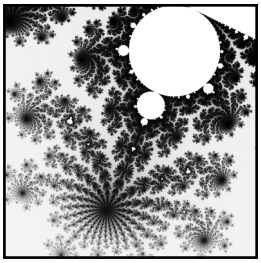
CHAPTER 1



The FPGA as a Computing Platform

As the cost per gate of FPGAs declines, embedded and high-performance systems designers are being presented with new opportunities for creating accelerated software applications using FPGA-based programmable hardware platforms. From a hardware perspective, these new platforms effectively bridge the gap between software programmable systems based on traditional microprocessors, and application-specific platforms based on custom hardware functions. From a software perspective, advances in design tools and methodology for FPGA-based platforms enable the rapid creation of hardware-accelerated algorithms.

The opportunities presented by these programmable hardware platforms include creation of custom hardware functions by software engineers, later design freeze dates, simplified field updates, and the reduction or elimination of custom chips from many categories of electronic products. Increasingly, systems designers are seeing the benefits of using FPGAs as the basis for applications that are traditionally in the domain of application-specific integrated circuits (ASICs). As FPGAs have grown in logic capacity, their ability to host high-performance software algorithms and complete applications has grown correspondingly.

In this chapter, we will present a brief overview of FPGAs and FPGA-based platforms and present the general philosophy behind using the C language for FPGA application development. Experienced FPGA users will find

much of this information familiar, but nonetheless we hope you stay with us as we take the FPGA into new, perhaps unfamiliar territory: that of high-performance computing.

1.1 A Quick Introduction to FPGAs

A field-programmable gate array (FPGA) is a large-scale integrated circuit that can be programmed after it is manufactured rather than being limited to a predetermined, unchangeable hardware function. The term "field-programmable" refers to the ability to change the operation of the device "in the field," while "gate array" is a somewhat dated reference to the basic internal architecture that makes this after-the-fact reprogramming possible.

FPGAs come in a wide variety of sizes and with many different combinations of internal and external features. What they have in common is that they are composed of relatively small blocks of programmable logic. These blocks, each of which typically contains a few registers and a few dozen low-level, configurable logic elements, are arranged in a grid and tied together using programmable interconnections. In a typical FPGA, the logic blocks that make up the bulk of the device are based on lookup tables (of perhaps four or five binary inputs) combined with one or two single-bit registers and additional logic elements such as clock enables and multiplexers. These basic structures may be replicated many thousands of times to create a large programmable hardware *fabric*.

In more complex FPGAs these general-purpose logic blocks are combined with higher-level arithmetic and control structures, such as multipliers and counters, in support of common types of applications such as signal processing. In addition, specialized logic blocks are found at the periphery of the devices that provide programmable input and output capabilities.

Common FPGA Characteristics

FPGAs are mainly characterized by their logic size (measured either by the number of transistors or, more commonly, by the number of fundamental logic blocks that they contain), by their logic structure and processing features, and by their speed and power consumption. While they range in size from a few thousand to many millions of logic gate equivalents, all FPGAs share the same basic characteristics:

- *Logic elements.* All FPGA devices are based on arrays of relatively small digital logic elements. To use such a device, digital logic problems must be decomposed into an arrangement of smaller logic circuits that can be mapped to one or more of these logic elements, or *logic cells*, through a

process of *technology mapping*. This technology mapping process may be either manual or automatic, but it involves substantial intelligence on the part of the human or (more typically) the software program performing the mapping.

