EMBEDDED SYSTEM DESIGN

A Unified Hardware/Software Introduction

Frank Vahid / Tony Givargis

Embedded System Design: A Unified Hardware/Software Approach

Frank Vahid and Tony Givargis

Department of Computer Science and Engineering
University of California
Riverside, CA 92521
vahid@cs.ucr.edu
http://www.cs.ucr.edu/~vahid

Preface

This book introduces embedded system design using a modern approach. Modern design requires a designer to have a unified view of software and hardware, seeing them not as completely different domains, but rather as two implementation options along a continuum of options varying in their design metrics (cost, performance, power, flexibility, etc.).

Three important trends have made such a unified view possible. First, integrated circuit (IC) capacities have increased to the point that both software processors and custom hardware processors now commonly coexist on a single IC. Second, quality-compiler availability and average program sizes have increased to the point that C compilers (and even C++ or in some cases Java) have become commonplace in embedded systems. Third, synthesis technology has advanced to the point that synthesis tools have become commonplace in the design of digital hardware. Such tools achieve nearly the same for hardware design as compilers achieve in software design: they allow the designer to describe desired processing in a high-level programming language, and they then automatically generate an efficient (in this case custom-hardware) processor implementation. The first trend makes the past separation of software and hardware design nearly impossible. Fortunately, the second and third trends enable their unified design, by turning embedded system design, at its highest level, into the problem of selecting (for software), designing (for hardware), and integrating processors.

ESD focuses on design principles, breaking from the traditional book that focuses on the details a particular microprocessor and its assembly-language programming. While stressing a processor-independent high-level language approach to programming of embedded systems, it still covers enough assembly language programming to enable programming of device drivers. Such processor-independence is possible because of compiler availability, as well as the fact that integrated development environments (IDE's) now commonly support a variety of processor targets, making such independence even more attractive to instructors as well as designers. However, these developments don't entirely eliminate the need for some processor-specific knowledge. Thus, a course with a hands-on lab may supplement this book with a processor-specific databook and/or a compiler manual (both are typically very low cost or even free), or one of many commonly available "extended databook" processor- specific textbooks.

ESD describes not only the programming of microprocessors, but also the design of custom-hardware processors (i.e., digital design). Coverage of this topic is made possible by the above-mentioned elimination of a detailed processor architecture study. While other books often have a review of digital design techniques, ESD uses the new top-down approach to custom-hardware design, describing simple steps for converting high-level program code into digital hardware. These steps build on the trend of digital design books of introducing synthesis into an undergraduate curriculum (e.g., books by Roth, Gajski, and Katz). This book assists designers to become users of synthesis. Using a draft of ESD, we have at UCR successfully taught both programming of embedded microprocessors, design of custom-hardware processors, and integration of the two, in a one-quarter course having a lab, though a semester or even two quarters would be

preferred. However, instructors who do not wish to focus on synthesis will find that the top-down approach covered still provides the important unified view of hardware and software.

ESD includes coverage of some additional important topics. First, while the need for knowledge specific to a microprocessor's internals is decreasing, the need for knowledge of interfacing processors is increasing. Therefore, ESD not only includes a chapter on interfacing, but also includes another chapter describing interfacing protocols common in embedded systems, like CAN, I2C, ISA, PCI, and Firewire. Second, while high-level programming languages greatly improve our ability to describe complex behavior, several widely accepted computation models can improve that ability even further. Thus, ESD includes chapters on advanced computation models, including state machines and their extensions (including Statecharts), and concurrent programming models. Third, an extremely common subset of embedded systems is control systems. ESD includes a chapter that introduces control systems in a manner that enables the reader to recognize open and closed-loop control systems, to use simple PID and fuzzy controllers, and to be aware that a rich theory exists that can be drawn upon for design of such systems. Finally, ESD includes a chapter on design methodology, including discussion of hardware/software codesign, a user's introduction to synthesis (from behavioral down to logic levels), and the major trend towards Intellectual Property (IP) based design.

Additional materials: A web page will be established to be used in conjunction with the book. A set of slides will be available for lecture presentations. Also available for use with the book will be a simulatable and synthesizable VHDL "reference design," consisting of a simple version of a MIPS processor, memory, BIOS, DMA controller, UART, parallel port, and an input device (currently a CCD preprocessor), and optionally a cache, two-level bus architecture, a bus bridge, and an 8051 microcontroller. We have already developed a version of this reference design at UCR. This design can be used in labs that have the ability to simulate and/or synthesize VHDL descriptions. There are numerous possible uses depending on the course focus, ranging from simulation to see first-hand how various components work in a system (e.g., DMA, interrupt processing, arbitration, etc.), to synthesis of working FPGA system prototypes.

