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.

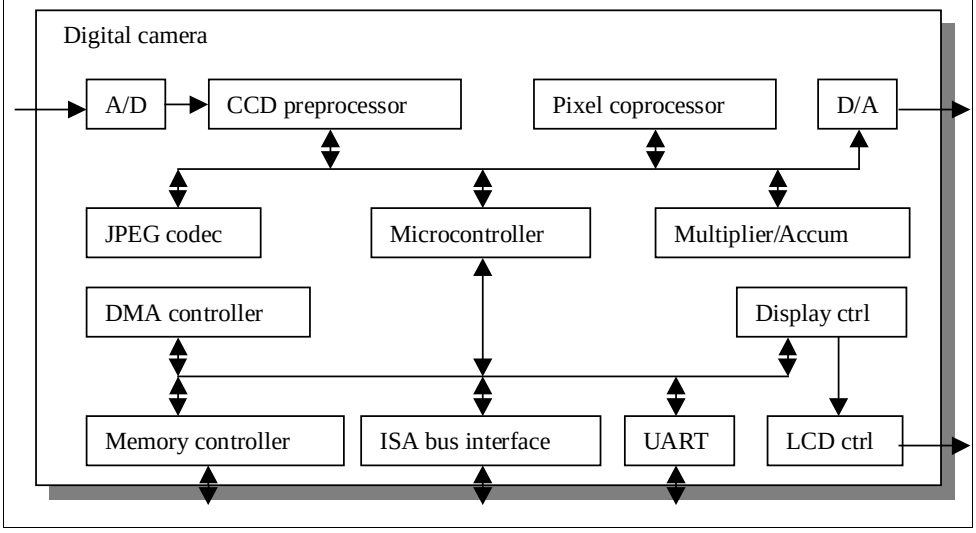
Embedded systems are found in a variety of common electronic devices, such as: (a) consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants; (b) home appliances -- microwave ovens, answering machines, 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.



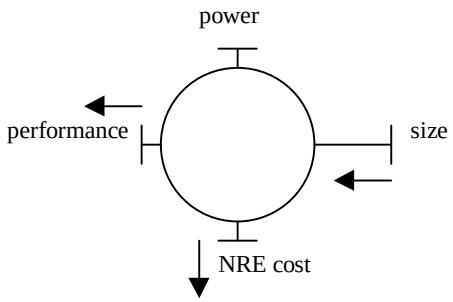
3) *Reactive and real-time:* Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration 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.

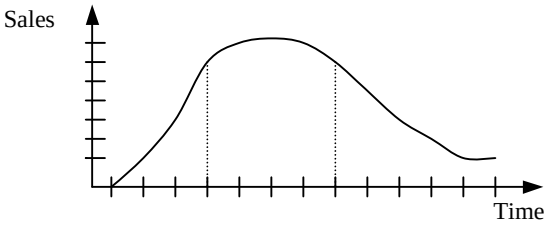
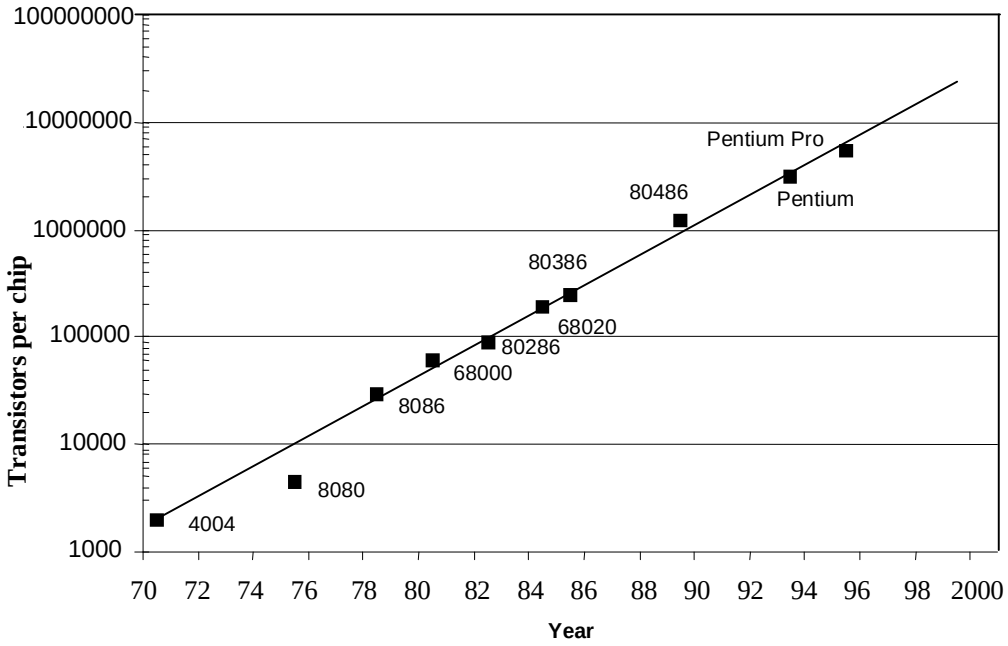


