CHAPTER 21

IMPLEMENTING APPLICATIONS WITH FPGAS

Brad L. Hutchings, Brent E. Nelson
Department of Electrical and Computer Engineering
Brigham Young University

Developers can choose various devices when implementing electronic systems: field-programmable gate arrays (FPGAs), microprocessors, and other standard products such as ASSPs, and custom chips or application-specific integrated circuits (ASICs). This chapter discusses how FPGAs compare to other digital devices, outlines the considerations that will help designers to determine when FPGAs are appropriate for a specific application, and presents implementation strategies that exploit features specific to FPGAs.

The chapter is divided into four major sections. Section 21.1 discusses the strengths and weaknesses of FPGAs, relative to other available devices. Section 21.2 suggests when FPGA devices are suitable choices for specific applications/algorithms, based upon their I/O and computation requirements. Section 21.3 discusses general implementation strategies appropriate for FPGA devices. Then Section 21.4 discusses FPGA-specific arithmetic design techniques.

21.1 STRENGTHS AND WEAKNESSES OF FPGAs

Developers can choose from three general classes of devices when implementing an algorithm or application: microprocessor, FPGA, or ASIC (for simplicity, ASSPs are not considered here). This section provides a brief summary of the advantages and disadvantages of these devices in terms of time to market, cost, development time, power consumption, and debug and verification.

21.1.1 Time to Market

Time to market is often touted as one of the FPGA's biggest strengths, at least relative to ASICs. With an ASIC, from specification to product requires (at least): (1) design, (2) verification, (3) fabrication, (4) packaging, and (5) device test. In addition, software development requires access to the ASIC device (or an emulation of such) before it can be verified and completed. As immediately available standard devices, FPGAs have already been fabricated, packaged, and tested by the vendor, thereby eliminating at least four months from time to market.

More difficult to quantify but perhaps more important are: (1) refabrications (respins) caused by either errors in the design or late changes to the specification, due to a change in an evolving standard, for example, and (2) software development schedules that depend on access to the ASIC. Both of these items impact product production schedules; a respin can easily consume an additional four months, and early access to hardware can greatly accelerate software development and debug, particularly for the embedded software that communicates directly with the device.

In light of these considerations, a conservative estimate of the time-to-market advantage of FPGAs relative to ASICs is 6 to 12 months. Such a reduction is significant; in consumer electronics markets, many products have only a 24-month lifecycle.

21.1.2 Cost

Per device, FPGAs can be much less expensive than ASICs, especially in lower volumes, because the nonrecurring costs of FPGA fabrication are borne by many users. However, because of their reprogrammability, FPGAs require much more silicon area to implement equivalent functionality. Thus, at the highest volumes possible in consumer electronics, FPGA device cost will eventually exceed ASIC device cost.

21.1.3 Development Time

FPGA application development is most often approached as hardware design: applications are described in Verilog or VHDL, simulated to determine correctness, and synthesized using commercial logic synthesis tools. Commercial tools are available that synthesize behavioral programs written in sequential languages such as C to FPGAs. However, in most cases, much better performance and higher densities are achieved using HDLs, because they allow the user to directly describe and exploit the intrinsic parallelism available in an application. Exploiting application parallelism is the single best way to achieve high FPGA performance. However, designing highly parallel implementations of applications in HDLs requires significantly more development effort than software development with conventional sequential programming languages such as Java or C++.

21.1.4 Power Consumption

FPGAs consume more power than ASICs simply because programmability requires many more transistors, relative to a customized integrated circuit (IC). FPGAs may consume more or less power than a microprocessor or digital signal processor (DSP), depending on the application.

21.1.5 Debug and Verification

FPGAs are developed with standard hardware design techniques and tools. Coded in VHDL or Verilog and synthesized, FPGA designs can be debugged

in simulators just as typical ASIC designs are. However, many designers verify their designs directly, by downloading them into an FPGA and testing them in a system. With this approach the application can be tested at speed (a million times faster than simulation) in the actual operating environment, where it is exposed to real-world conditions. If thorough, this testing provides a stronger form of functional verification than simulation. However, debugging applications in an FPGA can be difficult because vendor tools provide much less observability and controllability than, for example, an hardware description language (HDL) simulator.

