INTRODUCTION

In the computer and electronics world, we are used to two different ways of performing computation: hardware and software. Computer hardware, such as application-specific integrated circuits (ASICs), provides highly optimized resources for quickly performing critical tasks, but it is permanently configured to only one application via a multimillion-dollar design and fabrication effort. Computer software provides the flexibility to change applications and perform a huge number of different tasks, but is orders of magnitude worse than ASIC implementations in terms of performance, silicon area efficiency, and power usage.

Field-programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. They implement circuits just like hardware, providing huge power, area, and performance benefits over software, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However, unlike in ASICs, these computations are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times.

Sometimes reprogramming is merely a bug fix to correct faulty behavior, or it is used to add a new feature. Other times, it may be carried out to reconfigure a generic computation engine for a new task, or even to reconfigure a device during operation to allow a single piece of silicon to simultaneously do the work of numerous special-purpose chips.

However, merging the benefits of both hardware and software does come at a price. FPGAs provide nearly all of the benefits of software flexibility and development models, and nearly all of the benefits of hardware efficiency—but not quite. Compared to a microprocessor, these devices are typically several orders of magnitude faster and more power efficient, but creating efficient programs for them is more complex. Typically, FPGAs are useful only for operations that process large streams of data, such as signal processing, networking, and the like. Compared to ASICs, they may be 5 to 25 times worse in terms of area, delay, and performance. However, while an ASIC design may take months to years to develop and have a multimillion-dollar price tag, an FPGA design might only take days to create and cost tens to hundreds of dollars. For systems that do not require the absolute highest achievable performance or power efficiency, an FPGA's development simplicity and the ability to easily fix bugs and upgrade functionality make them a compelling design alternative. For many tasks, and particularly for beginning electronics designers, FPGAs are the ideal choice.

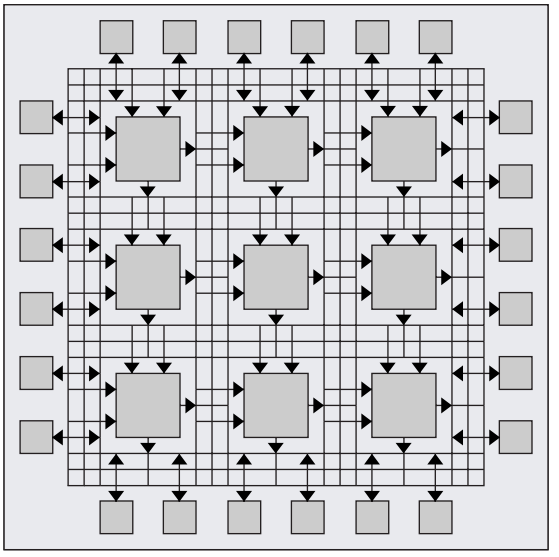

FIGURE I.1 ■ An abstract view of an FPGA; logic cells are embedded in a general routing structure.

Figure I.1 illustrates the internal workings of a field-programmable gate array, which is made up of logic blocks embedded in a general routing structure. This *array* of logic *gates* is the *G* and *A* in *FPGA*. The logic blocks contain processing elements for performing simple combinational logic, as well as flip-flops for implementing sequential logic. Because the logic units are often just simple memories, any Boolean combinational function of perhaps five or six inputs can be implemented in each logic block. The general routing structure allows arbitrary wiring, so the logical elements can be connected in the desired manner.

Because of this generality and flexibility, an FPGA can implement very complex circuits. Current devices can compute functions on the order of millions of basic gates, running at speeds in the hundreds of Megahertz. To boost speed and capacity, additional, special elements can be embedded into the array, such as large memories, multipliers, fast-carry logic for arithmetic and logic functions, and even complete microprocessors. With these predefined, fixed-logic units, which are fabricated into the silicon, FPGAs are capable of implementing complete systems in a single programmable device.