Figure 1.4: IC capacity exponential increase.



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.

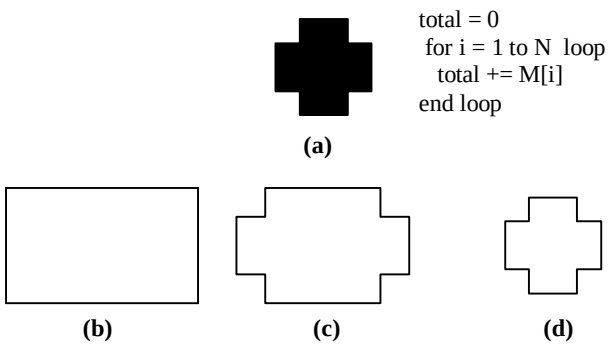
Using a general-purpose processor in an embedded system may result in several design-metric benefits. *Design time* and *NRE cost* are low, because the designer must only write a program, but need not do any digital design. *Flexibility* is high, because changing functionality requires only changing 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.

Figure 1.5: Processors very in their customization for the problem at hand: (a) desired functionality, (b) general-purpose processor, (b) application-specific processor, (c) single-purpose processor.



purpose covers the desired functionality, but not necessarily efficiently. Figure 1.6(a) shows a simple architecture of a general-purpose processor implementing the array-summing functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the datapath for this instruction and executes the instruction. Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

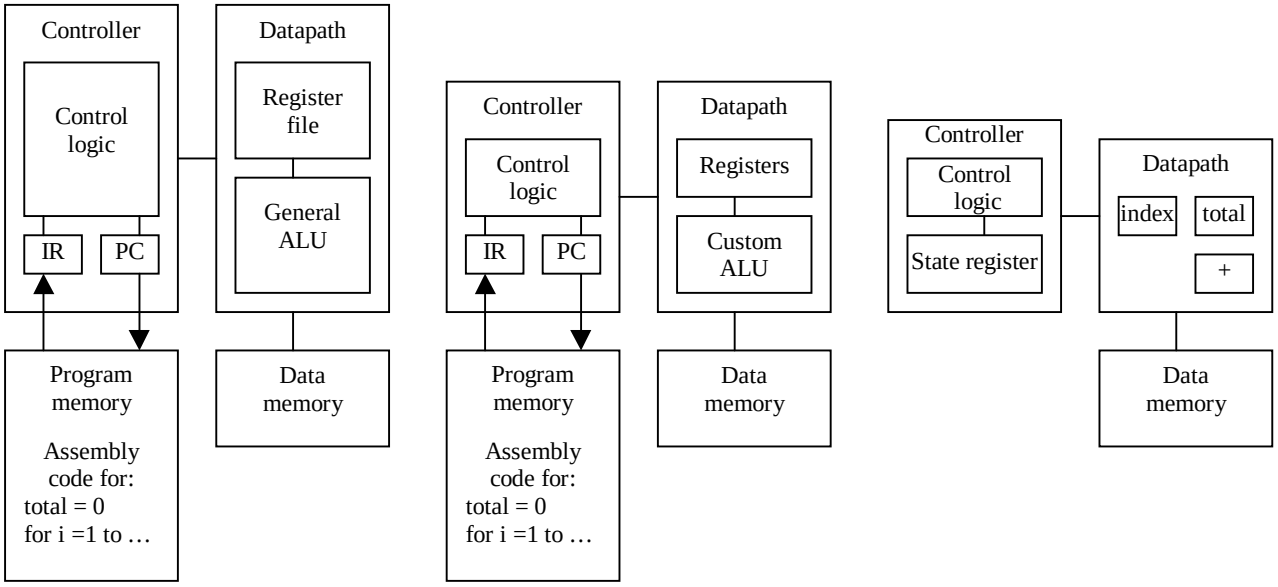
1.3.2 Single-purpose processors -- hardware