21.1.6 FPGAs and Microprocessors

As discussed previously, FPGAs are most often contrasted with custom ASICs. However, if a programmable solution is dictated because of changing application requirements or other factors, it is important to study the application carefully to determine if it is possible to meet performance requirements with a programmable processor—microprocessor or DSP. Code development for programmable processors requires much less effort than that required for FPGAs or ASICs, because developing software with sequential languages such as C or Java is much less taxing than writing parallel descriptions with Verilog or VHDL. Moreover, the coding and debugging environments for programmable processors are far richer than their HDL counterparts. Microprocessors are also generally much less expensive than FPGAs. If the microprocessor can meet application requirements (performance, power, etc.), it is almost always the best choice.

In general, FPGAs are well suited to applications that demand extremely high performance and reprogrammability, for interfacing components that communicate with many other devices (so-called glue-logic) and for implementing hardware systems at volumes that make their economies of scale feasible. They are less well suited to products that will be produced at the highest possible volumes or for systems that must run at the lowest possible power.

21.2 APPLICATION CHARACTERISTICS AND PERFORMANCE

Application performance is largely determined by the computational and I/O requirements of the system. Computational requirements dictate how much hardware parallelism can be used to increase performance. I/O system limitations and requirements determine how much performance can actually be exploited from the parallel hardware.

21.2.1 Computational Characteristics and Performance

FPGAs can outperform today's processors only by exploiting massive amounts of parallelism. Their technology has always suffered from a significant clock-rate disadvantage; FPGA clock rates have always been slower than CPU clock rates by about a factor of 10. This remains true today, with clock rates for FPGAs

limited to about 300 to 350 MHz and CPUs operating at approximately 3 GHz. As a result, FPGAs must perform at least 10 times the computational work per cycle to perform on par with processors. To be a compelling alternative, an FPGA-based solution should exceed the performance of a processor-based solution by 5 to 10 times and hence must actually perform 50 to 100 times the computational work per clock cycle. This kind of performance is feasible only if the target application exhibits a corresponding amount of exploitable parallelism.

The guideline of 5 to 10 times is suggested for two main reasons. First of all, prior to actual implementation, it is difficult or impossible to foresee the impact of various system and I/O issues on eventual performance. In our experience, 5 times can quickly become 2 times or less as various system and algorithmic issues arise during implementation. Second, application development for FPGAs is much more difficult than conventional software development. For that reason, the additional development effort must be carefully weighed against the potential performance advantages. A guideline of 5 to 10 times provides some insurance that any FPGA-specific performance advantages will not completely vanish during the implementation phase.

Ultimately, the intrinsic characteristics of the application place an upper bound on FPGA performance. They determine how much raw parallelism exists, how exploitable it is, and how fast the clock can operate. A review of the literature [3–6, 11, 16, 19–21, 23, 26, 28] shows that the application characteristics that have the most impact on application performance are: data parallelism, amenability to pipelining, data element size and arithmetic complexity, and simple control requirements.

Data parallelism

Large datasets with few or no data dependencies are ideal for FPGA implementation for two reasons: (1) They enable high performance because many computations can occur concurrently, and (2) they allow operations to be extensively rescheduled. As previously mentioned, concurrency is extremely important because FPGA applications must be able to achieve 50 to 100 times the operations per clock cycle of a microprocessor to be competitive. The ability to reschedule computations is also important because it makes it feasible to tailor the circuit design to FPGA hardware and achieve higher performance. For example, computations can be scheduled to maximize data reuse to increase performance and reduce memory bandwidth requirements. Image-processing algorithms with their attendant data parallelism have been among the highest-performing algorithms mapped to FPGA devices.

Data element size and arithmetic complexity

Data element size and arithmetic complexity are important because they strongly influence circuit size and speed. For applications with large amounts of exploitable parallelism, the upper limit on this parallelism is often determined by how many operations can be performed concurrently on the FPGA device. Larger data elements and greater arithmetic complexity lead to larger

and fewer computational elements and less parallelism. Moreover, larger and more complex circuits exhibit more delay that slows clock rate and impacts performance. Not surprisingly, representing data with the fewest possible bits and performing computation with the simplest operators generally lead to the highest performance. Designing high-performance applications in FPGAs almost always involves a precision/performance trade-off.

Pipelining
Pipelining is essential to achieving high performance in FPGAs. Because FPGA performance is limited primarily by interconnect delay, pipelining (inserting registers on long circuit pathways) is an essential way to improve clock rate (and therefore throughput) at the cost of latency. In addition, pipelining allows computational operations to be overlapped in time and leads to more parallelism in the implementation. Generally speaking, because pipelining is used extensively throughout FPGA-based designs, applications must be able to tolerate some latency (via pipelining) to be suitable candidates for FPGA implementation.

