CHAPTER 1

DEVICE ARCHITECTURE

Mark L. Chang
Electrical and Computer Engineering
Franklin W. Olin College of Engineering

The best race car drivers understand how their cars work. The best architects know how carpenters, bricklayers, and electricians do their jobs. And the best programmers know how the hardware they are programming does computation. Knowing how your device works, "down to the metal," is essential for efficient utilization of available resources.

In this chapter, we take a look inside the package to discover the basic hardware elements that make up a typical field-programmable gate array (FPGA). We'll talk about how computation happens in an FPGA—from the blocks that do the computation to the interconnect that shuttles data from one place to another. We'll talk about how these building blocks fit together in terms of FPGA architecture. And, of course, because programmability (as well as reprogrammability) is part of what makes an FPGA so useful, we'll spend some time on that, too. Finally, we'll take an in-depth look at the architectures of some commercially available FPGAs in Section 1.5, Case Studies.

We won't be covering many of the research architectures from universities and industry—we'll save that for later. We also won't be talking much about how you successfully program these things to make them useful parts of a computational platform. That, too, is later in the book.

What you *will* learn is what's "under the hood" of a typical commercial FPGA so that you will become more comfortable using it as a platform for solving problems and performing computations. The first step in our journey starts with how computation in an FPGA is done.

1.1 LOGIC—THE COMPUTATIONAL FABRIC

Think of your typical desktop computer. Inside the case, among other things, are storage and communication devices (hard drives and network cards), memory, and, of course, the central processing unit, or CPU, where most of the computation happens. The FPGA plays a similar role in a reconfigurable computing platform, but we're going to break it down.

In very general terms, there are only two types of resources in an FPGA: *logic* and *interconnect*. Logic is where we do things like arithmetic, `1+1=2`, and logical functions, `if (ready) x=1 else x=0`. Interconnect is how we get data (like the

results of the previous computations) from one node of computation to another. Let's focus on logic first.

1.1.1 Logic Elements

From your digital logic and computer architecture background, you know that any computation can be represented as a Boolean equation (and in some cases as a Boolean equation where inputs are dependent on past results—don't worry, FPGAs can hold state, too). In turn, any Boolean equation can be expressed as a truth table. From these humble beginnings, we can build complex structures that can do arithmetic, such as adders and multipliers, as well as decision-making structures that can evaluate conditional statements, such as the classic if-then-else. Combining these, we can describe elaborate algorithms *simply by using truth tables*.

From this basic observation of digital logic, we see the truth table as the computational heart of the FPGA. More specifically, one hardware element that can easily implement a truth table is the lookup table, or LUT. From a circuit implementation perspective, a LUT can be formed simply from an *N*:1 (*N*-to-one) multiplexer and an *N*-bit memory. From the perspective of our previous discussion, a LUT simply enumerates a truth table. Therefore, using LUTs gives an FPGA the generality to implement arbitrary digital logic. Figure 1.1 shows a typical *N*-input lookup table that we might find in today's FPGAs. In fact, almost all commercial FPGAs have settled on the LUT as their basic building block.

The LUT can compute any function of *N* inputs by simply programming the lookup table with the truth table of the function we want to implement. As shown in the figure, if we wanted to implement a 3-input exclusive-or (XOR) function with our 3-input LUT (often referred to as a 3-LUT), we would assign values to the lookup table memory such that the pattern of select bits chooses the correct row's "answer." Thus, every "row" would yield a result of 0 except in the four cases where the XOR of the three select lines yields 1.

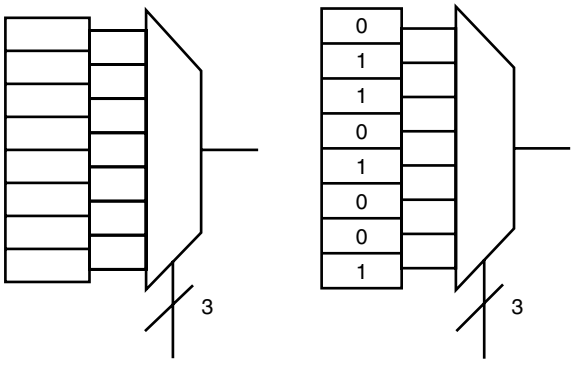

FIGURE 1.1 ■ A 3-LUT schematic (a) and the corresponding 3-LUT symbol and truth table (b) for a logical XOR.

Of course, more complicated functions, and functions of a larger number of inputs, can be implemented by aggregating several lookup tables together. For example, one can organize a single 3-LUT into an 8×1 ROM, and if the values of the lookup table are reprogrammable, an 8×1 RAM. But the basic building block, the lookup table, remains the same.

Although the LUT has more or less been chosen as the smallest computational unit in commercially available FPGAs, the size of the lookup table in each logic block has been widely investigated [1]. On the one hand, larger lookup tables would allow for more complex logic to be performed per logic block, thus reducing the wiring delay between blocks as fewer blocks would be needed. However, the penalty paid would be slower LUTs, because of the requirement of larger multiplexers, and an increased chance of waste if not all of the functionality of the larger LUTs were to be used. On the other hand, smaller lookup tables may require a design to consume a larger number of logic blocks, thus increasing wiring delay between blocks while reducing per–logic block delay.

Current empirical studies have shown that the 4-LUT structure makes the best trade-off between area and delay for a wide range of benchmark circuits. Of course, as FPGA computing evolves into wider arenas, this result may need to be revisited. In fact, as of this writing, Xilinx has released the Virtex-5 SRAM-based FPGA with a 6-LUT architecture.

The question of the number of LUTs per logic block has also been investigated [2], with empirical evidence suggesting that grouping more than one 4-LUT into a single logic block may improve area and delay. Many current commercial FPGAs incorporate a number of 4-LUTs into each logic block to take advantage of this observation.

Investigations into both LUT size and number of LUTs per block begin to address the larger question of computational *granularity* in an FPGA. On one end of the spectrum, the rather simple structure of a small lookup table (e.g., 2-LUT) represents *fine-grained* computational capability. Toward the other end, *coarse-grained*, one can envision larger computational blocks, such as full 8-bit arithmetic logic units (ALUs), more typical of CPUs. As in the case of lookup table sizing, finer-grained blocks may be more adept at bit-level manipulations and arithmetic, but require combining several to implement larger pieces of logic. Contrast that with coarser-grained blocks, which may be more optimal for datapath-oriented computations that work with standard "word" sizes (8/16/32 bits) but are wasteful when implementing very simple logical operations. Current industry practice has been to strike a balance in granularity by using rather fine-grained 4-LUT architectures and augmenting them with coarser-grained heterogeneous elements, such as multipliers, as described in the Extended Logic Elements section later in this chapter.