A *single-purpose* processor is a digital circuit designed to execute exactly one program. For example, consider the digital camera example of Figure 1.1. All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An embedded system designer creates a single-purpose processor by designing a custom digital circuit, as discussed in later chapters. Many people refer to this portion of the implementation simply as the "hardware" portion (although even software requires a hardware processor on which to run). Other common terms include coprocessor and accelerator.

Using a single-purpose processor in an embedded system results in several design-metric benefits and drawbacks, which are essentially the inverse of those for general-purpose processors. Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications.

For example, Figure 1.5(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality, nothing more, nothing less. Figure 1.6(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an *index* register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labeled *total* to the datapath. Since the example's only arithmetic operation is addition, we add a single adder to the datapath. Since the processor only executes this one program, we hardwire the program directly into the control logic.

Figure 1.6: Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.



1.3.3 Application-specific processors

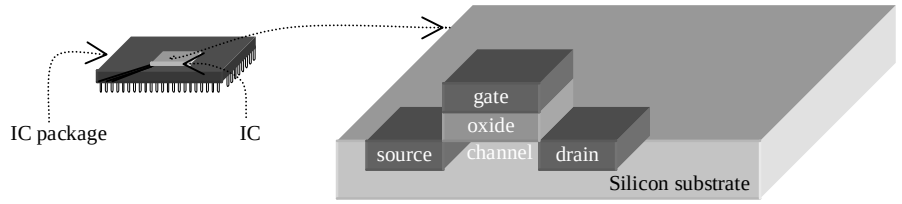
An *application-specific* instruction-set processor (or ASIP) can serve as a compromise between the above processor options. An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc. The designer of such a processor can optimize the datapath for the application class, perhaps adding special functional units for common operations, and eliminating other infrequently used units.

Using an ASIP in an embedded system can provide the benefit of flexibility while still achieving good performance, power and size. However, such processors can require large NRE cost to build the processor itself, and to build a compiler, if these items don't already exist. Much research currently focuses on automatically generating such processors and associated retargetable compilers. Due to the lack of retargetable compilers that can exploit the unique features of a particular ASIP, designers using ASIPs often write much of the software in assembly language.

Digital-signal processors (DSPs) are a common class of ASIP, so demand special mention. A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These operations carry out common signal processing tasks like signal filtering, transformation, or combination. Such operations are usually math-intensive, including operations like multiply and add or shift and add. To support such operations, a DSP may have special-purpose datapath components such a multiply-accumulate unit, which can perform a computation like T = T + M[i]*k using only one instruction. Because DSP programs often manipulate large arrays of data, a DSP may also include special hardware to fetch sequential data memory locations in parallel with other operations, to further speed execution.

Figure 1.5(c) illustrates the use of an ASIP for our example; while partially customized to the desired functionality, there is some inefficiency since the processor also contains features to support reprogramming. Figure 1.6(b) shows the general architecture of an ASIP for the example. The datapath may be customized for the example. It may have an auto-incrementing register, a path that allows the add of a

Figure 1.7: IC's consist of several layers. Shown is a simplified CMOS transistor; an IC may possess millions of these, connected by layers of metal (not shown).



register plus a memory location in one instruction, fewer registers, and a simpler controller. We do not elaborate further on ASIPs in this book (the interested reader will find references at the end of this chapter).

1.4 IC technology

Every processor must eventually be implemented on an IC. IC technology involves the manner in which we map a digital (gate-level) implementation onto an IC. An IC (Integrated Circuit), often called a "chip," is a semiconductor device consisting of a set of connected transistors and other devices. A number of different processes exist to build semiconductors, the most popular of which is CMOS (Complementary Metal Oxide Semiconductor). The IC technologies differ by how customized the IC is for a particular implementation. For lack of a better term, we call these technologies "IC technologies." IC technology is independent from processor technology; any type of processor can be mapped to any type of IC technology, as illustrated in Figure 1.8.