Simple control requirements
FPGAs achieve the highest performance if all operations can be statically scheduled as much as possible (this is true of many technologies). Put simply, it takes time to make decisions and decision-making circuitry is often on the critical path for many algorithms. Replacing runtime decision circuitry with static control eliminates circuitry and speeds up execution. It makes it much easier to construct circuit pipelines that are heavily utilized with few or no pipeline bubbles. In addition, statically scheduled controllers require less circuitry, making room for more datapath operators, for example. In general, datasets with few or no dependencies often have simple control requirements.

21.2.2 I/O and Performance

As mentioned previously, FPGA clock rates are at least one order of magnitude slower than those of CPUs. Thus, significant parallelism (either data parallelism or pipelining) is required for an FPGA to be an attractive alternative to a CPU. However, I/O performance is just as important: Data must be transmitted at rates that can keep all of the parallel hardware busy.

Algorithms can be loosely grouped into two categories: I/O bound and compute bound [17, 18]. At the simplest level, if the number of I/O operations is equal to or greater than the number of calculations in the computation, the computation is said to be I/O bound. To increase its performance requires an increase in memory bandwidth—doing more computation in parallel will have no effect. Conversely, if the number of computations is greater than the number of I/O operations, computational parallelism may provide a speedup.

A simple example of this, provided by Kung [18], is matrix–matrix multiplication. The total number of I/Os in the computation, for *n-by-n* matrices, is $3n^2$—each matrix must be read and the product written back. The total number of computations to be done, however, is n^3. Thus, this computation is

compute bound. In contrast, matrix–matrix addition requires $3n^2$ I/Os and $3n^2$ calculations and is thus I/O bound. Another way to see this is to note that each source element read from memory in a matrix–matrix multiplication is used n times and each result is produced using n multiply–accumulate operations. In matrix–matrix addition, each element fetched from memory is used only once and each result is produced from only a single addition.

Carefully coordinating data transfer, I/O movement, and computation order is crucial to achieving enough parallelism to provide effective speedup. The entire field of systolic array design is based on the concepts of (1) arranging the I/O and computation in a compute-bound application so that each data element fetched from memory is reused multiple times, and (2) keeping many processing elements busy operating in parallel on that data.

FPGAs offer a wide variety of memory elements that can be used to coordinate I/O and computation: flip-flops to provide single-bit storage (10,000s of bits); LUT-based RAM to provide many small blocks of randomly distributed memory (100,000s of bits); and larger RAM or ROM memories (1,000,000s of bits). Some vendors' FPGAs contain multiple sizes of random access memories, and these memories are often easily configured into special-purpose structures such as dynamic-length shift registers, content-addressable memories (CAMs), and so forth. In addition to these types of on-chip memory, most FPGA platforms provide off-chip memory as well.

Increasing the I/O bandwidth to memory is usually critical in harnessing the parallelism inherent in a computation. That is, after some point, further multiplying the number of processing elements (PEs) in a design (to increase parallelism) usually requires a corresponding increase in I/O. This additional I/O can often be provided by the many on-chip memories in a typical modern FPGA. The work of Graham and Nelson [8] describes a series of early experiments to map time-delay SONAR beam forming to an FPGA platform where memory bandwidth was the limiting factor in design speedup. While the data to be processed were an infinite stream of large data blocks, many of the other data structures in the computation were not large (e.g., coefficients, delay values). In this computation, it was not *the total amount of memory* that limited the speedup but rather the *number of memory ports available*. Thus, the use of multiple small memories in parallel were able to provide the needed bandwidth.

The availability of many small memories in today's FPGAs further supports the idea of trading off computation for table lookup. Conventional FPGA fabrics are based on a foundation of 4-input LUTs; in addition, larger on-chip memories can be used to support larger lookup structures. Because the memories already exist on chip, unlike in ASIC technology, using them adds no additional cost to the system. A common approach in FPGA-based design, therefore, is to evaluate which parts of the system's computations might lend themselves to table lookup and use the available RAM blocks for these lookups.

In summary, the performance of FPGA-based applications is largely determined by how much exploitable parallelism is available, and by the ability of the system to provide data to keep the parallel hardware operational.