- *Lookup tables.* FPGA logic elements are themselves usually composed of at least one programmable register (a flip-flop) and some input-forming logic, which is often implemented as a lookup table of n inputs, where n is generally five or less. The detailed structure of this logic element depends on the FPGA vendor (for example, Xilinx or Altera) and FPGA family (for example, Xilinx Virtex or Altera Cyclone). The lookup tables (LUTs) that make up most logic elements are very much akin to read-only memory (ROM). These LUTs are capable of implementing any combinational function of their inputs.
- *Memory resources.* Most modern FPGA devices incorporate some on-chip memory resources, such as SRAM. These memories may be accessible in a hierarchy, such as local memory within each logic element combined with globally shared memory blocks.
- *Routing resources.* Routing is the key to an FPGA's flexibility, but it also represents a compromise—made by the FPGA provider—between programming flexibility and area efficiency. Routing typically includes a hierarchy of channels that may range from high-speed, cross-chip *long lines* to more flexible block-to-block local connections. Programmable switches (which may be SRAM-based, electrically erasable, or one-time-programmable) connect the on-chip FPGA resources to each other, and to external resources as well.
- *Configurable I/O.* FPGA-based applications have a wide variety of system-level interface requirements and therefore FPGAs have many programmable I/O features. FPGA pins can be configured as TTL, CMOS, PCI, and more, allowing FPGAs to be interfaced with, and convert between, many different circuit technologies. Most FPGAs also have dedicated high-speed I/Os for clocks and global resets, and many FPGAs include PLLs and clock management schemes, allowing designs with multiple independent clock domains to be created and managed.

FPGA Programming Technologies

FPGA programming technologies range from one-time-programmable elements (such as those found in devices from Actel and Quicklogic) to electrically erasable or SRAM-based devices such as those available from Altera, Lattice Semiconductor, and Xilinx.

While most FPGAs in use today are programmed during system manufacturing to perform one specific task, it is becoming increasingly common

for FPGAs to be reprogrammed while the product is in the field. For SRAM-based and electrically erasable FPGAs this field upgrading may be as simple as providing an updated Flash memory card or obtaining a new binary device image from a website or CD-ROM.

Hardware applications implemented in FPGAs are generally slower and consume more power than the same applications implemented in custom ASICs. Nonetheless, the dramatically lowered risk and cost of development for FPGAs have made them excellent alternatives to custom ICs. The reduced development times associated with FPGAs often make them compelling platforms for ASIC prototyping as well.

Many modern FPGAs have the ability to be reprogrammed in-system, in whole or in part. This has led some researchers to create dynamically reconfigurable computing applications within one or more FPGAs in order to create extremely high-performance computing systems. The technology of reconfigurable computing is still in its infancy, however, due in large part to the high cost, in terms of power and configuration time, of dynamically reprogramming an FPGA. We will examine some of this research and make predictions regarding the future of reconfigurable computing in a later chapter.

Defining the behavior of an FPGA (the hardware that it contains) has traditionally been done either using a *hardware description language* (HDL) such as VHDL or Verilog or by arranging blocks of pre-existing functions, whether gate-level logic elements or higher-level macros, using a schematic- or block diagram-oriented design tool. More recently, design tools such as those described in this book that support variants of the C, C++, and Java languages have appeared. In any event, the result of the design entry process (and of design compilation and synthesis, as appropriate) is an intermediate file, called a netlist, that can be mapped to the actual FPGA architecture using proprietary FPGA place-and-route tools. After placement and mapping the resulting binary file—the bitmap—is downloaded to the FPGA, making it ready for use.

1.2 FPGA-Based Programmable Hardware Platforms

What constitutes a programmable hardware platform, and, in particular, one appropriate for high-performance computing? The term "platform" is somewhat arbitrary but generally refers to a known, previously verified hardware and/or software configuration that may be used as the basis for one or more specific applications. For our purposes, a programmable platform may be represented by a single FPGA (or other such programmable device), a complete board with multiple FPGAs, or even a development system such as a desktop PC or workstation that is used to emulate the behavior of an FPGA.

In short, a programmable hardware platform is a platform that includes at least one programmable hardware element, such as an FPGA, and that can implement all or part of a software algorithm. As you will see, you can also extend the definition of a platform to include various "soft" components that are found within the FPGA itself, such as embedded processors and related peripheral devices, and even such things as embedded operating systems running within the FPGA. All of these elements, taken as a whole, can be used to describe a particular FPGA-based platform.

FPGA-based platforms range from individual FPGAs, with or without embedded processors, and single-board, single-FPGA prototyping platforms to high-performance FPGA computing platforms consisting of multiple FPGAs combined with other processing resources on one or more boards. New FPGA-based platforms are being announced with increasing frequency, so the sample platforms appearing in this book represent only a small fraction of the FPGA-based solutions that are available.