The logic and routing elements in an FPGA are controlled by programming points, which may be based on antifuse, Flash, or SRAM technology. For reconfigurable computing, SRAM-based FPGAs are the preferred option, and in fact are the primary style of FPGA devices in the electronics industry as a whole. In these devices, every routing choice and every logic function is controlled by a simple memory bit. With all of its memory bits programmed, by way of a configuration file or bitstream, an FPGA can be configured to implement the user's desired function. Thus, the configuration can be carried out quickly and

without permanent fabrication steps, allowing customization at the user's electronics bench, or even in the final end product. This is why FPGAs are *field programmable,* and why they differ from mask-programmable devices, which have their functionality fixed by masks during fabrication.

Because customizing an FPGA merely involves storing values to memory locations, similarly to compiling and then loading a program onto a computer, the creation of an FPGA-based circuit is a simple process of creating a bitstream to load into the device (see Figure I.2). Although there are tools to do this from software languages, schematics, and other formats, FPGA designers typically start with an application written in a hardware description language (HDL) such as Verilog or VHDL. This abstract design is optimized to fit into the FPGA's available logic through a series of steps: Logic synthesis converts high-level logic constructs and behavioral code into logic gates, followed by technology mapping to separate the gates into groupings that best match the FPGA's logic resources. Next, placement assigns the logic groupings to specific logic blocks and routing determines the interconnect resources that will carry the user's signals. Finally, bitstream generation creates a binary file that sets all of the FPGA's programming points to configure the logic blocks and routing resources appropriately.

After a design has been compiled, we can program the FPGA to perform a specified computation simply by loading the bitstream into it. Typically either a host microprocessor/microcontroller downloads the bitstream to the device, or an EPROM programmed with the bitstream is connected to the FPGA's configuration port. Either way, the appropriate bitstream must be loaded every time the FPGA is powered up, as well as any time the user wants to change the circuitry when it is running. Once the FPGA is configured, it operates as a custom piece of digital logic.

Because of the FPGA's dual nature—combining the flexibility of software with the performance of hardware—an FPGA designer must think differently from designers who use other devices. Software developers typically write sequential programs that exploit a microprocessor's ability to rapidly step through a series of instructions. In contrast, a high-quality FPGA design requires thinking about spatial parallelism—that is, simultaneously using multiple resources spread across a chip to yield a huge amount of computation.

Hardware designers have an advantage because they already think in terms of hardware implementations; even so, the flexibility of FPGAs gives them new opportunities generally not available in ASICs and other fixed devices. Field-programmable gate array designs can be rapidly developed and deployed, and even reprogrammed in the field with new functionality. Thus, they do not demand the huge design teams and validation efforts required for ASICs. Also, the ability to change the configuration, even when the device is running, yields new opportunities, such as computations that optimize themselves to specific demands on a second-by-second basis, or even time multiplexing a very large design onto a much smaller FPGA. However, because FPGAs are noticeably slower and have lower capacity than ASICs, designers must carefully optimize their design to the target device.