21.3 GENERAL IMPLEMENTATION STRATEGIES FOR FPGA-BASED SYSTEMS

In contrast with other programmable technologies such as microprocessors or DSPs, FPGAs provide an extremely rich and complex set of implementation alternatives. Designers have complete control over arithmetic schemes and number representation and can, for example, trade precision for performance. In addition, reprogrammable, SRAM-based FPGAs can be configured any number of times to provide additional implementation flexibility for further tailoring the implementation to lower cost and make better use of the device.

There are two general configuration strategies for FPGAs: configure-once, where the application consists of a single configuration that is downloaded for the duration of the application's operation, and runtime reconfiguration (RTR), where the application consists of multiple configurations that are "swapped" in and out as the application operates [14].

21.3.1 Configure-once

Configure-once (during operation) is the simplest and most common way to implement applications with reconfigurable logic. The distinctive feature of configure-once applications is that they consist of a single system-wide configuration. Prior to operation, the FPGAs comprising the reconfigurable resource are loaded with their respective configurations. Once operation commences, they remain in this configuration until the application completes. This approach is very similar to using an ASIC for application acceleration. From the application point of view, it matters little whether the hardware used to accelerate the application is an FPGA or a custom ASIC because it remains constant throughout its operation.

The configure-once approach can also be applied to reconfigurable applications to achieve significant acceleration. There are classes of applications, for example, where the input data varies but remains constant for hours, days, or longer. In some cases, data-specific optimizations can be applied to the application circuitry and lead to dramatic speedup. Of course, when the data changes, the circuit-specific optimizations need to be reapplied and the bitstream regenerated. Applications of this sort consist of two elements: (1) the FPGA and system hardware, and (2) an application-specific compiler that regenerates the bitstream whenever the application-specific data changes. This approach has been used, for example, to accelerate SNORT, a popular packet filter used to improve network security [13]. SNORT data consists of regular expressions that detect malicious packets by their content. It is relatively static, and new regular expressions are occasionally added as new attacks are detected. The application-specific compiler translates these regular expressions into FPGA hardware that matches packets many times faster than software SNORT. When new regular expressions are added to the SNORT database, the compiler is rerun and a new configuration is created and downloaded to the FPGA.

21.3.2 Runtime Reconfiguration

Whereas configure-once applications statically allocate logic for the duration of an application, RTR applications use a dynamic allocation scheme that re-allocates hardware at runtime. Each application consists of *multiple* configurations per FPGA, with each one implementing some fraction of it. Whereas a configure-once application configures the FPGA once before execution, an RTR application typically reconfigures it many times during the normal operation.

There are two basic approaches that can be used to implement RTR applications: *global* and *local* (sometimes referred to as partial configuration in the literature). Both techniques use multiple configurations for a single application, and both reconfigure the FPGA during application execution. The principal difference between the two is the way the dynamic hardware is allocated.

Global RTR

Global RTR allocates *all* (FPGA) hardware resources in each configuration step. More specifically, global RTR applications are divided into distinct temporal phases, with each phase implemented as a single system-wide configuration that occupies all system FPGA resources. At runtime, the application steps through each phase by loading all of the system FPGAs with the appropriate configuration data associated with a given phase.

Local RTR

Local RTR takes an even more flexible approach to reconfiguration than does global RTR. As the name implies, these applications *locally* (or selectively) reconfigure subsets of the logic as they execute. Local RTR applications may configure any percentage of the reconfigurable resources at any time, individual FPGAs may be configured, or even single FPGA devices may themselves be partially reconfigured on demand. This flexibility allows hardware resources to be tailored to the runtime profile of the application with finer granularity than that possible with global RTR. Whereas global RTR approaches implement the execution process by loading relatively large, global application partitions, local RTR applications need load only the necessary functionality at each point in time. This can reduce the amount of time spent downloading configurations and can lead to a more efficient runtime hardware allocation.

The organization of local RTR applications is based more on a *functional* division of labor than the phased partitioning used by global RTR applications. Typically, local RTR applications are implemented by functionally partitioning an application into a set of fine-grained operations. These operations need not be temporally exclusive—many of them may be active at one time. This is in direct contrast to global RTR, where only one configuration (per FPGA) may be active at any given time. Still, with local RTR it is important to organize the operations such that idle circuitry is eliminated or greatly reduced. Each operation is implemented as a distinct circuit module, and these circuit modules are then downloaded to the FPGAs as necessary during operation. Note that, unlike global RTR, several of these operations may be loaded simultaneously, and each may consume any portion of the system FPGA resources.