Instructors will likely want to have a prototyping environment consisting of a microprocessor development board and/or in-circuit emulator, and perhaps an FPGA development board. These environments vary tremendously among universities. However, we will make the details of our environments and lab projects available on the web page. Again, these have already been developed.

Chapter 1 *Introduction*

1.1 Embedded systems overview

Computing systems are everywhere. It's probably no surprise that millions of computing systems are built every year destined for desktop computers (Personal Computers, or PC's), workstations, mainframes and servers. What may be surprising is that *billions* of computing systems are built every year for a very different purpose: they are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user. Creating a precise definition of such embedded computing systems, or simply *embedded systems*, is not an easy task. We might try the following definition: An embedded system is nearly any computing system other than a desktop, laptop, or mainframe computer. That definition isn't perfect, but it may be as close as we'll get. We can better understand such systems by examining common examples and common characteristics. Such examination will reveal major challenges facing designers of such systems.

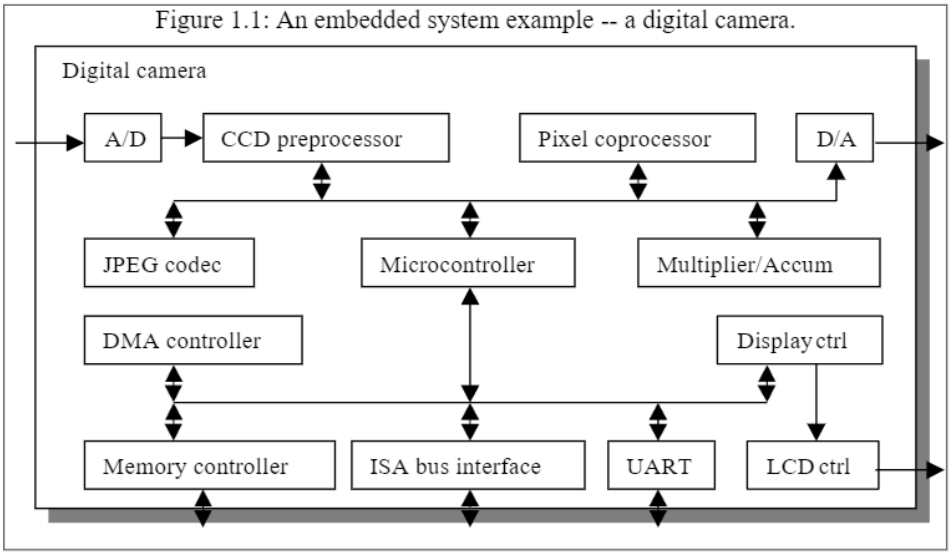
Embedded systems are found in a variety of common electronic devices, such as: (a) consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants; (b) home appliances -- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems; (c) office automation -- fax machines, copiers, printers, and scanners; (d) business equipment -- cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines; (e) automobiles -- transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension. One might say that nearly any device that runs on electricity either already has, or will soon have, a computing system embedded within it. While about 40% of American households had a desktop computer in 1994, each household had an average of more than 30 embedded computers, with that number expected to rise into the hundreds by the year 2000. The electronics in an average car cost $1237 in 1995, and may cost $2125 by 2000. Several billion embedded microprocessor units were sold annually in recent years, compared to a few hundred million desktop microprocessor units.

Embedded systems have several common characteristics:

1) *Single-functioned:* An embedded system usually executes only one program, repeatedly. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.[1]
2) *Tightly constrained:* All computing systems have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

[1] There are some exceptions. One is the case where an embedded system's program is updated with a newer program version. For example, some cell phones can be updated in such a manner. A second is the case where several programs are swapped in and out of a system due to size limitations. For example, some missiles run one program while in cruise mode, then load a second program for locking onto a target.

Figure 1.1: An embedded system example -- a digital camera.