To understand the differences among IC technologies, we must first recognize that semiconductors consist of numerous layers. The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. One way to create these layers is by depositing photo-sensitive chemicals on the chip surface and then shining light through *masks* to change regions of the chemicals. Thus, the task of building the layers is actually one of designing appropriate masks. A set of masks is often called a *layout.* The narrowest line that we can create on a chip is called the *feature size,* which today is well below one micrometer (sub-micron). For each IC technology, all layers must eventually be built to get a working IC; the question is who builds each layer and when.

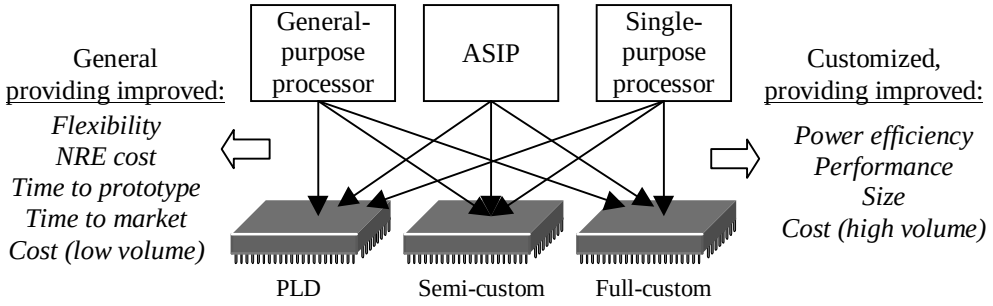
1.4.1 Full-custom/VLSI

In a full-custom IC technology, we optimize all layers for our particular embedded system's digital implementation. Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors. Once we complete all the masks, we send the mask specifications to a fabrication plant that builds the actual ICs. Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent performance with small size and power. It is usually used only in high-volume or extremely performance-critical applications.

1.4.2 Semi-custom ASIC (gate array and standard cell)

In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers. In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates). The remaining task is to connect these gates to achieve our particular implementation. In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by

Figure 1.8: The independence of processor and IC technologies: any processor technology can be mapped to any IC technology.



hand. Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells. ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

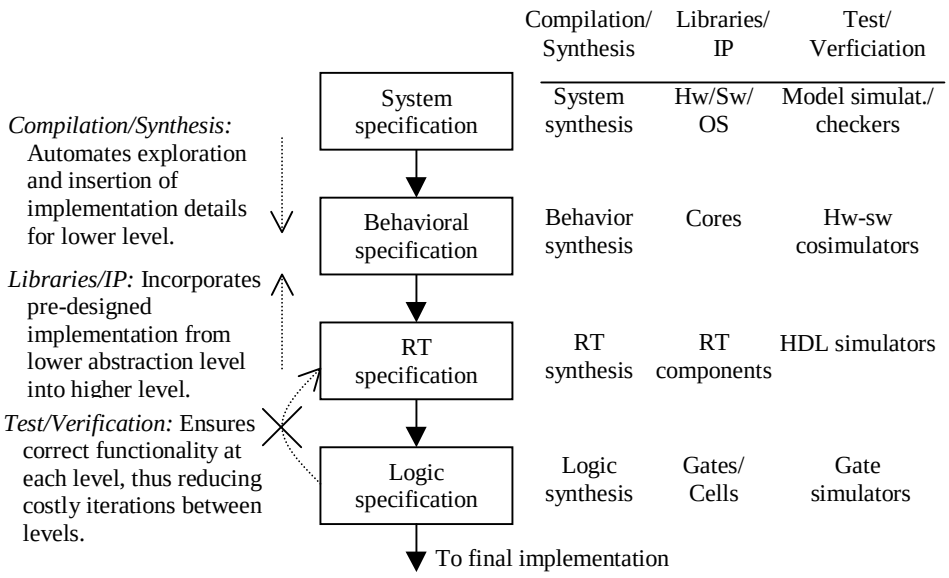
1.4.3 PLD

In a PLD (Programmable Logic Device) technology, all layers already exist, so we can purchase the actual IC. The layers implement a programmable circuit, where programming has a lower-level meaning than a software program. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. Small devices, called programmers, connected to a desktop computer can typically perform such programming. We can divide PLD's into two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates. Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components. One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and are thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability. However, they are typically bigger than ASICs, may have higher unit cost, may consume more power, and may be slower (especially FPGAs). They still provide reasonable performance, though, so are especially well suited to rapid prototyping.