RTR applications

Runtime Reconfigured Artificial Neural Network (RRANN) is an early example of a global RTR application [7]. RRANN divided the back-propagation algorithm (used to train neural networks) into three temporally exclusive configurations that were loaded into the FPGA in rapid succession during operation. It demonstrated a 500 percent increase in density by eliminating idle circuitry in individual algorithm phases.

RRANN was followed up with RRANN-2 [9], an application using local RTR. Like RRANN, the algorithm was still divided into three distinct phases. However, unlike the earlier version, the phases were carefully designed so that they shared common circuitry, which was placed and routed into identical physical locations for each phase. Initially, only the first configuration was loaded; thereafter, the common circuitry remained resident and only circuit differences were loaded during operation. This reduced configuration overhead by 25 percent over the global RTR approach.

The Dynamic Instruction Set Computer (DISC) [29] used local RTR to create a sequential control processor with a very small fixed core that remained resident at all times. This resident core was augmented by circuit modules that were dynamically loaded as required by the application. DISC was used to implement an image-processing application that consisted of various filtering operations. At runtime, the circuit modules were loaded as necessary. Although the application used all of the filtering circuit modules, it did not require all of them to be loaded simultaneously. Thus, DISC loaded circuit modules on demand as required. Only a few active circuit modules were ever resident at any time, allowing the application to fit in a much smaller device than possible with global RTR.

21.3.3 Summary of Implementation Issues

Of the two general implementation techniques, configure-once is the simplest and is best supported by commercially available tool flows. This is not surprising, as all FPGA CAD tools are derivations of conventional ASIC CAD flows. While the two RTR implementation approaches (local and global) can provide significant performance and capacity advantages, they are much more challenging to employ, primarily because of a lack of specific tool support.

The designer's primary task when implementing global RTR applications is to temporally divide the application into roughly equal-size partitions to efficiently use reconfigurable resources. This is largely a manual process—although the academic community has produced some partitioning tools, no commercial offerings are currently available. The main disadvantage of global RTR is the need for equal-size partitions. If it is not possible to evenly partition the application, inefficient use of FPGA resources will result.

The main advantage of local RTR over global RTR is that it uses fine-grained functional operators that may make more efficient use of FPGA resources. This is important for applications that are not easily divided into equal-size temporally exclusive circuit partitions. However, partitioning a local RTR design may require an inordinate amount of designer effort. For example, unlike global

RTR, where circuit interfaces typically remain fixed between configurations, local RTR allows these interfaces to change with each configuration. When circuit configurations become small enough for multiple configurations to fit into a single device, the designer needs to ensure that all configurations will *interface* correctly one with another. Moreover, the designer may have to ensure not only structural compliance but *physical* compliance as well. That is, when the designer creates circuit configurations that do not occupy an entire FPGA, he or she will have to ensure that the physical footprint of each is compatible with that of others that may be loaded concurrently.

21.4 IMPLEMENTING ARITHMETIC IN FPGAs

Almost since their invention, FPGAs have employed dedicated circuitry to accelerate arithmetic computation. In earlier devices, dedicated circuitry sped up the propagation of carry signals for ripple-carry, full-adder blocks. Later devices added dedicated multipliers, DSP function blocks, and more complex fixed-function circuitry. The presence of such dedicated circuitry can dramatically improve arithmetic performance, but also restricts designers to a very small subset of choices when implementing arithmetic.

Well-known approaches such as carry-look-ahead, carry-save, signed-digit, and so on, generally do not apply to FPGAs. Though these techniques are commonly used to create very high-performance arithmetic blocks in custom ICs, they are not competitive when applied to FPGAs simply because they cannot access the faster, dedicated circuitry and must be constructed using slower, general-purpose user logic. Instead, FPGA designers accelerate arithmetic in one of two ways with FPGAs: (1) using dedicated blocks if they fit the needs of the application, and (2) avoiding the computation entirely, if possible. Designers apply the second option by, for example, replacing full-blown floating-point computation with simpler, though not equivalent, fixed-point, or block floating-point, computations. In some cases, they can eliminate multiplication entirely with constant propagation. Of course, the feasibility of replacing slower, complex functions with simpler, faster ones is application dependent.