Now that we have chosen the logic block, we must ask ourselves if this is sufficient to implement all of the functionality we want in our FPGA. Indeed, it is not. With just LUTs, there is no way for an FPGA to maintain any sense of state, and therefore we are prohibited from implementing any form of sequential, or state-holding, logic. To remedy this situation, we will add a simple single-bit storage element in our base logic block in the form of a D flip-flop.

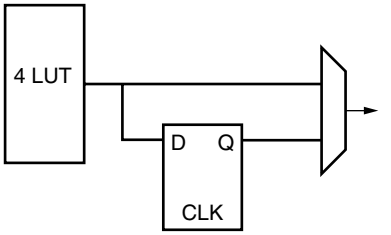

FIGURE 1.2 ■ A simple lookup table logic block.

Now our logic block looks something like Figure 1.2. The output multiplexer selects a result either from the function generated by the lookup table or from the stored bit in the D flip-flop. In reality, this logic block bears a very close resemblance to those in some commercial FPGAs.

1.1.2 Programmability

Looking at our logic block in Figure 1.2, it is a simple task to identify all the programmable points. These include the contents of the 4-LUT, the select signal for the output multiplexer, and the initial state of the D flip-flop. Most current commercial FPGAs use volatile static-RAM (SRAM) bits connected to configuration points to configure the FPGA. Thus, simply writing a value to each configuration bit sets the configuration of the entire FPGA.

In our logic block, the 4-LUT would be made up of 16 SRAM bits, one per output; the multiplexer would use a single SRAM bit; and the D flip-flop initialization value could also be held in a single SRAM bit. How these SRAM bits are initialized in the context of the rest of the FPGA will be the subject of later sections.

1.2 THE ARRAY AND INTERCONNECT

With the LUT and D flip-flop, we begin to define what is commonly known as the *logic block*, or *function block*, of an FPGA. Now that we have an understanding of how computation is performed in an FPGA at the single logic block level, we turn our focus to how these computation blocks can be tiled and connected together to form the fabric that is our FPGA.

Current popular FPGAs implement what is often called *island-style* architecture. As shown in Figure 1.3, this design has logic blocks tiled in a two-dimensional array and interconnected in some fashion. The logic blocks form the islands and "float" in a sea of interconnect.

With this array architecture, computations are performed spatially in the fabric of the FPGA. Large computations are broken into 4-LUT-sized pieces and mapped into physical logic blocks in the array. The interconnect is configured to route signals between logic blocks appropriately. With enough logic blocks, we can make our FPGAs perform any kind of computation we desire.

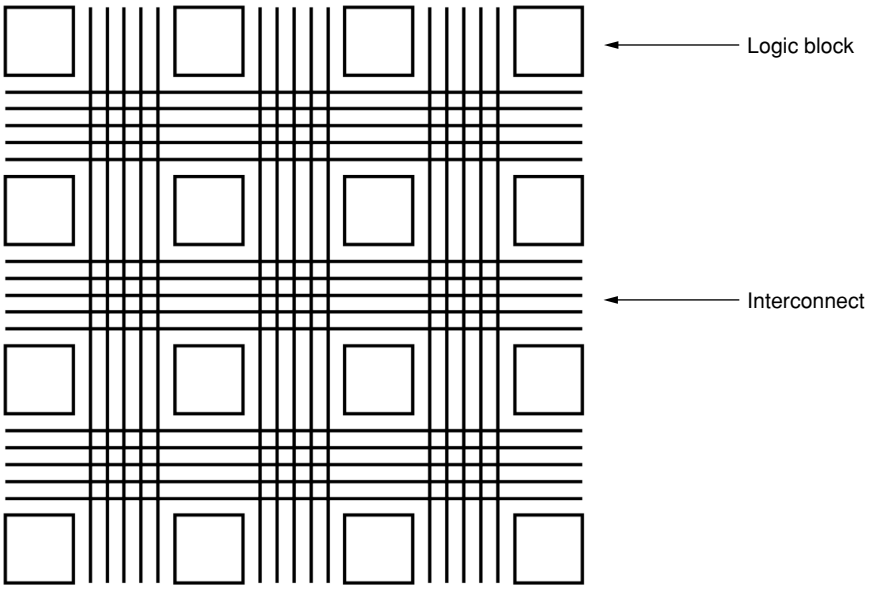

FIGURE 1.3 ■ The island-style FPGA architecture. The interconnect shown here is not representative of structures actually used.

1.2.1 Interconnect Structures

Figure 1.3 does not tell the whole story. The interconnect structure shown is not representative of any structures used in actual FPGAs, but is more of a cartoon placeholder. This section introduces the interconnect structures present in many of today's FPGAs, first by considering a small area of interconnection and then expanding out to understand the need for different styles of interconnect. We start with the simplest case of nearest-neighbor communication.

Nearest neighbor

Nearest-neighbor communication is as simple as it sounds. Looking at a 2×2 array of logic blocks in Figure 1.4, one can see that the only needs in this neighborhood are input and output connections in each direction: north, south, east, and west. This allows each logic block to communicate directly with each of its immediate neighbors.

Figure 1.4 is an example of one of the simplest routing architectures possible. While it may seem nearly degenerate, it has been used in some (now obsolete) commercial FPGAs. Of course, although this is a simple solution, this structure suffers from severe delay and connectivity issues. Imagine, instead of a 2×2 array, a 1024×1024 array. With only nearest-neighbor connectivity, the delay scales linearly with distance because the signal must go through many cells (and many switches) to reach its final destination.

From a connectivity standpoint, without the ability to bypass logic blocks in the routing structure, all routes that are more than a single hop away require

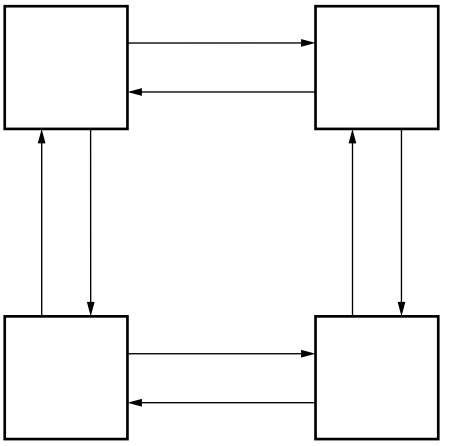

FIGURE 1.4 ■ Nearest-neighbor connectivity.

traversing a logic block. With only one bidirectional pair in each direction, this limits the number of logic block signals that may cross. Signals that are passing through must not overlap signals that are being actively consumed and produced.

Because of these limitations, the nearest-neighbor structure is rarely used *exclusively*, but it is almost always available in current FPGAs, often augmented with some of the techniques that follow.

Segmented

As we add complexity, we begin to move away from the pure logic block architecture that we've developed thus far. Most current FPGA architectures look less like Figure 1.3 and more like Figure 1.5.