Today's FPGAs are capable of much higher levels of system integration than those of previous generations. In particular, the ability to combine embedded processor cores and related standard peripheral devices with custom hardware functions (intellectual property [IP] blocks as illustrated in Figure 1-1) has made it possible to custom-craft a programmable platform ideally suited to a particular task or particular application domain.

The challenges of programmable platform-based design are primarily (but not exclusively) in the domain of the system architect and the software application developer. The success or failure of a development project—as measured in terms of development time and the final performance achieved—depends to a large degree on how well a large application and its constituent algorithms are mapped onto platform resources. To a software application developer it may be less than obvious which portions of the design should go into hardware (for example, in one or more FPGA devices) and which should be implemented as software on a traditional processor, whether a discrete device or a processor core embedded within an FPGA platform. Even more fundamental is the need for software application developers to consider their applications and algorithms in entirely new ways, and to make use of parallel programming techniques that may be unfamiliar to them in order to increase algorithmic concurrency and extract the maximum possible performance.

The fundamental decisions made by an application developer in the early phases of the design, the effectiveness of the algorithm partitioning, and the mapping of the application to hardware resources through the use of automated compiler and hardware synthesis tools can impact the final system's performance by orders of magnitude.

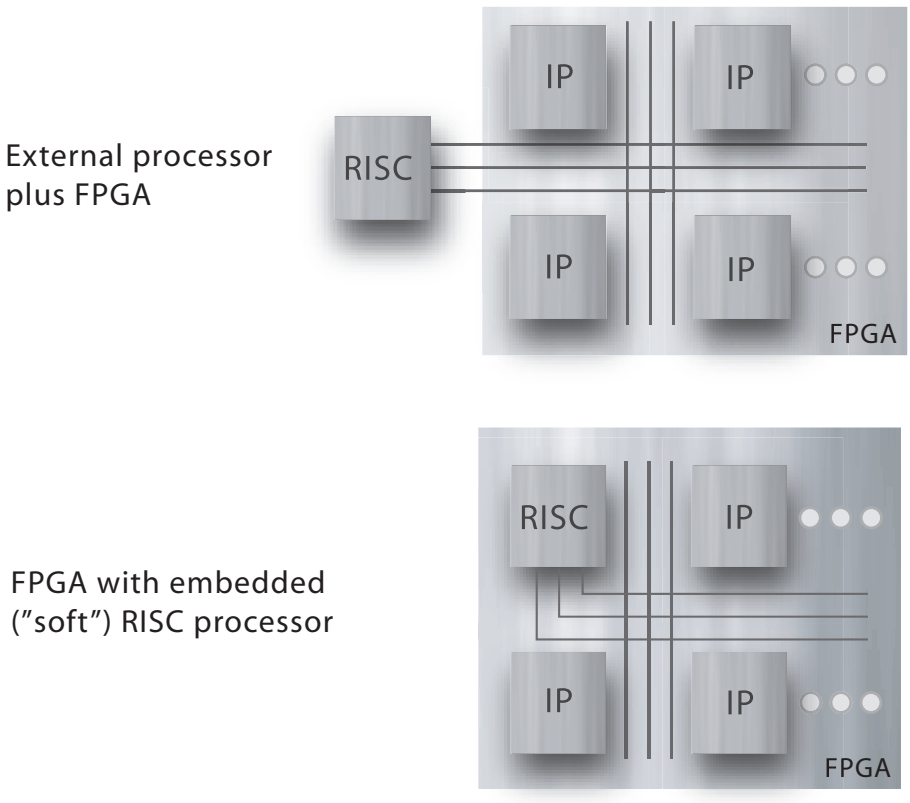


Figure 1-1. FPGA-based platforms may include embedded or adjacent microprocessors combined with application-specific blocks of intellectual property.

1.3 Increasing Performance While Lowering Costs

The decision to add an FPGA to a new or existing embedded system may be driven by the desire to extend the usefulness of a common, low-cost microprocessor (and avoid introducing a higher-end, specialized processor such as a DSP chip) or to eliminate the need for custom ASIC processing functions. In cases where the throughput of an existing system must increase to handle higher resolutions or larger signal bandwidths, the required performance increases may be primarily computational (requiring a scaling of computational resources) or may require completely new approaches to resolve bandwidth issues.