21.4.1 Fixed-point Number Representation and Arithmetic

A fixed-point number representation is simply an integer representation with an implied binary point, usually in 2's complement format to enable the representation of both positive and negative values. A common way of describing the structure of a fixed-point number is to use a tuple: n, m, where n is the number of bits to the left of the binary point and m is the number of bits to the right. A 16.0 format would thus be a standard 16-bit integer; a 3.2 format fixed-point number would have a total of 5 bits with 3 to the left of the implied binary point and 2 to the right. A range of numbers from +1 to −1A is common in digital signal-processing applications. Such a representation might be of the

form 1.9, where the largest number is 0.111111111 = 0.99810 and the smallest is $1.000000000 = -1_{10}$. As can be seen, fixed-point arithmetic exactly follows the rules learned in grade school, where lining up the implied binary point is required for performing addition or subtraction.

When designing with fixed-point values, one must keep track of the number format on each wire; such bookkeeping is one of the design costs associated with fixed-point design. At any point in a computation, either truncation or rounding can be used to reduce the number of bits to the right of the binary point, the effect being to simply reduce the precision with which the number is represented.

21.4.2 Floating-point Arithmetic

Floating-point arithmetic overcomes many of the challenges of fixed-point arithmetic but at increased circuit cost and possibly reduced precision. The most common format for a floating-point number is of the form *seeeeeffffff*, where *s* is a sign bit, *eeeee* is an exponent, and *ffffff* is the mantissa. In the IEEE standard for single-precision floating point, the number of exponent bits is 8 and the number of mantissa bits is 23, but nonstandard sizes and formats have also been used in FPGA work [2, 24].

IEEE reserves various combinations of exponent and mantissa to represent special values: zero, not a number (NAN), infinity (+8 and −8), and so on. It supports denormalized numbers (no leading implied 1 in the mantissa) and flags them using a special exponent value. Finally, the IEEE specification describes four rounding modes. Because supporting all special case number representations and rounding modes in hardware can be very expensive, FPGA-based floating-point support often omits some of them in the interest of reducing complexity and increasing performance.

For a given number of bits, floating point provides extended *range* to a computation at the expense of *accuracy*. An IEEE single-precision floating-point number allocates 23 bits to the mantissa, giving an effective mantissa of only 24 bits when the implied 1 is considered. The advantage of floating point is that its exponent allows for the representation of numbers across a broad range (IEEE normalized single-precision values range from $\approx \pm 3 \times 10^{38}$ to $\approx \pm 1 \times 10^{-38}$). Conversely, while a 32-bit fixed-point representation (1.31 format) has a range of only −1 to ≈+1, it can represent some values within that range much more accurately than a floating-point format can—for example, numbers close to +1 such as 0.11111111111111111111111111111111. However, for numbers very close to +0, the fixed-point representation would have many leading zeroes, and thus would have *less* precision than the competing floating-point representation.

An important characteristic of floating point is its auto-scaling behavior. After every floating-point operation, the result is normalized and the exponent adjusted accordingly. No work on the part of the designer is required in this respect (although significant hardware resources are used). Thus, it is useful in cases where the range of intermediate values cannot be bounded by the designer and therefore where fixed point is unsuitable.

The use of floating point in FPGA-based design has been the topic of much research over the past decade. Early papers, such as Ligon and colleagues [15] and Shirazi et al. [24], focused on the cost of floating point and demonstrated that small floating-point formats as well as single-precision formats could be eventually implemented using FPGA technology. Later work, such as that by Bellows and Hutchings [1] and Roesler and Nelson [22], demonstrated novel ways of leveraging FPGA-specific features to more efficiently implement floating-point modules. Finally, Underwood [27] argued that the capabilities of FPGA-based platforms for performing floating point would eventually surpass those of standard computing systems.

All of the research just mentioned contains size and performance estimates for floating-point modules on FPGAs at the time they were published. Clever design techniques and growing FPGA densities and clock rates continually combine to produce smaller, faster floating-point circuits on FPGAs. At the time of this writing, floating-point module libraries are available from a number of sources, both commercial and academic.

21.4.3 Block Floating Point

Block floating point (BFP) is an alternative to fixed-point and floating-point arithmetic that allows entire blocks of data to share a single exponent. Fixed-point arithmetic is then performed on a block of data with periodic rescaling of its data values. A typical use of block floating point is as follows:

1. The largest value in a block of data is located, a corresponding exponent is chosen, and that value's fractional part is normalized to that exponent.
2. The mantissas of all other values in the block are adjusted to use the same exponent as that largest value.
3. The exponent is dropped and fixed-point arithmetic proceeds on the resulting values in the data block.
4. As the computation proceeds, renormalization of the entire block of data occurs—after every individual computation, only when a value overflows, or after a succession of computations.