As mentioned earlier and illustrated in Figure 1.8, the choice of an IC technology is independent of processor types. For example, a general-purpose processor can be implemented on a PLD, semi-custom, or full-custom IC. In fact, a company marketing a commercial general-purpose processor might first market a semi-custom implementation to reach the market early, and then later introduce a full-custom implementation. They might also first map the processor to an older but more reliable technology, like 0.2 micron, and then later map it to a newer technology, like 0.08 micron. These two evolutions of mappings to a large extent explain why a processor's clock speed improves on the market over time.

Furthermore, we often implement multiple processors of different types on the same IC. Figure 1.1 was an example of just such a situation – the digital camera included a microcontroller (general-purpose processor) plus numerous single-purpose processors on the same IC.

Figure 1.9: Ideal top-down design process, and productivity improvers.



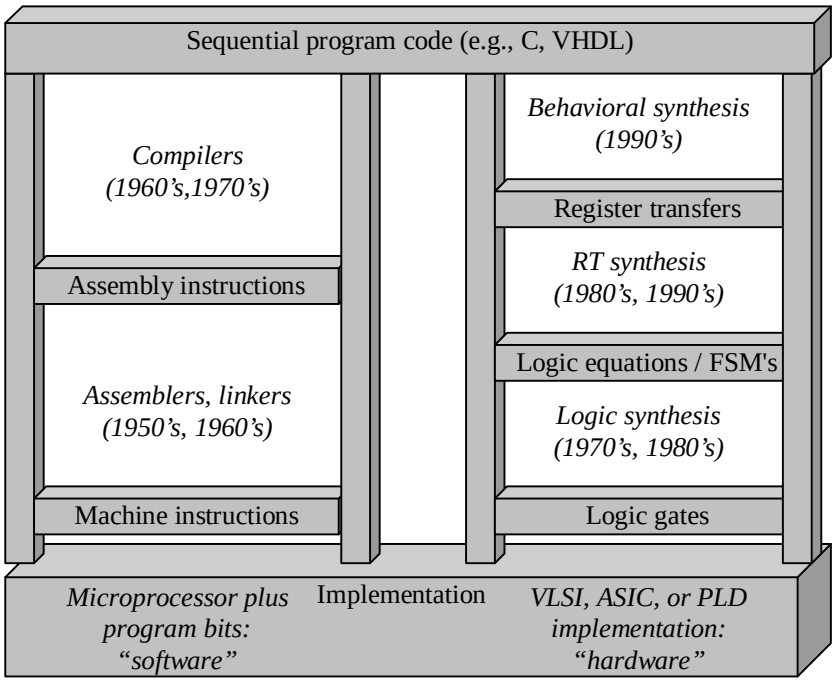
1.5 Design technology

Design technology involves the manner in which we convert our concept of desired system functionality into an implementation. We must not only design the implementation to optimize design metrics, but we must do so quickly. As described earlier, the designer must be able to produce larger numbers of transistors every year, to keep pace with IC technology. Hence, improving design technology to enhance productivity has been a focus of the software and hardware design communities for decades.

To understand how to improve the design process, we must first understand the design process itself. Variations of a *top-down* design process have become popular in the past decade, an ideal form of which is illustrated in Figure 1.9. The designer refines the system through several abstraction levels. At the system level, the designer describes the desired functionality in some language, often a natural language like English, but preferably an executable language like C; we shall call this the *system specification.* The designer refines this specification by distributing portions of it among chosen processors (general or single purpose), yielding *behavioral specifications* for each processor. The designer refines these specifications into *register-transfer (RT) specifications* by converting behavior on general-purpose processors to assembly code, and by converting behavior on single-purpose processors to a connection of register-transfer components and state machines. The designer then refines the register-transfer-level specification of a single-purpose processor into a *logic specification* consisting of Boolean equations. Finally, the designer refines the remaining specifications into an implementation, consisting of machine code for general-purpose processors, and a gate-level netlist for single-purpose processors.