Digital signal processing (DSP) applications are excellent examples of the types of problems that can be effectively addressed using high-density

FPGAs. Implementing signal processing functions within an FPGA eliminates or reduces the need for an instruction-based processor. There has therefore been a steady rise in the use of FPGAs to handle functions that are traditionally the domain of DSP processors. System designers have been finding the cost/performance trade-offs tipping increasingly in favor of FPGA devices over high-performance DSP chips and—perhaps most significantly—when compared to the risks and up-front costs of a custom ASIC solution.

Most computationally intensive algorithms can be described using a relatively small amount of C source code, when compared to a hardware-level, equivalent description. The ability to quickly try out new algorithmic approaches from C-language models is therefore an important benefit of using a software-oriented approach to design. Reengineering low-level hardware designs, on the other hand, can be a tedious and error-prone process.

FPGAs help address this issue in two ways. First, they have the potential to implement high-performance applications as dedicated hardware without the up-front risk of custom ASICs. Mainstream, relatively low-cost FPGA devices now have the capacity and features necessary to support such applications. Second, and equally important, FPGAs are becoming dramatically easier to use due to advances in design tools. It is now possible to use multiple software-oriented, graphical, and language-based design methods as part of the FPGA design process.

When implementing signal processing or other computationally intensive applications, FPGAs may be used as prototyping vehicles with the goal of a later conversion to a dedicated ASIC or structured ASIC. Alternatively, FPGAs may be used as actual product platforms, in which case they offer unique benefits for field software upgrades and a compelling cost advantage for low- to medium-volume products.

High-capacity FPGAs also make sense for value engineering. In such cases, multiple devices (including processor peripherals and "glue" logic) may be consolidated into a single FPGA. While reduced size and system complexity are advantageous by-products of such consolidation, the primary benefit is lower cost. Using an FPGA as a catchall hardware platform is becoming common practice, but such efforts often ignore the benefits of using the FPGAs for primary processing as well as just using them for traditional hardware functions.

In many DSP applications, the best solution turns out to be a mixed processor design, in which the application's less-performance-critical components (including such elements as an operating system, network stack, and user interface) reside on a host microprocessor such as an ARM, PowerPC, or Pentium. More computationally intensive components reside in either a high-end DSP, in dedicated hardware in an FPGA, or in a custom ASIC as appropriate. It is not unusual, in fact, for such systems to include all three types of processing resources, allocated as needed. Developing such a system requires

multiple tools and knowledge of hardware design methods and tools but provides the greatest benefit in terms of performance per unit cost.

For each processor type in the system (standard processor, DSP, or FPGA), there are different advantages, disadvantages, and levels of required design expertise to consider. For example, while DSPs are software-programmable and require a low initial investment in tools, they require some expertise in DSP-specific design techniques and often require assembly-level programming skills. FPGAs, on the other hand, require a relatively large investment in design time and tools expertise, particularly when hardware design languages are used as the primary design input method.

When compared to the expertise and tools investment required for custom ASIC design, however, FPGAs are clearly the lower-cost, lower-risk solution for developing custom hardware.

FPGAs provide additional advantages related to the design process. By using an FPGA throughout the development process, a design team can incrementally port and verify algorithms that have been previously prototyped in software. This can be done manually (by hand-translating C code to lower-level HDL), but C-based design tools such as those described in this book can speed this aspect of the design process.

1.4 The Role of Tools

Software development tools, whether intended for deeply embedded systems or for enterprise applications, add value and improve the results of the application development process in two fundamental ways. First, a good set of tools provides an appropriate and easily understood abstraction of a target platform (whether an embedded processor, a desktop PC, or a supercomputer). A good abstraction of the platform allows software developers to create, test, and debug relatively portable applications while encouraging them to use programming methods that will result in the highest practical performance in the resulting end product.