In Figure 1.5 we introduce the connection block and the switch box. Here the routing structure is more generic and meshlike. The logic block accesses nearby communication resources through the connection block, which connects logic block input and output terminals to routing resources through programmable switches, or multiplexers. The connection block (detailed in Figure 1.6) allows logic block inputs and outputs to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility.

The switch block appears where horizontal and vertical routing tracks converge as shown in Figure 1.7. In the most general sense, it is simply a matrix of programmable switches that allow a signal on a track to connect to another track. Depending on the design of the switch block, this connection could be, for example, to turn the corner in either direction or to continue straight. The design of switch blocks is an entire area of research by itself and has produced many varied designs that exhibit varying degrees of connectivity and efficiency [3–5]. A detailed discussion of this research is beyond the scope of this book.

With this slightly modified architecture, the concept of a segmented interconnect becomes more clear. Nearest-neighbor routing can still be accomplished, albeit through a pair of connect blocks and a switch block. However, for

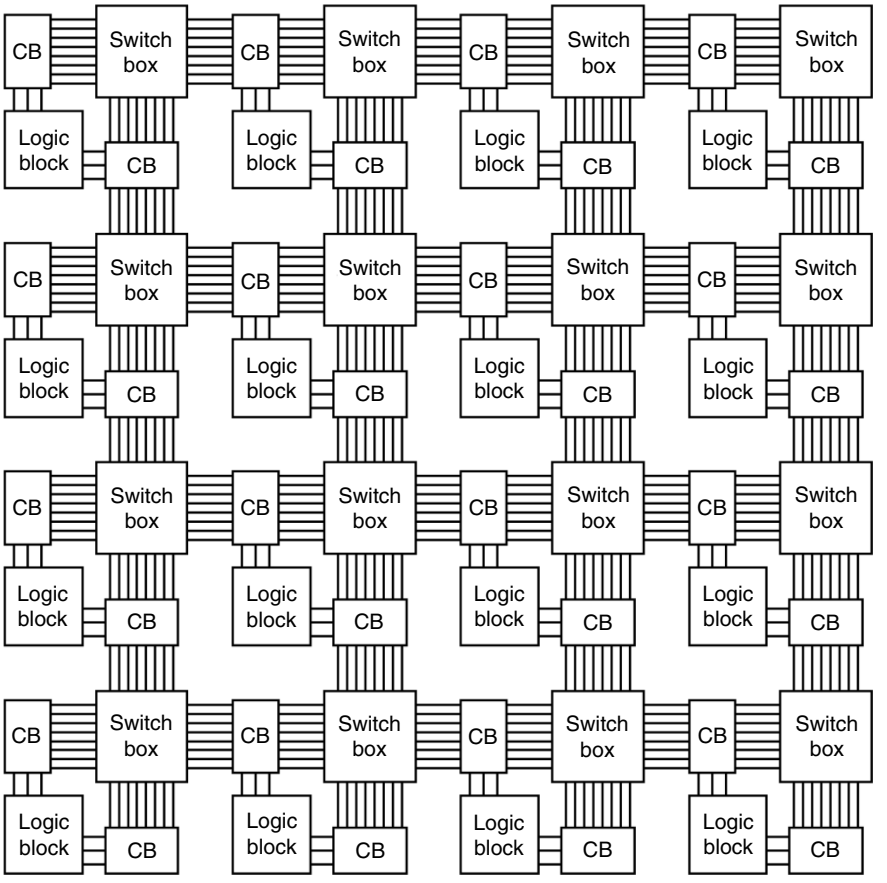

FIGURE 1.5 ■ An island-style architecture with connect blocks and switch boxes to support more complex routing structures. (The difference in relative sizes of the blocks is for visual differentiation.)

signals that need to travel longer distances, individual segments can be switched together in a switch block to connect distant logic blocks together. Think of it as a way to emulate long signal paths that can span arbitrary distances. The result is a long wire that actually comprises shorter "segments."

This interconnect architecture alone does not radically improve on the delay characteristics of the nearest-neighbor interconnect structure. However, the introduction of connection blocks and switch boxes separates the interconnect from the logic, allowing long-distance routing to be accomplished without consuming logic block resources.

To improve on our structure, we introduce longer-length wires. For instance, consider a wire that spans one logic block as being of length-1 (L1). In some segmented routing architectures, longer wires may be present to allow signals to travel greater distances more efficiently. These segments may be

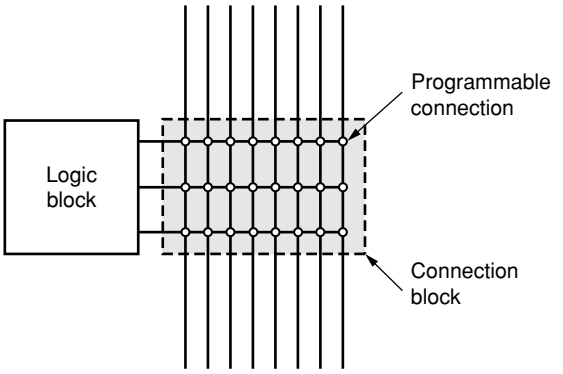

FIGURE 1.6 ■ Detail of a connection block.

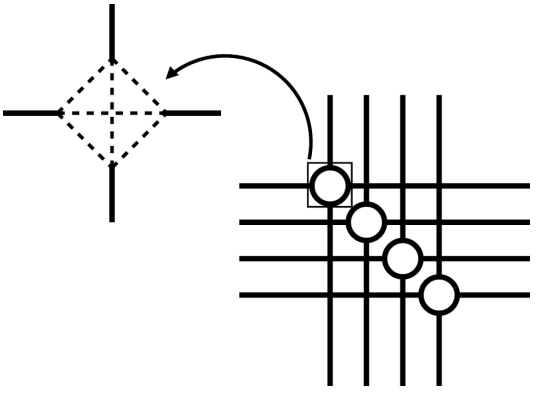

FIGURE 1.7 ■ An example of a common switch block architecture.

length-4 (L4), length-8 (L8), and so on. The switch blocks (and erhaps more embedded switches) become points where signals can switch from shorter to longer segments. This feature allows signal delay to be less than $O(N)$ when covering a distance of N logic blocks by reducing the number of intermediate switches in the signal path.

Figure 1.8 illustrates augmenting the single-segment interconnect with two additional lengths: direct-connect between logic blocks and length-2 (L2) lines. The direct-connect lines leave general routing resources free for other uses, and L2 lines allow signals to travel longer distances for roughly the same amount of switch delay. This interconnect architecture closely matches that of the Xilinx XC4000 series of commercial FPGAs.

Hierarchical

A slightly different approach to reducing the delay of long wires uses a hierarchical approach. Consider the structure in Figure 1.9. At the lowest level of hierarchy, 2×2 arrays of logic blocks are grouped together as a single cluster.