There are three main approaches to improving the design process for increased productivity, which we label as compilation/synthesis, libraries/IP, and test/verification. Several other approaches also exist. We now discuss all of these approaches. Each approach can be applied at any of the four abstraction levels.

Figure 1.10: The co-design ladder: recent maturation of synthesis enables a unified view of hardware and software.



1.5.1 Compilation/Synthesis

Compilation/Synthesis lets a designer specify desired functionality in an abstract manner, and automatically generates lower-level implementation details. Describing a system at high abstraction levels can improve productivity by reducing the amount of details, often by an order of magnitude, that a design must specify.

A logic synthesis tool converts Boolean expressions into a connection of logic gates (called a netlist). A register-transfer (RT) synthesis tool converts finite-state machines and register-transfers into a datapath of RT components and a controller of Boolean equations. A behavioral synthesis tool converts a sequential program into finite-state machines and register transfers. Likewise, a software compiler converts a sequential program to assembly code, which is essentially register-transfer code. Finally, a system synthesis tool converts an abstract system specification into a set of sequential programs on general and single-purpose processors.

The relatively recent maturation of RT and behavioral synthesis tools has enabled a unified view of the design process for single-purpose and general-purpose processors. Design for the former is commonly known as "hardware design," and design for the latter as "software design." In the past, the design processes were radically different – software designers wrote sequential programs, while hardware designers connected components. But today, synthesis tools have converted the hardware design process essentially into one of writing sequential programs (albeit with some knowledge of how the hardware will be synthesized). We can think of abstraction levels as being the rungs of a ladder, and compilation and synthesis as enabling us to step up the ladder and hence enabling designers to focus their design efforts at higher levels of abstraction, as illustrated in Figure 1.10. Thus, the starting point for either hardware or software is sequential programs, enhancing the view that system functionality can be implemented in hardware, software, or some combination thereof. The choice of hardware versus software for a particular function is simply a tradeoff among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference

between what the two can implement. *Hardware/software codesign* is the field that emphasizes this unified view, and develops synthesis tools and simulators that enable the co-development of systems using both hardware and software.

1.5.2 Libraries/IP

Libraries involve re-use of pre-existing implementations. Using libraries of existing implementations can improve productivity if the time it takes to find, acquire, integrate and test a library item is less than that of designing the item oneself.

A logic-level library may consist of layouts for gates and cells. An RT-level library may consist of layouts for RT components, like registers, multiplexors, decoders, and functional units. A behavioral-level library may consist of commonly used components, such as compression components, bus interfaces, display controllers, and even general-purpose processors. The advent of system-level integration has caused a great change in this level of library. Rather than these components being IC's, they now must also be available in a form, called *cores*, that we can implement on just one portion of an IC. This change from behavioral-level libraries of IC's to libraries of cores has prompted use of the term *Intellectual Property (IP)*, to emphasize the fact that cores exist in a "soft" form that must be protected from copying. Finally, a system-level library might consist of complete systems solving particular problems, such as an interconnection of processors with accompanying operating systems and programs to implement an interface to the Internet over an Ethernet network.

1.5.3 Test/Verification

Test/Verification involves ensuring that functionality is correct. Such assurance can prevent time-consuming debugging at low abstraction levels and iterating back to high abstraction levels.

Simulation is the most common method of testing for correct functionality, although more formal verification techniques are growing in popularity. At the logic level, gate-level simulators provide output signal timing waveforms given input signal waveforms. Likewise, general-purpose processor simulators execute machine code. At the RT-level, hardware description language (HDL) simulators execute RT-level descriptions and provide output waveforms given input waveforms. At the behavioral level, HDL simulators simulate sequential programs, and co-simulators connect HDL and general-purpose processor simulators to enable hardware/software co-verification. At the system level, a model simulator simulates the initial system specification using an abstract computation model, independent of any processor technology, to verify correctness and completeness of the specification. Model checkers can also verify certain properties of the specification, such as ensuring that certain simultaneous conditions never occur, or that the system does not deadlock.

1.5.4 Other productivity improvers

There are numerous additional approaches to improving designer productivity. *Standards* focus on developing well-defined methods for specification, synthesis and libraries. Such standards can reduce the problems that arise when a designer uses multiple tools, or retrieves or provides design information from or to other designers. Common standards include language standards, synthesis standards and library standards.