The second fundamental value that tools provide is in the mechanical process of converting an application from its original high-level description (whether written in C or Java, as a dataflow diagram or in some other representation) into an optimized low-level equivalent that can be implemented—loaded and executed—on the target platform.

In an ideal tool flow, the specific steps of this process would be of no concern to the programmer; the application would simply operate at its highest possible efficiency through the magic of automated tools. In practice this is rarely the case: any programmer seeking high performance must have at least a rudimentary understanding of how the optimization, code generation,

and mapping process works, and must exert some level of control over the process either by adjusting the flow (specifying compiler options, for example) or by revisiting the original application and optimizing at the algorithm level, or both.

An Emphasis on Software-Based Methods

To fulfill the dual role of tools just described, emerging tools for automated hardware generation are focusing both on the automatic compilation/optimization problem and on delivering programming abstractions, or *programming models*, that make sense for the emerging classes of FPGA-based programmable platforms. All of these emerging tools focus on creating a software-oriented design experience. Software-oriented design tools are appropriate because

- Software provides a higher level of abstraction than traditional RTL design, thus helping to manage the growing complexity of platform-based systems.
- Algorithms are often specified, tested, and verified as software, so a software-oriented design environment requires fewer manual (and error-prone) translations.
- Microprocessors that inherently run software have become part of virtually every modern system. With emerging technologies enabling the ability to directly compile software into hardware implementations, software is becoming the lingua franca of system design.

Software-oriented programming, simulation, and debugging tools that provide appropriate abstractions of FPGA-based programmable platforms allow software and system designers to begin application development, experiment with alternative algorithms, and make critical design decisions without the need for specific hardware knowledge. This is of particular importance during design prototyping. As illustrated in Figures 1-2 and 1-3, the traditional hardware and software design process can be improved by introducing software-to-hardware design methods and tools. It is important to realize, however, that doing so will almost certainly not eliminate the need for hardware engineering skills. In fact, it is highly unlikely that a complete and well-optimized hardware/software application can be created using only software knowledge. On the plus side, it is certainly true that working prototypes can be more quickly generated using hardware and software design skills in combination with newly emerging tools for software-to-hardware compilation.

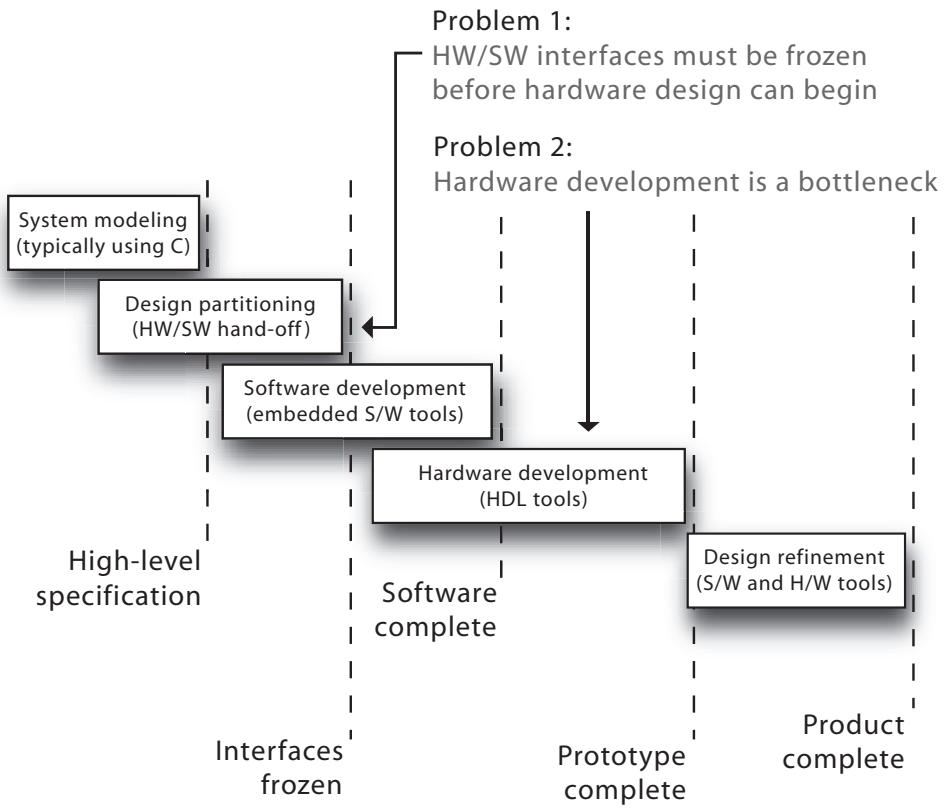


Figure 1-2. In a traditional hardware/software development process, hardware design may represent a significant bottleneck.