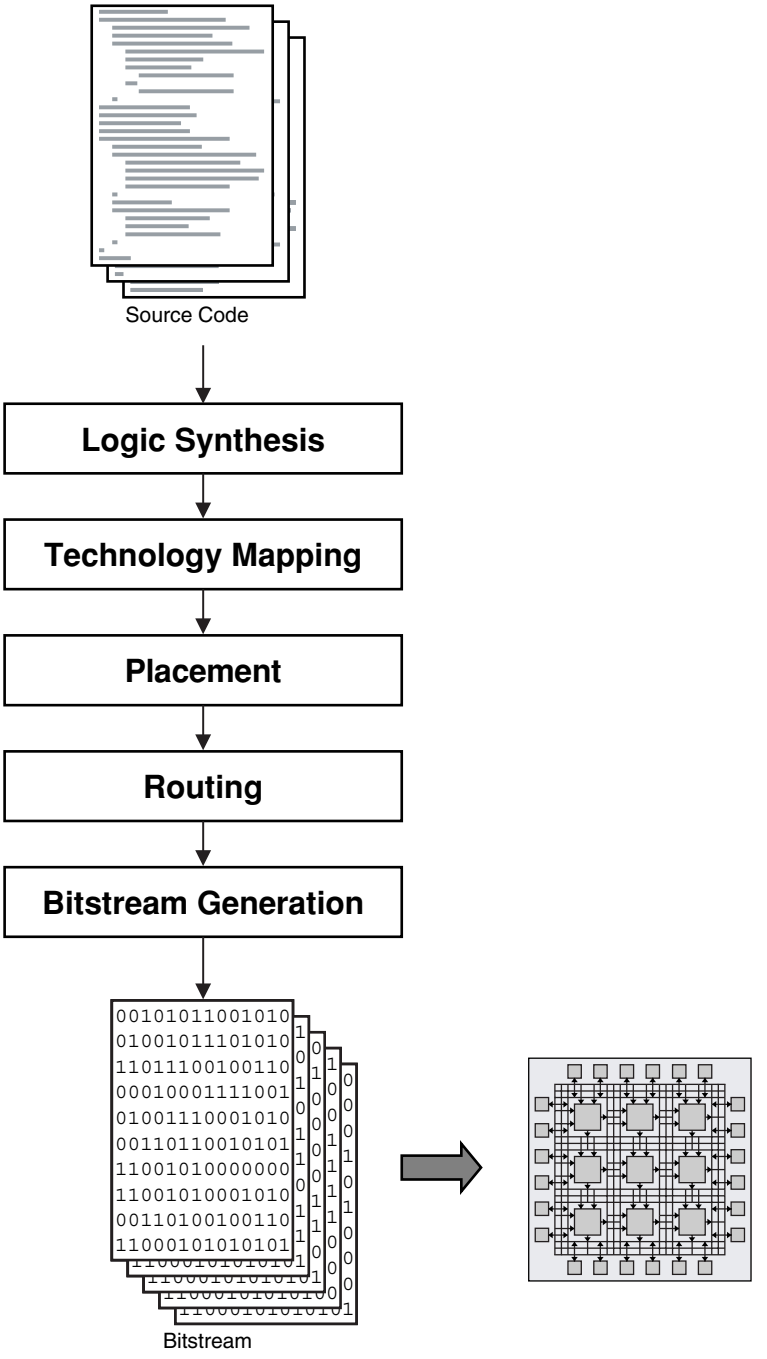

FIGURE I.2 ■ A typical FPGA mapping flow.

FPGAs are a very flexible medium, with unique opportunities and challenges. The goal of *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation* is to introduce all facets of FPGA-based systems—both positive and problematic. It is organized into six major parts:

- Part I introduces the hardware devices, covering both generic FPGAs and those specifically optimized for reconfigurable computing (Chapters 1 through 4).
- Part II focuses on programming reconfigurable computing systems, considering both their programming languages and programming models (Chapters 5 through 12).
- Part III focuses on the software mapping flow for FPGAs, including each of the basic CAD steps of Figure I.2 (Chapters 13 through 20).
- Part IV is devoted to application design, covering ways to make the most efficient use of FPGA logic (Chapters 21 through 26). This part can be viewed as a finishing school for FPGA designers because it highlights ways in which application development on an FPGA is different from both software programming and ASIC design.
- Part V is a set of case studies that show complete applications of reconfigurable logic (Chapters 27 through 35).
- Part VI contains more advanced topics, such as theoretical models and metric for reconfigurable computing, as well as defect and fault tolerance and the possible synergies between reconfigurable computing and nanotechnology (Chapters 36 through 38).

As the 38 chapters that follow will show, the challenges that FPGAs present are significant. However, the effort entailed in surmounting them is far outweighed by the unique opportunities these devices offer to the field of computing technology.