Languages focus on capturing desired functionality with minimum designer effort. For example, the sequential programming language of C is giving way to the object-oriented language of C++, which in turn has given some ground to Java. As another example, state-machine languages permit direct capture of functionality as a set of states and transitions, which can then be translated to other languages like C.

Frameworks provide a software environment for the application of numerous tools throughout the design process and management of versions of implementations. For example, a framework might generate the UNIX directories needed for various simulators

and synthesis tools, supporting application of those tools through menu selections in a single graphical user interface.

1.6 Summary and book outline

Embedded systems are large in numbers, and those numbers are growing every year as more electronic devices gain a computational element. Embedded systems possess several common characteristics that differentiate them from desktop systems, and that pose several challenges to designers of such systems. The key challenge is to optimize design metrics, which is particularly difficult since those metrics compete with one another. One particularly difficult design metric to optimize is time-to-market, because embedded systems are growing in complexity at a tremendous rate, and the rate at which productivity improves every year is not keeping up with that growth. This book seeks to help improve productivity by describing design techniques that are standard and others that are very new, and by presenting a unified view of software and hardware design. This goal is worked towards by presenting three key technologies for embedded systems design: processor technology, IC technology, and design technology. Processor technology is divided into general-purpose, application-specific, and single-purpose processors. IC technology is divided into custom, semi-custom, and programmable logic IC's. Design technology is divided into compilation/synthesis, libraries/IP, and test/verification.

This book focuses on processor technology (both hardware and software), with the last couple of chapters covering IC and design technologies. Chapter 2 covers general-purpose processors. We focus on programming of such processors using structured programming languages, touching on assembly language for use in driver routines; we assume the reader already has familiarity with programming in both types of languages. Chapter 3 covers single-purpose processors, describing a number of common peripherals used in embedded systems. Chapter 4 describes digital design techniques for building custom single-purpose processors. Chapter 5 describes memories, components necessary to store data for processors. Chapters 6 and 7 describe buses, components necessary to communicate data among processors and memories, with Chapter 6 introducing concepts, and Chapter 7 describing common buses. Chapters 8 and 9 describe advanced techniques for programming embedded systems, with Chapter 8 focusing on state machines, and Chapter 9 providing an introduction to real-time programming. Chapter 10 introduces a very common form of embedded system, called control systems. Chapter 11 provides an overview of IC technologies, enough for a designer to understand what options are available and what tradeoffs exist. Chapter 12 focuses on design methodology, emphasizing the need for a "new breed" of engineers for embedded systems, proficient with both software and hardware design.

1.7 References and further reading

[1] Semiconductor Industry Association, National Technology Roadmap for Semiconductors, 1997.

1.8 Exercises

1. Consider the following embedded systems: a pager, a computer printer, and an automobile cruise controller. Create a table with each example as a column, and each row one of the following design metrics: unit cost, performance, size, and power. For each table entry, explain whether the constraint on the design metric is very tight. Indicate in the performance entry whether the system is highly reactive or not.
2. List three pairs of design metrics that may compete, providing an intuitive explanation of the reason behind the competition.
3. The design of a particular disk drive has an NRE cost of $100,000 and a unit cost of $20. How much will we have to add to the cost of the product to cover our NRE cost, assuming we sell: (a) 100 units, and (b) 10,000 units.

4. (a) Create a general equation for product cost as a function of unit cost, NRE cost, and number of units, assuming we distribute NRE cost equally among units. (b) Create a graph with the x-axis the number of units and the y-axis the product cost, and then plot the product cost function for an NRE of $50,000 and a unit cost of $5.
5. Redraw Figure 1.4 to show the transistors per IC from 1990 to 2000 on a linear, not logarithmic, scale. Draw a square representing a 1990 IC and another representing a 2000 IC, with correct relative proportions.
6. Create a plot with the three processor technologies on the x-axis, and the three IC technologies on the y-axis. For each axis, put the most programmable form closest to the origin, and the most customized form at the end of the axis. Plot the 9 points, and explain features and possible occasions for using each.
7. Give an example of a recent consumer product whose prime market window was only about one year.