1.5 THE FPGA AS AN EMBEDDED SOFTWARE PLATFORM

Because of their reprogrammability, designing for FPGAs is conceptually similar to designing for common embedded processors. Simulation tools can be used to debug and verify the functionality of an application prior to actually programming a physical device, and there are tools readily available from FPGA vendors for performing in-system debugging. Although the tools are more complex and design processing times are substantially longer (it can take literally hours to process a large application through the FPGA place-and-route process), the basic design flow can be viewed as one of software rather than hardware development.

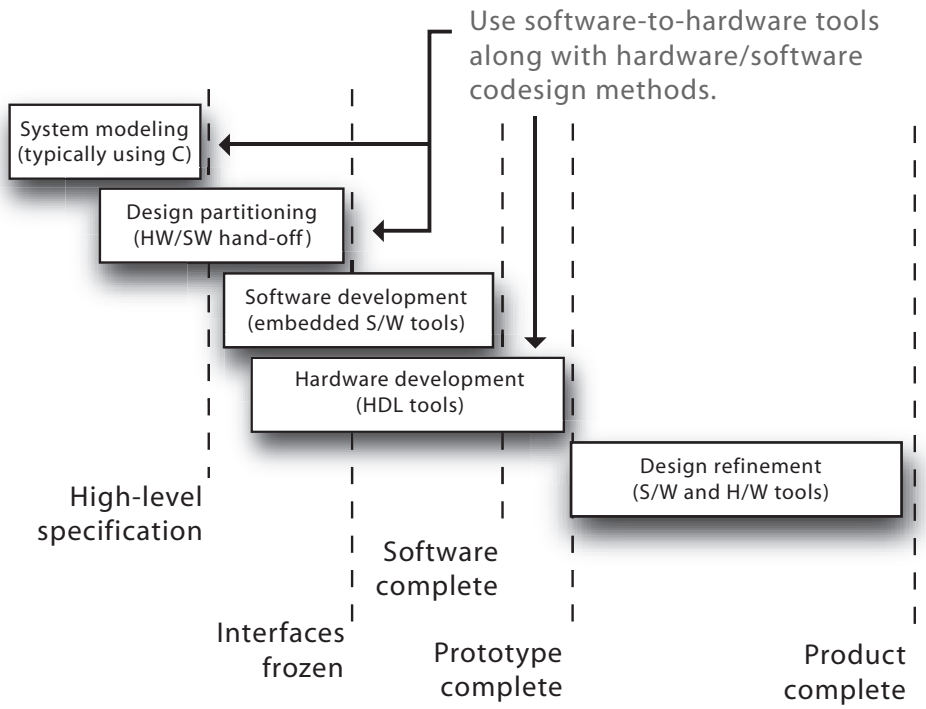


Figure 1-3. By introducing software-to-hardware compilation into the design process, it's possible to get to a working prototype faster, with more time available for later design refinement.

As any experienced FPGA application designer will tell you, however, the skills required to make the most efficient use of FPGAs, with all their low-level peculiarities and vendor-specific architectural features, are quite specialized and often daunting. To put this in the proper perspective, however, it's important to keep in mind that software development for embedded processors such as DSPs can also require specialized knowledge. DSP programmers, in fact, often resort to assembly language in order to obtain the highest possible performance and use C programming only in the application prototyping phase. The trend for both FPGA and processor application design has been to allow engineers to more quickly implement applications, or application prototypes, without the need to understand all the intricate details of the target, while at the same time providing access (through custom instructions, built-in functions/macros, assembly languages, and hardware description languages as appropriate) to low-level features for the purpose of extracting the maximum possible performance.