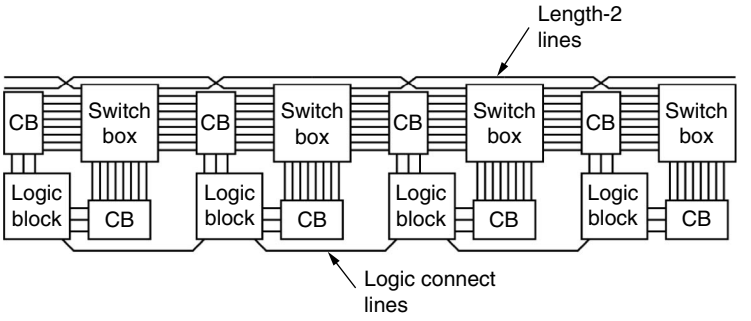

FIGURE 1.8 ■ Local (direct) connections and L2 connections augmenting a switched interconnect.

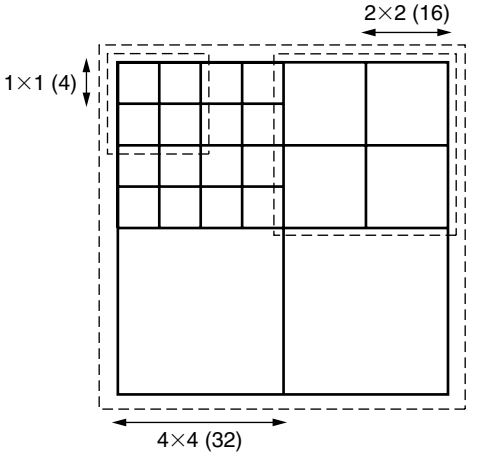

FIGURE 1.9 ■ Hierarchical routing used by long wires to connect clusters of logic blocks.

Within this block, local, nearest-neighbor routing is all that is available. In turn, a 2×2 cluster of these clusters is formed that encompasses 16 logic blocks. At this level of hierarchy, longer wires at the boundary of the smaller, 2×2 clusters, connect each cluster of four logic blocks to the other clusters in the higher-level grouping. This is repeated in higher levels of hierarchy, with larger clusters and longer wires.

The pattern of interconnect just described exploits the assumption that a well-designed (and well-placed) circuit has mostly local connections and only a limited number of connections that need to travel long distances. By providing fewer resources at the higher levels of hierarchy, this interconnect architecture remains area-efficient while preserving some long-length wires to minimize the delay of signals that need to cross large distances.

As in the segmented architecture, the connection points that connect one level of routing hierarchy to another can be anywhere in the interconnect structure. New points in the existing switch blocks may be created, or completely independent

switching sites elsewhere in the interconnect can be created specifically for the purpose of moving between hierarchy levels.

1.2.2 Programmability

As with the logic blocks in a typical commercial FPGA, each switch point in the interconnect structure is programmable. Within the connection block, programmable multiplexers select which routing track each logic block's input and output terminals map to; in the switch block, the junction between vertical and horizontal routing tracks is switched through a programmable switch; and, finally, switching between routing tracks of different segment lengths or hierarchy levels is accomplished, again through programmable switches.

For all of these programmable points, as in the logic block, modern FPGAs use SRAM bits to hold the user-defined configuration values. More discussion of these configuration bits comes later in this chapter.

1.2.3 Summary

Programmable routing resources are the natural counterpart to the logic resources in an FPGA. Where the logic performs the arithmetic and logical computations, the interconnection fabric takes the results output from logic blocks and routes them as inputs to other logic blocks. By tiling logic blocks together and connecting them through a series of programmable interconnects as described here, an FPGA can implement complex digital circuits. The true nature of *spatial computing* is realized by spreading the computation across the physical area of an FPGA.

Today's commercial FPGAs typically use bits of each of these interconnect architectures to provide a smooth and flexible set of routing resources. In actual implementation, segmentation and hierarchy may not always exhibit the logarithmic scaling seen in our examples. In modern FPGAs, the silicon area consumed by interconnect greatly dominates the area dedicated to logic. Anecdotally, 90 percent of the available silicon is interconnect whereas only 10 percent is logic. With this imbalance, it is clear that interconnect architecture is increasingly important, especially from a delay perspective.

1.3 EXTENDING LOGIC

With a logic block like the one shown in Figure 1.2, tiled in a two-dimensional array with a supporting interconnect structure, we can implement any combinational and sequential logic. Our only constraint is area in terms of the number of available logic blocks. While this is comprehensive, it is far from optimal. In this section, we investigate how FPGA architects have augmented this simple design to increase performance.

1.3.1 Extended Logic Elements

Modern FPGA interconnect architectures have matured to include much more than simple nearest-neighbor connectivity to give increased performance for

common applications. Likewise, the basic logic elements have been augmented to increase performance for common operations such as arithmetic functions and data storage.

Fast carry chain

One fundamental operation that the FPGA is likely to perform is an addition. From the basic logic block, it is apparent that we can implement a full-adder structure with two logic blocks given at least a 3-LUT. One logic block is configured to compute the sum, and one is configured to compute the carry. Cascading *N* pairs of logic blocks together will yield a simple *N*-bit full adder.

As you may already know from digital arithmetic, the critical path of this type of addition comes not from the computation of the sum bits but rather from the rippling of the carry signal from lower-order bits to higher-order bits (see Figure 1.10). This path starts with the low-order primary inputs, goes through the logic block, out into the interconnect, into the adjacent logic block, and so on. Delay is accumulated at every switch point along the way.

One clever way to increase speed is to shortcut the carry chain between adjacent logic blocks. We can accomplish this by providing a dedicated, minimally switched path from the output of the logic block computing the carry signal to the adjacent higher-order logic block pair. This carry chain will not need to be routed on the general interconnect network. By adding a minimal amount of overhead (wires), we dramatically speed up the addition operation.

This feature does force some constraints on the spatial layout of a multibit addition. If, for instance, the dedicated fast carry chain only goes vertically, along columns of logic blocks, all additions must be oriented along the carry chain to take advantage of this dedicated resource. Additionally, to save switching area, the dedicated carry chain may not be a bidirectional path, which further restricts the physical layout to be oriented vertically and dictates the order of the bits relative to one another. The fast carry-chain of the Xilinx XC4000E is shown in Figure 1.11. Note that the bidirectional fast carry-chain wires are arranged along the columns while the horizontal lines are unidirectional. This allows large adder structures to be placed in a zig-zag pattern in the array and still make use of the dedicated carry-chain interconnect.

