Friedman. *The World Is Flat: A Brief History of the Twenty-First Century*. Farrar, Straus and Giroux. 2005.

Google. "Android developer challenge." 2008.
http://code.google.com/android/adc.html

Paul Graham. *Hackers & Painters: Big Ideas from the Computer Age*. O'Reilly Media: Sebastopol CA. 2004.

Lev Grossman. "How Apple does it," *Time.com*. October 16, 2005.
http://www.time.com/time/magazine/article/0,9171,1118384,00.html

Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley. 1979.

Elinor Mills. "Microsoft eyes health care app developers with $3 million fund," *C/Net News.com*. February 24, 2008.
http://www.news.com/8301-10784_3-9877656-7.html

Cait Murphy. "America's most admired companies 2008: 10 Most admired for innovation," *CNNMoney.com*. 2008. http://money.cnn.com/galleries/2008/fortune/0803/gallery.innovation.fortune/index.html

James Oberg. "Why the Mars probe went off course," *IEEE Spectrum*, vol. 36, no. 12, pp. 34–39. December 1999.