The Impulse C toolset used extensively in the book represents a relatively simple, C-based approach to hardware/software partitioning and hardware/software process synchronization that is coupled with efficient, FPGA-specific hardware compilers to produce a complete design path for mixed processor and FPGA implementations. This design flow has been integrated into traditional embedded and desktop programming tool flows without the need for custom behavioral simulators or other EDA-centric technologies. The combined environment enables systems-oriented development of highly parallel applications for FPGA-based platforms using the familiar C language, which may or may not be combined with existing HDL methods of design for those parts of an application that are not naturally expressed as software.

One of the key attributes of such a software-oriented system design flow is the ability to implement a design specification, captured in software, in the most appropriate platform resource. If the most appropriate resource is a microprocessor, this should be a simple matter of cross-compiling to that particular processor. If, however, the best-fitting resource is an FPGA, traditional flows would require a complete rewrite of the design into register transfer level (RTL) hardware description language. This is not only time-consuming, but also error-prone and acts as a significant barrier to the designer in exploring the entire hardware/software solution space. With a software-oriented flow, the design can simply be modified in its original language, no matter which resource is targeted.

1.6 THE IMPORTANCE OF A PROGRAMMING ABSTRACTION

As the line between what is hardware and what is software continues to blur, the skill sets of the hardware engineer and the software engineer have begun to merge. Hardware engineers have embraced language-based design (HDLs are simply a way to abstract a certain class of hardware as a written description). Software engineers—in particular, those developing embedded, multiprocessor applications—have started to introduce important hardware-oriented concepts such as concurrency and message-driven process synchronization to their design arsenals. Until recently, however, it has been difficult or impossible for software engineers to make the transition from software application programming to actual hardware implementation—to turn a conceptual application expressed in a high-level language into an efficient hardware description appropriate for the selected hardware target.

Developments in the area of high-level language compilers for hardware coupled with advances in behavioral synthesis technologies have helped ease this transition, but it remains challenging to create an efficient, mixed hard-

ware/software application without having substantial expertise in the area of RTL hardware design and in the optimization of applications at a very low level. Early adopters of commercial tools for C-based hardware synthesis have found that they must still drop to a relatively low level of abstraction to create the hardware/software interfaces and deal with the timing-driven nature of hardware design. The availability of high-level language synthesis has not yet eliminated the need for hardware design expertise. A major goal of this book, then, is to show when software-based design methods and tools for FPGAs are appropriate, while acknowledging that not every application is well-suited to such an approach.

Emerging research and technologies are making the FPGA design process easier and placing more control in the hands of software-savvy engineers. Improved software compilers and specialized optimizers for FPGAs and other hardware targets play an important role in this change, but as we've pointed out, higher-level system and algorithm design decisions can have more dramatic impacts than can be achieved through improvements in automated tools. No optimizer, regardless of its power, can improve an application that has been poorly conceived, partitioned, and optimized at an algorithmic level by the original programmer.

1.7 When Is C Language Appropriate for FPGA Design?

Experimenting with mixed hardware/software solutions can be a time-consuming process due to the historical disconnect between software development methods and the lower-level methods required for hardware design, including design for FPGAs. For many applications, the complete hardware/software design is represented by a collection of software and hardware source files that are not easily compiled, simulated, or debugged with a single tool set. In addition, because the hardware design process is relatively inefficient, hardware and software design cycles may be out of sync, requiring system interfaces and fundamental software/hardware partitioning decisions to be prematurely locked down.

With the advent of C-based FPGA design tools, however, it is now possible to use familiar software design tools and standard C language for a much larger percentage of a given application—in particular, those parts of the design that are computationally intensive. Later performance tweaks may introduce handcrafted HDL code as a replacement for the automatically generated hardware, just as DSP users often replace higher-level C code with handcrafted assembly language. Because the design can be compiled directly from C code to an initial FPGA implementation, however, the point at which a hardware engineer needs to be brought in to make such performance tweaks

is pushed further back in the design cycle, and the system as a whole can be designed using more productive software design methods.