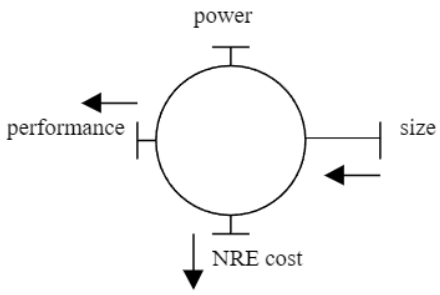
3) *Reactive and real-time:* Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

For example, consider the digital camera system shown in Figure 1.1. The *A2D* and *D2A* circuits convert analog images to digital and digital to analog, respectively. *The CCD preprocessor* is a charge-coupled device preprocessor. The *JPEG codec* compresses and decompresses an image using the JPEG[2] compression standard, enabling compact storage in the limited memory of the camera. The *Pixel coprocessor* aids in rapidly displaying images. The *Memory controller* controls access to a memory chip also found in the camera, while the *DMA controller* enables direct memory access without requiring the use of the microcontroller. The *UART* enables communication with a PC's serial port for uploading video frames, while the *ISA bus interface* enables a faster connection with a PC's ISA bus. The *LCD ctrl* and *Display ctrl* circuits control the display of images on the camera's liquid-crystal display device. A *Multiplier/Accum* circuit assists with certain digital signal processing. At the heart of the system is a *microcontroller*, which is a processor that controls the activities of all the other circuits. We can think of each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks.

This example illustrates some of the embedded system characteristics described above. First, it performs a single function repeatedly. The system always acts as a digital camera, wherein it captures, compresses and stores frames, decompresses and displays frames, and uploads frames. Second, it is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must be fast so that it can process numerous images in milliseconds. It must consume little power so that the camera's battery will last a long

[2] JPEG is short for the Joint Photographic Experts Group. The 'joint' refers to its status as a committee working on both ISO and ITU-T standards. Their best known standard is for still image compression.

Figure 1.2: Design metric competition -- decreasing one may increase others.



time. However, this particular system does not posses a high degree of the characteristic of being reactive and real-time, as it only needs to respond to the pressing of buttons by a user, which even for an avid photographer is still quite slow with respect to processor speeds.

1.2 Design challenge – optimizing design metrics

The embedded-system designer must of course construct an implementation that fulfills desired functionality, but a difficult challenge is to construct an implementation that simultaneously optimizes numerous design metrics. For our purposes, an implementation consists of a software processor with an accompanying program, a connection of digital gates, or some combination thereof. A design metric is a measurable feature of a system's implementation. Common relevant metrics include:

- *Unit cost:* the monetary cost of manufacturing each copy of the system, excluding NRE cost.
- *NRE cost (Non-Recurring Engineering cost):* The monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost (hence the term "non-recurring").
- *Size:* the physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- *Performance:* the execution time or throughput of the system.
- *Power:* the amount of power consumed by the system, which determines the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.
- *Flexibility:* the ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.
- *Time-to-market:* The amount of time required to design and manufacture the system to the point the system can be sold to customers.
- *Time-to-prototype:* The amount of time to build a working version of the system, which may be bigger or more expensive than the final system implementation, but can be used to verify the system's usefulness and correctness and to refine the system's functionality.
- *Correctness:* our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.
- *Safety:* the probability that the system will not cause harm.
- Many others.

These metrics typically compete with one another: improving one often leads to a degradation in another. For example, if we reduce an implementation's size, its performance may suffer. Some observers have compared this phenomenon to a wheel with numerous pins, as illustrated in Figure Figure 1.2. If you push one pin (say size) in, the others pop out. To best meet this optimization challenge, the designer must be

Figure 1.3: Market window.

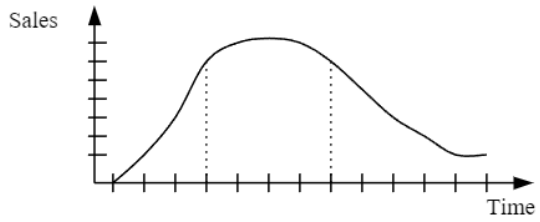
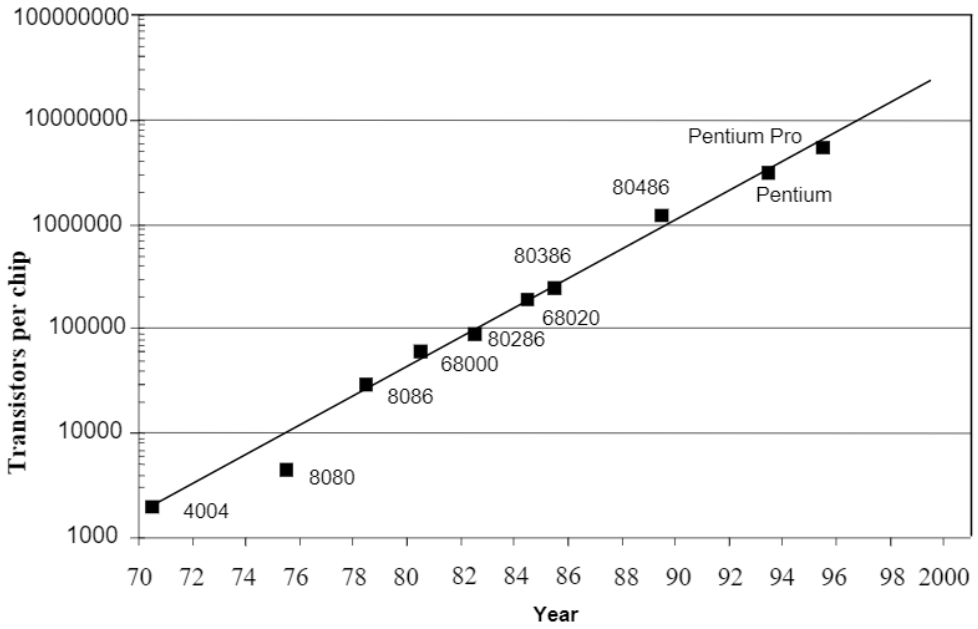