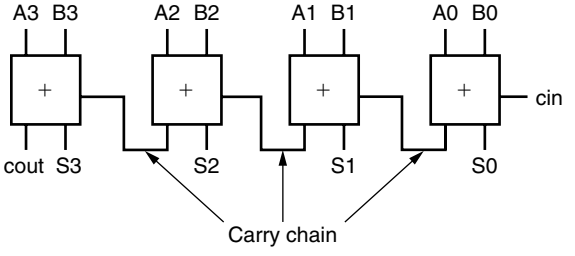

FIGURE 1.10 ■ A simple 4-bit full adder.

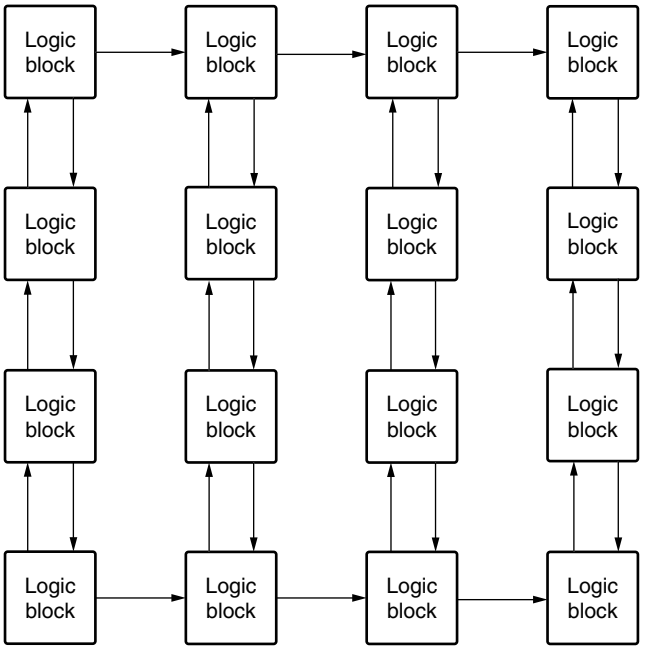

FIGURE 1.11 ■ The Xilinx XC4000E fast carry chain. (*Source:* Adapted from [6], Figure 11, p. 6-18.)

The fast carry-chain logic is now commonplace in commercial FPGAs, with the physical design constraints at this point completely abstracted away by the tools provided by manufacturers. The success of this optimization relies on the toolset's ability to identify additions in the designer's circuit description and then use the dedicated logic. With today's tools, this kind of optimization is nearly transparent to the end user.

Multipliers

If addition is commonplace in algorithms, multiplication is certainly not rare. Several implementations are available if we wish to use general logic block resources to build our multipliers. From the area-efficient iterative shift-accumulate method to the area-consumptive array multiplier, we can use logic blocks to either compute additions or store intermediate values. While we can certainly implement a multiplication, we can do so only with a large delay penalty, or a large logic block footprint, depending on our implementation. In essence, our logic blocks *aren't very efficient* at performing a multiplication.

Instead of doing it with logic blocks, why not build *real* multipliers outside, but still connected to, the general FPGA fabric? Then, instead of inefficiently using simple LUTs to implement a multiply, we can route the values that need to be multiplied to actual multipliers implemented in silicon. How does this save space and time? Recall that FPGAs trade speed and power for configurability when compared to their ASIC (application-specific integrated circuit) counterparts. If you asked a VLSI designer to implement a fast multiplier out of transistors

any way she wanted, it would take up far less silicon area, be much faster, and consume less power than we could ever manage using LUTs.

The result is that, for a small price in silicon area, we can offload the otherwise area-prohibitive multiplication onto dedicated hardware that does it much better. Of course, just like fast carry chains, multipliers impose important design considerations and physical constraints, but we add one more option for computation to our palette of operations. It is now just a matter of good design and good tools to make an efficient design. Like fast carry chains, multipliers are commonplace in modern FPGAs.

RAM

Another area that has seen some customization beyond the general FPGA fabric is in the area of on-chip data storage. While logic blocks can individually provide a few bits of storage via the lookup table structure—and, in aggregate, many bits—they are far from an efficient use of FPGA resources. Like the fast carry chain and the "hard" multiplier, FPGA architectures have given their users generous amounts of on-chip RAM that can be accessed from the general FPGA fabric.

Static RAM cells are extremely small and, when physically distributed throughout the FPGA, can be very useful for many algorithms. By grouping many static RAM cells into banks of memory, designers can implement large ROMs for extremely fast lookup table computations and constant-coefficient operations, and large RAMs for buffering, queuing, and basic scratch use—all with the convenience of a simple clocking strategy and the speed gained by avoiding off-chip communication to an external memory. Today's FPGAs provide anywhere from kilobits to megabits of dedicated RAM.

Processor blocks

Tying all these blocks together, most commercial FPGAs now offer entire dedicated processors in the FPGA, sometimes even more than one. In a general sense, FPGAs are extremely efficient at implementing raw computational pipelines, exploiting nonstandard bit widths, and providing data and functional parallelism. The inclusion of dedicated CPUs recognizes the fact that algorithm flows that are very procedural and contain a high degree of branching do not lend themselves readily to acceleration using FPGAs.

Entire CPU blocks can now be found in high-end FPGA devices. At the time of this writing, these CPUs are on the scale of 300 MHz PowerPC devices, complete, without floating-point units. They are capable of running an entire embedded operating system, and some are even able to reprogram the FPGA fabric around them.

The CPU cores are not nearly as easily exploited as the carry chains, multipliers, and on-chip RAMs, but they represent a distinct shift toward making FPGAs more "platform"-oriented. With a traditional CPU on board (and perhaps up to four), a single FPGA can serve nearly as an entire "system-on-a-chip"—the holy grail of system integrators and embedded device manufacturers. With standard programming languages and toolchains available to developers, an entire project might indeed be implemented with a single-chip solution, dramatically reducing cost and time to market.

1.3.2 Summary

In the end, modern commercially available FPGAs provide a rich variety of basic, and not so basic, computational building blocks. With much more than simple lookup tables, the task for the FPGA architect is to decide in what proportion to provide these resources and how they should be connected. The task of the hardware designer is then to fully understand the capabilities of the target FPGAs to create designs that exploit their potential.

The common thread among these extended logical elements is that they provide critical functionality that cannot be implemented very efficiently in the general FPGA fabric. As much as the technology drives FPGA architectures, applications provide a much needed push. If multiplies were rare, it wouldn't make sense to waste silicon space on a "hard" multiplier. As FPGAs become more heterogeneous in nature, and become useful computational platforms in new application domains, we can expect to see even more varied blocks in the next generation of devices.

1.4 CONFIGURATION

One of the defining features of an FPGA is its ability to act as "blank hardware" for the end user. Providing more performance than pure software implementations on general-purpose processors, and more flexibility than a fixed-function ASIC solution, relies on the FPGA being a reconfigurable device. In this section, we will discuss the different approaches and technologies used to provide programmability in an FPGA.