These emerging hardware compiler tools allow C-language applications to be processed and optimized to create hardware, in the form of FPGA netlists, and also include the necessary C language extensions to allow highly parallel, multiple-process applications to be described. For target platforms that include embedded processors, these tools can be used to generate the necessary hardware/software interfaces as well as generating low-level hardware descriptions for specific processes.

One key to success with these tools, and with hardware/software approaches in general, is to partition the application appropriately between software and hardware resources. A good partitioning strategy must consider not only the computational requirements of a given algorithmic component, but also the data bandwidth requirements. This is because hardware/software interfaces may represent a significant performance bottleneck.

Making use of a programming model appropriate for highly parallel applications is also important. It is tempting to off-load specific functions to an FPGA using traditional programming methods such as remote procedure calls (whereby values are pushed onto some stack or stack equivalent, a hardware function is invoked, and the processor waits for a result) or by creating custom processor instructions that allow key calculations to be farmed out to hardware. Research has demonstrated, however, that alternate, more dataflow-oriented methods of programming are more efficient and less likely to introduce blockages or deadlocks into an application. In many cases, this means rethinking the application as a whole and finding new ways to express data movement and processing. The results of doing so, however, can be dramatic. By increasing application-level parallelism and taking advantage of programmable hardware resources, for example, it is possible to accelerate common algorithms by orders of magnitude over a software-only implementation.

During the development of such applications, design tools can be used to visualize and debug the interconnections of multiple parallel processes. Application monitoring, for example, can provide an overall view of the application and its constituent processes as it runs under the control of a standard C debugger. Such instrumentation can help quantify the results of a given partitioning strategy by identifying areas of high data throughput that may represent application bottlenecks. When used in conjunction with familiar software profiling methods, these tools allow specific areas of code to be identified for more detailed analysis or performance tweaking. The use of cycle-accurate or instruction-set simulators later in the development process can help further optimize the application.

What about SystemC?

SystemC is becoming increasingly popular as an alternative to existing hardware and system-level modeling languages—in particular, VHDL and Verilog.

SystemC provides (according to www.SystemC.org) "hardware-oriented constructs within the context of C++ as a class library implemented in standard C++." What this means is that SystemC includes features, implemented as C++ class and template libraries, that allow the definition of hardware behaviors at many levels, from the high-level representation of a complete system to the smallest details of transistor-level timing.

While SystemC does provide the features needed to describe parallel systems of interconnected processes, the complexity of the libraries coupled with a historically low acceptance of C++ as a language for embedded software development has limited its appeal to mainstream embedded systems programmers. This is not to say that SystemC is inappropriate for FPGA-based applications; on the contrary, most of the concepts and specific programming techniques covered in this book are applicable to SystemC FPGA design flows.

1.8 How to Use This Book

Our goal in writing this book is to make you a more productive designer of mixed hardware/software applications, using FPGAs as a vehicle, and to provide you with an alternate method of design entry, debugging, and compilation. We have not written this book with the goal of eliminating, or even reducing the importance of, existing hardware and software design methods.

If you are an experienced software developer or embedded systems designer who is considering FPGAs for hardware acceleration of key algorithms, this book is designed primarily, but not exclusively, with you in mind. We will introduce you to FPGAs, provide some history and background on their use as general-purpose hardware devices, and help you understand some of the advantages and pitfalls of introducing them into an embedded system as an additional computing resource. We will then spend a substantial amount of time introducing you to the concepts of parallel programming and creating mixed hardware/software applications using C.

If you are an experienced hardware developer currently using VHDL or Verilog, our goal is to help you use the C language as a complement to your

existing methods of design. Using C will allow you to more quickly generate prototype hardware for algorithms that have previously been expressed in C and give you the freedom to experiment with alternative methods of creating large, interconnected systems in hardware. This flexibility will in turn accelerate your development of custom hardware elements. The C language will also provide you with alternative methods of creating hardware/software test benches and related components for the purpose of unit or system testing.

Finally, both hardware and software developers can learn from this book about the demands of designing for the "other side" of the narrowing hardware/software divide. We hope that as development methods for hardware and software begin to converge, so too will the cooperation between hardware and software designers be improved.