Figure 1.4: IC capacity exponential increase.



comfortable with a variety of hardware and software implementation technologies, and must be able to migrate from one technology to another, in order to find the best implementation for a given application and constraints. Thus, a designer cannot simply be a hardware expert or a software expert, as is commonly the case today; the designer must be an expert in both areas.

Most of these metrics are heavily constrained in an embedded system. The time-to-market constraint has become especially demanding in recent years. Introducing an embedded system to the marketplace early can make a big difference in the system's profitability, since market time-windows for products are becoming quite short, often measured in months. For example, Figure 1.3 shows a sample market window providing during which time the product would have highest sales. Missing this window (meaning the product begins being sold further to the right on the time scale) can mean significant loss in sales. In some cases, each day that a product is delayed from introduction to the market can translate to a one million dollar loss. Adding to the difficulty of meeting the time-to-market constraint is the fact that embedded system complexities are growing due to increasing IC capacities. IC capacity, measured in transistors per chip, has grown

exponentially over the past 25 years[3], as illustrated in Figure 1.4; for reference purposes, we've included the density of several well-known processors in the figure. However, the rate at which designers can produce transistors has not kept up with this increase, resulting in a widening gap, according to the Semiconductor Industry Association. Thus, a designer must be familiar with the state-of-the-art design technologies in both hardware and software design to be able to build today's embedded systems.

We can define *technology* as a manner of accomplishing a task, especially using technical processes, methods, or knowledge. This textbook focuses on providing an overview of three technologies central to embedded system design: processor technologies, IC technologies, and design technologies. We describe all three briefly here, and provide further details in subsequent chapters.

1.3 Embedded processor technology

Processor technology involves the architecture of the computation engine used to implement a system's desired functionality. While the term "processor" is usually associated with programmable software processors, we can think of many other, non-programmable, digital systems as being processors also. Each such processor differs in its specialization towards a particular application (like a digital camera application), thus manifesting different design metrics. We illustrate this concept graphically in Figure 1.5. The application requires a specific embedded functionality, represented as a cross, such as the summing of the items in an array, as shown in Figure 1.5(a). Several types of processors can implement this functionality, each of which we now describe. We often use a collection of such processors to best optimize our system's design metrics, as was the case in our digital camera example.

1.3.1 General-purpose processors -- software

The designer of a *general-purpose* processor builds a device suitable for a variety of applications, to maximize the number of devices sold. One feature of such a processor is a program memory – the designer does not know what program will run on the processor, so cannot build the program into the digital circuit. Another feature is a general datapath – the datapath must be general enough to handle a variety of computations, so typically has a large register file and one or more general-purpose arithmetic-logic units (ALUs). An embedded system designer, however, need not be concerned about the design of a general-purpose processor. An embedded system designer simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality. Many people refer to this portion of an implementation simply as the "software" portion.

Using a general-purpose processor in an embedded system may result in several design-metric benefits. *Design time* and *NRE cost* are low, because the designer must only write a program, but need not do any digital design. *Flexibility* is high, because changing functionality requires only changing the program. *Unit cost* may be relatively low in small quantities, since the processor manufacturer sells large quantities to other customers and hence distributes the NRE cost over many units. *Performance* may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.

However, there are also some design-metric drawbacks. *Unit cost* may be too high for large quantities. *Performance* may be slow for certain applications. *Size* and *power* may be large due to unnecessary processor hardware.

For example, we can use a general-purpose processor to carry out our array-summing functionality from the earlier example. Figure 1.5(b) illustrates that a general-

[3] Gordon Moore, co-founder of Intel, predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18-24 months. His very accurate prediction is known as "Moore's Law." He recently predicted about another decade before such growth slows down.