Each configurable element in an FPGA requires 1 bit of storage to maintain a user-defined configuration. For a simple LUT-based FPGA, these programmable locations generally include the contents of the logic block and the connectivity of the routing fabric. Configuration of the FPGA is accomplished through programming the storage bits connected to these programmable locations according to user definitions. For the lookup tables, this translates into filling it with 1s and 0s. For the routing fabric, programming enables and disables switches along wiring paths.

The configuration can be thought of as a flat binary file whose contents map, bit for bit, to the programmable bits in the FPGA. This *bitstream* is generated by the vendor-specific tools after a hardware design is finalized. While its exact format is generally not publicly known, the larger the FPGA, the larger the bitstream becomes.

Of course, there are many known methods for storing a single bit of binary information. We discuss the most popular methods used for FPGAs next.

1.4.1 SRAM

As discussed in previous sections, the most widely used method for storing configuration information in commercially available FPGAs is volatile static RAM, or SRAM. This method has been made popular because it provides fast and infinite reconfiguration in a well-known technology.

Drawbacks to SRAM come in the form of power consumption and data volatility. Compared to the other technologies described in this section, the SRAM cell is large (6–12 transistors) and dissipates significant static power because of leakage current. Another significant drawback is that SRAM does not maintain its contents without power, which means that at power-up the FPGA is not configured and must be programmed using off-chip logic and storage. This can be accomplished with a nonvolatile memory store to hold the configuration and a micro-controller to perform the programming procedure. While this may seem to be a trivial task, it adds to the component count and complexity of a design and prevents the SRAM-based FPGA from being a truly single-chip solution.

1.4.2 Flash Memory

Although less popular than SRAM, several families of devices use Flash memory to hold configuration information. Flash memory is different from SRAM in that it is nonvolatile and can only be written a finite number of times.

The nonvolatility of Flash memory means that the data written to it remains when power is removed. In contrast with SRAM-based FPGAs, the FPGA remains configured with user-defined logic even through power cycles and does not require extra storage or hardware to program at boot-up. In essence, a Flash-based FPGA can be ready immediately.

A Flash memory cell can also be made with fewer transistors compared to an SRAM cell. This design can yield lower static power consumption as there are fewer transistors to contribute to leakage current.

Drawbacks to using Flash memory to store FPGA configuration information stem from the techniques necessary to write to it. As mentioned, Flash memory has a limited write cycle lifetime and often has slower write speeds than SRAM. The number of write cycles varies by technology, but is typically hundreds of thousands to millions. Additionally, most Flash write techniques require higher voltages compared to normal circuits; they require additional off-chip circuitry or structures such as charge pumps on-chip to be able to perform a Flash write.

1.4.3 Antifuse

A third approach to achieving programmability is antifuse technology. Antifuse, as its name suggests, is a metal-based link that behaves the opposite of a fuse. The antifuse link is normally open (i.e., unconnected). A programming procedure that involves either a high-current programmer or a laser melts the link to form an electrical connection across it—in essence, creating a wire or a short-circuit between the antifuse endpoints.

Antifuse has several advantages and one clear disadvantage, which is that it is not reprogrammable. Once a link is fused, it has undergone a physical transformation that cannot be reversed. FPGAs based on this technology are generally considered one-time programmable (OTP). This severely limits their flexibility in terms of reconfigurable computing and nearly eliminates this technology for use in prototyping environments.

However, there are some distinct advantages to using antifuse in an FPGA platform. First, the antifuse link can be made very small, compared to the large multi-transistor SRAM cell, and does not require any transistors. This results in very low propagation delays across links and zero static power consumption, as there is no longer any transistor leakage current. Antifuse links are also not susceptible to high-energy radiation particles that induce errors known as single-event upsets, making them more likely candidates for space and military applications.

1.4.4 Summary

There are several well-known methods for storing user-defined configuration data in an FPGA. We have reviewed the three most common in this section. Each has its strengths and weaknesses, and all can be found in current commercial FPGA products.

Regardless of the technology used to store or convey configuration data, the idea remains the same. From vendor-specific tools, a device-specific programming bitstream is created and used either to program an SRAM or Flash memory, or to describe the pattern of antifuse links to be used. In the end, the user-defined configuration is reflected in the FPGA, bringing to reality part of the vision of reconfigurable computing.

1.5 CASE STUDIES

If you've read everything thus far, the FPGA should no longer seem like a magical computational black box. In fact, you should have a good grasp of the components that make up modern commercial FPGAs and how they are put together. In this section, we'll take it one step further and solidify the abstractions by taking a look at two real commercial architectures—the Altera Stratix and the Xilinx Virtex-II Pro—and linking the ideas introduced earlier in this chapter with concrete industry implementations.

Although these devices represent near-current technologies, having been introduced in 2002, they are not the latest generation of devices from their respective manufacturers. The reason for choosing them over more cutting-edge examples is in part due to the level of documentation available at the time of this writing. As is often the case, detailed architecture information is not available as soon as a product is released and may never be available depending on the manufacturer.

Finally, the devices discussed here are much more complex than we have space to describe. The myriad ways modern devices can be used to perform computation and the countless hardware and software features that allow you to create powerful and efficient designs are all part of a larger, more advanced dialog. So if something seems particularly interesting, we encourage you to grab a copy of the device handbook(s) and dig a little deeper.

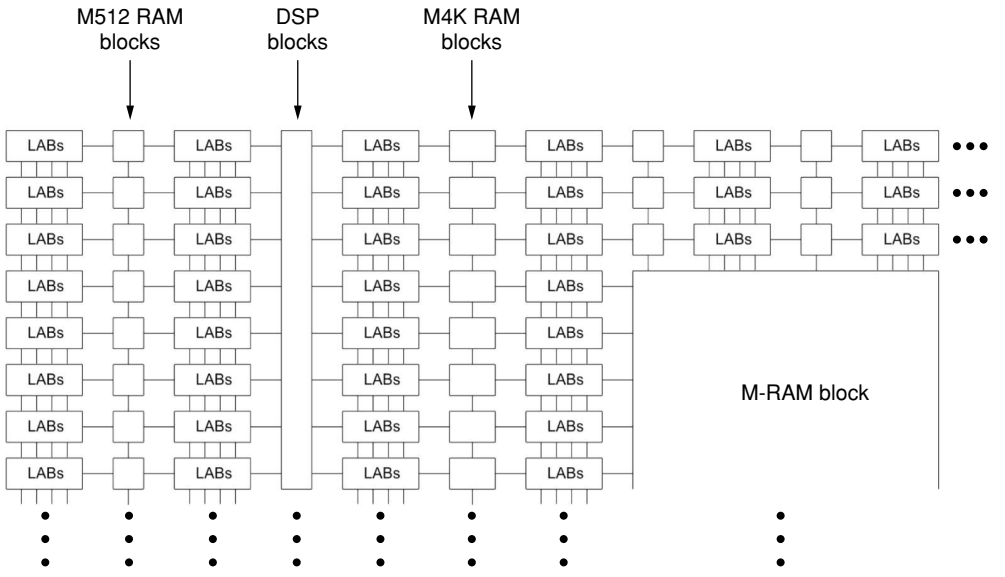

FIGURE 1.12 ■ Altera Stratix block diagram. (*Source:* Adapted from [7], Chapter 2, p. 2-2.)

