PART IV

APPLICATION DEVELOPMENT

Creating an efficient FPGA-based computation is similar to creating any other hardware. A designer carefully optimizes his or her computation to the needs of the underlying technology, exploiting the parallelism available while meeting resource and performance constraints. These designs are typically written in a hardware description language (HDL), such as Verilog, and CAD tools are then used to create the final implementation.

Field-programmable gate arrays (FPGAs) do have unique constraints and opportunities that must be understood in order for this technology to be employed most effectively. The resource mix is fixed, and the devices are never quite fast enough or have high enough capacity for what we want to do. However, because the chips are reprogrammable we can change the system in response to bugs or functionality upgrades, or even change the computation as it executes.

Because of the unique restrictions and opportunities inherent in FPGAs, a set of approaches to application development have proven critical to exploiting these devices to the fullest. Many of them are covered in the chapters that follow. Although not every FPGA-based application will use each of the approaches, a true FPGA expert will make them all part of his or her repertoire.

Some of the most challenging questions in the design process come at the very beginning of a new project: Are FPGAs a good match for the application? If so, what problems must be considered and overcome? Will runtime reconfiguration be part of the solution? Will fixed- or floating-point computation be used? Chapter 21 focuses on this level of design, covering the important issues that arise when we first consider an application and the problems that must be avoided or solved. It also offers a quick overview of application development. Chapters 22 through 26 delve into individual concerns in more detail.

FPGAs are unique in their potential to be more efficient than even ASICs for some types of problems: Because the circuit design is completely programmable, we can create a custom circuit not just for a given problem but for a specific problem *instance*. Imagine, for example, that we are creating an engine for solving Boolean equations (e.g., a SAT solver, discussed in Chapter 29 in Part V). In an ASIC design, we

would create a generic engine capable of handling any possible Boolean equation because each use of the chip would be for a different equation. In an FPGA-based system, the equation can be folded into the circuit mapping itself, creating a custom FPGA mapping optimized to solving that Boolean equation and no other. As long as there is a CPU available to dynamically create a new FPGA bitstream each time a new Boolean equation must be solved, a much more aggressively optimized design can be created. However, because this means that the time to create the new mapping is part of system execution, fast mapping algorithms are often the key (Chapter 20). This concept of instance-specific circuits is covered in Chapter 22.

In most cases, the time to create a completely new mapping in response to a specific problem instance is too long. Indeed, if it takes longer to create the custom circuit than for a generic circuit to solve the problem, the generic circuit is the better choice. However, more restricted versions of this style of optimization are still valuable. Consider a simple FIR filter, which involves multiplication of an incoming datastream with a set of constant coefficients. We could use a completely generic multiplier to handle the constant * variable computation. However, the bits of the constant are known in advance, so many parts of this multiplication can be simplified out. Multipliers, for example, generally compute a set of partial products—the result of multiplying one input with a single bit of the other input. These partial products are then added together. If the constant coefficient provided that single bit for a partial product, we can know at mapping creation time whether that partial product will be 0 or equal to the variable input—no hardware is necessary to create it. Also, in cases where the partial product is a 0, we no longer need to add it into the final result. In general, the use of constant inputs to a computation can significantly improve most metrics in FPGA mapping quality. These techniques, called constant propagation and partial evaluation, are covered in Chapter 22.

Number formats in FPGAs are another significant concern. For microprocessor-based systems we are used to treating everything as a 64-bit integer or an IEEE-format floating-point value. Because the underlying hardware is hardcoded to efficiently support these specific number formats, any other format is unlikely to be useful. However, in an FPGA we custom create the datapath. Thus, using a 64-bit adder on values that are at most 18 bits in length is wasteful because each bit position consumes one or more lookup tables (LUTs) in the device.

For this reason, an FPGA designer will carefully consider the required wordlength of the numbers in the system, hoping to shave off some bits of precision and thus reduce the hardware requirements of the design.

Fractional values, such as π or fractions of a second, are more problematic. In many cases, we can use a fixed-point format. We might use numbers in the range of $0 \ldots 31$ to represent the values from 0 to $\frac{31}{32}$ in steps of $\frac{1}{32}$ by just remembering that the number is actually scaled by a factor of 32. Techniques for addressing each of the concerns just mentioned are treated in Chapter 23.

Sometimes these optimizations simply are not possible, particularly for signals that require a high dynamic range (i.e., they must represent both very large and very small values simultaneously), so we need to use a floating-point format. This means that each operation will consume significantly more resources than its integer or fixed-point alternatives will. Chapter 31 in Part V covers floating-point operations on FPGAs in detail.

Once the number format is decided, it is important to determine how best to perform the actual computation. For many applications, particularly those from signal processing, the computation will involve a large number of constant coefficient multiplications and subsequent addition operations, such as in finite impulse response (FIR) filters. While these can be carried out in the normal, parallel adders and multipliers from standard hardware design, the LUT-based logic of an FPGA allows an even more efficient implementation. By converting to a bit–serial dataflow and storing the appropriate combination of constants into the LUTs in the FPGA, the multiply–accumulate operation can be compressed to a small table lookup and an addition. This technique, called distributed arithmetic, is covered in Chapter 24. It is capable of providing very efficient FPGA-based implementations of important classes of digital signal processing (DSP) and similar operations.

Complex mathematical operations such as sine, cosine, division, and square root, though less common than multiply–add, are still important in many applications. In some cases they can be handled by table lookup, with a table of precomputed results stored in memories inside the FPGA or in attached chips. However, as the size of the operand(s) for these functions grows, the size of the memory explodes, limiting this technique's effectiveness. A particularly efficient alternative in FPGA logic is the CORDIC algorithm. By the careful creation of an iterative circuit, FPGAs can efficiently compute many of these complex functions. The full details of the CORDIC algorithm, and its implementation in FPGAs, are covered in Chapter 25.

A final concern is the coupling of both FPGAs and central processing units (CPUs). In early systems, FPGAs were often deployed together with microprocessors or microcontrollers, either by placing an FPGA card in a host PC or by placing both resources on a single circuit board. With modern FPGAs, which can contain complete microprocessors

(either by mapping their logic into LUTs or embedding a complete microprocessor into the chip's silicon layout), the coupling of CPUs and FPGAs is even more attractive. The key driver is the relative advantages of each technology. FPGAs can provide very high performance for streaming applications with a lot of data parallelism—if we have to apply the same repetitive transformation to a large amount of data, an FPGA's performance is generally very high. However, for more sequential operations FPGAs are a poor choice. Sometimes long sequences of operations, with little or no opportunity for parallelism, come up in the control of the overall system. Also, exceptional cases do occur and must be handled—for example, the failure of a component, using denormal numbers in floating point, or interfacing to command-based peripherals. In each case a CPU is a much better choice for those portions of a computation. As a result, for many computations the best answer is to use the FPGA for the data-parallel kernels and a CPU for all the other operations. This process of segmenting a complete computation into software/CPU portions and hardware/FPGA portions is the focus of Chapter 26.