The key is that BFP allows for growth in the range of values in the data block while retaining the low cost of fixed-point computations. Block floating point has found extensive use in fast Fourier transform (FFT) computations where an input block (such as from an A/D converter) may have a limited range of values, the data is processed in stages, and stage boundaries provide natural renormalization locations.

21.4.4 Constant Folding and Data-oriented Specialization

As mentioned Section 21.3.2, when the data for a computation changes, an FPGA can be readily reconfigured to take advantage of that change. As a simple example of data folding, consider the operation: $a =?b$, where a and b are 4-bit

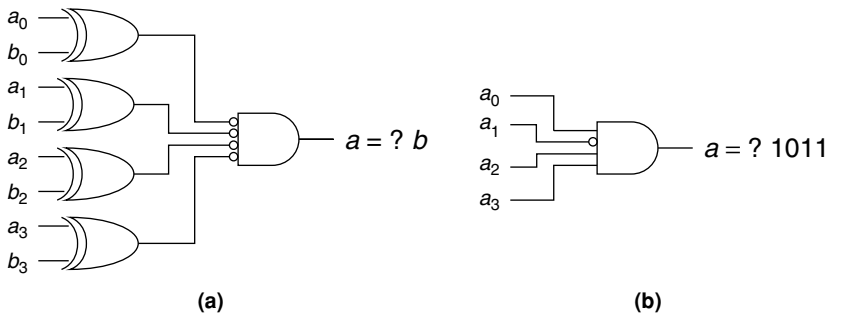

FIGURE 21.1 ■ Two comparator implementations: (a) with and (b) without constant folding.

numbers. Figure 21.1 shows two implementations of a comparator. On the left (a) is a conventional comparator; on the right (b) is a comparator that may be used when b is known ($b = 1011$). Implementation (a) requires three 4-LUTs to implement while implementation (b) requires just one. Such logic-level constant folding is usually performed by synthesis tools.

A more complex example is given by Wirthlin [30], who proposed a method for creating constant coefficient multipliers. When one constant to a multiplier was known, a custom multiplier consuming far fewer resources than a general multiplier could usually be created. Wirthlin's manipulations [30], going far beyond what logic optimization performed, created a custom structure for a given multiplier instance based on specific characteristics of the constant.

Hemmert et al. [10] offer an even more complex example in which a pipeline of image morphology processing stages was created, each of which could perform one image morphology step (e.g., one iteration in an erosion operation). The LUT contents in each pipeline stage controlled the stage's operation; thus, reconfiguring a stage required modifying only LUT programming. A compiler was then created to convert programs, written in a special image morphology language, into the data required to customize each pipeline stage's operation.

When a new image morphology program was compiled, a new bitstream for the FPGA could be created in a second or two (by directly modifying the original bitstream) and reconfigured onto the platform. This provided a way to create a custom computing solution on a per-program basis with turnarounds on the order of a few seconds. In each case, the original morphology program that was compiled provided the constant data that was folded into the design.

Additional examples in the literature show the power of constant folding. However, its use typically requires specialized CAD support. Slade and Nelson [25] argue that a fundamentally different approach to CAD for FPGAs is the solution to providing generalized support for such data-specific specialization. They advocate the use of JHDL [1, 12] to provide deployment time support for data-specific modifications to an operating FPGA-based system.

In summary, FPGAs provide architectural features that can accelerate simple arithmetic operations such as fixed-point addition and multiplication.

Floating-point operations can be accelerated using block floating point or by reducing the number of bits to represent floating-point values. Finally, constants can be propagated into arithmetic circuits to reduce circuit area and accelerate arithmetic performance.

21.5 SUMMARY

FPGAs provide a flexible, high-performance, and reprogrammable means for implementing a variety of electronic applications. Because of their reprogrammability, they are well suited to applications that require some form of direct reprogrammability, and to situations where reprogrammability can be used indirectly to increase reuse and thereby reduce device cost or count. FPGAs achieve the highest performance when the application can be implemented as many parallel hardware units operating in parallel, and where the aggregate I/O requirements for these parallel units can be reasonably met by the overall system. Most FPGA applications are described using HDLs because HDL tools and synthesis software are mature and well developed, and because, for now, they provide the best means for describing applications in a highly parallel manner.