1.5.1 Altera Stratix

We begin by taking a look at the Altera Stratix FPGA. Much of the information presented here is adapted from the July 2005 edition of the *Altera Stratix Device Handbook* (available online at *http://www.altera.com*).

The Stratix is an SRAM-based island-style FPGA containing many heterogeneous computational elements. The basic logical tile is the logic array block (LAB), which consists of 10 logic elements (LEs). The LABs are tiled across the device in rows and columns with a multilevel interconnect bringing together logic, memory, and other resources. Memory is provided through TriMatrix memory structures, which consist of three memory block sizes—M512, M4K, and M-RAM—each with its own unique properties. Additional computational resources are provided in DSP blocks, which can efficiently perform multiplication and accumulation. These resources are shown in a high-level block diagram in Figure 1.12.

Logic architecture

The smallest logical block in the array is the LE, shown in Figure 1.13. The general architecture of the LE is very similar to the structure that we introduced earlier—a single 4-LUT function generator and a programmable register as a state-holding element. In the Altera LE, you can see additional components to facilitate driving the interconnect (*right* side of Figure 1.12), setting and clearing the programmable register, choosing from several programmable clocks, and propagating the carry chain.

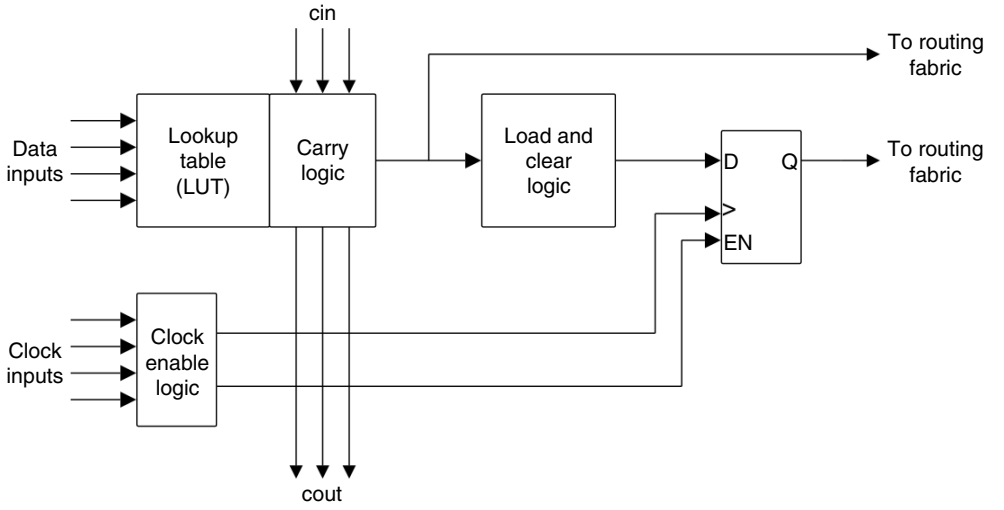

FIGURE 1.13 ■ Simplified Altera Stratix logic element. (*Source:* Adapted from [7], Chapter 2, p. 2-5.)

Because the LEs are simple structures that may appear tens of thousands of times in a single device, Altera groups them into LABs. The LAB is then the basic structure that is tiled into an array and connected via the routing structure. Each LAB consists of 10 LEs, all LE carry chains, LAB-wide control signals, and several local interconnection lines. In the largest device, the EP1S80, there are 101 LAB rows and 91 LAB columns, yielding a total of 79,040 LEs. This is fewer than would be expected given the number of rows and columns because of the presence of the TriMatrix memory structures and DSP blocks embedded in the array.

As shown in Figure 1.14, the LAB structure is dominated, at least conceptually, by interconnect. The local interconnect allows LEs in the same LAB to send signals to one another without using the general interconnect. Neighboring LABs, RAM blocks, and DSP blocks can also drive the local interconnect through direct links. Finally, the general interconnect (both horizontal and vertical channels) can drive the local interconnect. This high degree of connectivity is the lowest level of a rich, multilevel routing fabric.

The Stratix has three types of memory blocks—M512, M4K, and M-RAM—collectively dubbed TriMatrix memory. The largest distinction between these blocks is their size and number in a given device. Generally speaking, they can be configured in a number of ways, including single-port RAM, dual-port RAM, shift-register, FIFO, and ROM table. These memories can optionally include parity bits and have registered inputs and outputs.

The M512 RAM block is nominally organized as a 32×18-bit memory; the M4K RAM as a 128×36-bit memory; and the M-RAM as a $4\text{K} \times 144$-bit memory. Additionally, each block can be configured for a variety of widths depending on the needs of the user. The different-sized memories throughout the array provide

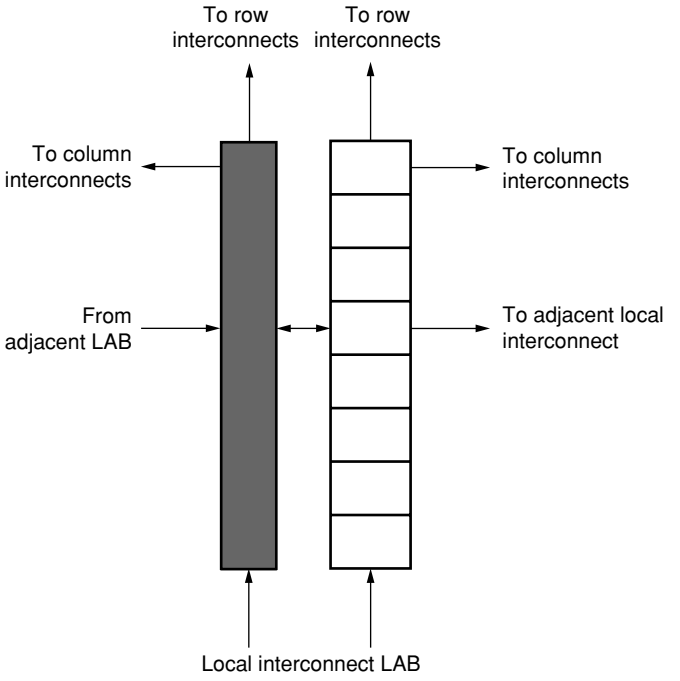

FIGURE 1.14 ■ Simplified Altera Stratix LAB structure. (*Source:* Adapted from [8], Chapter 2, p. 2-4.)

an efficient mapping of variable-sized memory designs to the device. In total, on the EP1S80 there are over 7 million memory bits available for use, divided into 767 M512 blocks, 364 M4K blocks, and 9 M-RAM blocks.