Once FPGAs are determined to be a suitable choice, there are several ways to tailor the system design to exploit their reprogrammability by reconfiguring them at runtime or by compiling specific, temporary application-specific data into the FPGA circuitry. Performance can be further enhanced by crafting arithmetic circuitry to work around FPGA limitations and to exploit the FPGA's special arithmetic features. Finally, FPGAs provide additional debug and verification methods that are not available in ASICs and that enable debug and verification to occur in a system and at speed.

In summary, FPGAs combine the advantages and disadvantages of microprocessors and ASICs. On the positive side, they can provide high performance that is achievable only with custom hardware, they are reprogrammable, and they can be purchased in volume as a fully tested, standard product. On the negative side, they remain largely inaccessible to the software community; moreover, high-performance application development requires hardware design and the use of standard synthesis tools and Verilog or VHDL.

References

[1] P. Bellows, B. L. Hutchings. JHDL—An HDL for reconfigurable systems. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1998.

[2] B. Catanzaro, B. Nelson. Higher radix floating-point representations for FPGA-based arithmetic. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 2005.

[3] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider. Exploring architectures for volume visualization on the Teramac custom computer. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[4] A. Dandalis, V. K. Prasanna. Fast parallel implementation of DFT using configurable devices. Field-programmable logic: Smart applications, new paradigms, and compilers. *Proceedings 6th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, 1997.

[5] C. H. Dick, F. Harris. FIR filtering with FPGAs using quadrature sigma-delta modulation encoding. Field-programmable logic: Smart applications, new paradigms, and compilers. *Proceedings 6th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag 1996.

[6] C. Dick. Computing the discrete Fourier transform on FPGA-based systolic arrays. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1996.

[7] J. G. Eldredge, B. L. Hutchings. Density enhancement of a neural network using FPGAs and runtime reconfiguration. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[8] P. Graham, B. Nelson. FPGA-based sonar processing. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1998.

[9] J. D. Hadley, B. L. Hutchings. Design methodologies for partially reconfigured systems. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[10] S. Hemmert, B. Hutchings, A. Malvi. An application-specific compiler for high-speed binary image morphology. *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[11] R. Hudson, D. Lehn, P. Athanas. A runtime reconfigurable engine for image interpolation. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, April 1998.

[12] B. L. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting. A CAD suite for high-performance FPGA design. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1999.

[13] B. L. Hutchings, R. Franklin, D. Carver. Assisting network intrusion detection with reconfigurable hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, April 2002.

[14] B. L. Hutchings, M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. *Field-Programmable Logic and Applications*, August 1995.

[15] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, K. D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[16] W. E. King, T. H. Drayer, R. W. Conners, P. Araman. Using MORPH in an industrial machine vision system. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[17] H. T. Kung. Why Systolic Architectures? *IEEE Computer* 15(1), 1982.

[18] S. Y. Kung. *VLSI Array Processors*, Prentice-Hall, 1988.

[19] T. Moeller, D. R. Martinez. Field-programmable gate array based radar front-end digital signal processing. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Mac*hines, April 1999.

[20] G. Panneerselvam, P. J. W. Graumann, L. E. Turner. Implementation of fast Fourier transforms and discrete cosine transforms in FPGAs. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.

[21] R. J. Petersen. *An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing*, Master's thesis, Brigham Young University, 1995.

[22] E. Roesler, B. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, August 2002.

[23] N. Shirazi, P. M. Athanas, A. L. Abbott. Implementation of a 2D fast Fourier transform on an FPGA-based custom computing machine. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.

[24] N. Shirazi, A. Walters, P. Athanas. Quantitative analysis of floating point arithmetic on FPGA-based custom computing machines. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[25] A. Slade, B. Nelson. Reconfigurable computing application frameworks. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[26] L. E. Turner, P. J. W. Graumann, S. G. Gibb. Bit-serial FIR filters with CSD coefficients for FPGAs. *Fifth International Workshop on Field-Programmable Logic and Applications*, September 1995.

[27] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. *Proceedings of the ACM/SIGDA 12th International Symposium on Field-Programmable Gate Arrays*, 2004.

[28] J. E. Vuillemin. On computing power. Programming languages and system architectures. *Lecture Notes in Computer Science*, vol. 781, Springer-Verlag, 1994.

[29] M. J. Wirthlin, B. L. Hutchings (eds). A dynamic instruction set computer. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[30] M. J. Wirthlin. Constant coefficient multiplication using look-up tables. *Journal of VLSI Signal Processing* 36, 2004.