The final element of logic present in the Altera Stratix is the DSP block. Each device has two columns of DSP blocks that are designed to help implement DSP-type functions, such as finite-impulse response (FIR) and infinite-impulse response (IIR) filters and fast Fourier transforms (FFT), without using the general logic resources of the LEs. The common computational function required in these operations is often a multiplication and an accumulation. Each DSP block can be configured by the user to support a single 36×36-bit multiplication, four 18×18-bit multiplications, or eight 9×9-bit multiplications, in addition to an optional accumulation phase. In the EP1S80, there are 22 total DSP blocks.

Routing architecture

The Altera Stratix provides an interconnect system dubbed MultiTrack that connects all the elements just discussed using routing lines of varying fixed lengths. Along the row (horizontal) dimension, the routing resources include direct connections left and right between blocks (LABs, RAMs, and DSP) and interconnects of lengths 4, 8, and 24 that traverse either 4, 8, or 24 blocks left and right, respectively. A detailed depiction of an R4 interconnect at a single

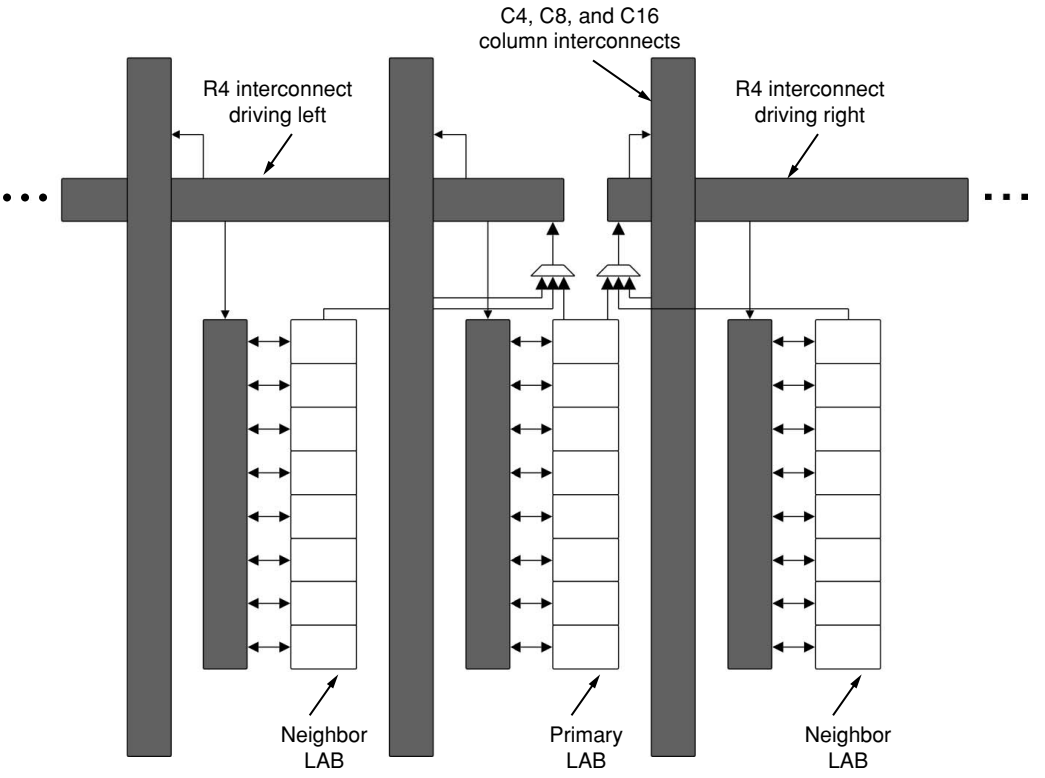

FIGURE 1.15 ■ Simplified Altera Stratix MultiTrack interconnect. (*Source:* Adapted from [7], Chapter 2, p. 2-14.)

LAB is shown in Figure 1.15. The R4 interconnect shown spans 4 blocks, left to right. The relative sizing of blocks in the Stratix allows the R4 interconnect to span four LABs; three LABs and one M512 RAM; two LABs and one M4K RAM; or two LABs and one DSP block, in either direction.

This structure is repeated for every LAB in the row (i.e., every LAB has its own set of dedicated R4 interconnects driving left and right). R4 interconnects can drive C4 and C16 interconnects to propagate signals vertically to different rows. They can also drive R24 interconnects to efficiently travel long distances.

The R8 interconnects are identical to the R4 interconnects except that they span 8 blocks instead of 4 and only connect to R8 and C8 interconnects. By design, the R8 interconnect is faster than two R4 interconnects joined together. The R24 interconnect provides the fastest long-distance interconnection. It is similar to the R4 and R8 interconnects, but does not connect directly to the LAB local interconnects. Instead, it is connected to row and column interconnects at every fourth LAB and only communicates to LAB local interconnects through R4 and C4 routes. R24 interconnections connect with all interconnection routes except L8s.

In the column (vertical) dimension, the resources are very similar. They include LUT chain and register chain direct connections and interconnects of lengths 4, 8, and 16 that traverse 4, 8, or 16 blocks up and down, respectively. The LAB local interconnects found in row routing resources are mirrored through LUT chain and register chain interconnects. The LUT chain connects the combinatorial output of one LE to the fast input of the LE directly below it without consuming general routing resources. The register chain connects the register output of one LE to the register input of another LE to implement fast shift registers.

Finally, although this discussion was LAB-centric, all blocks connect to the MultiTrack row and column interconnect using a direct connection similar to the LAB local connection interfaces. These direct connection blocks also support fast direct communication to neighboring LABs.

1.5.2 Xilinx Virtex-II Pro

Launched and shipped right behind the Altera Stratix, the Xilinx Virtex-II Pro FPGA was the flagship product of Xilinx, Inc. for much of 2002 and 2003. A good deal of the information that is presented here is adapted from "Module 2 (Functional Description)" of the October 2005 edition of *Xilinx Virtex-II Pro™ and Virtex-II Pro X™ Platform FPGA Handbook* (available at *http://www.xilinx.com*).

The Virtex-II Pro is an SRAM-based island-style FPGA with several heterogeneous computational elements interconnected through a complex routing matrix. The basic logic tile is the configurable logic block (CLB), consisting of four *slices* and two 3-state buffers. These CLBs are tiled across the device in rows and columns with a segmented, hierarchical interconnect tying all the resources together. Dedicated memory blocks, SelectRAM+, are spread throughout the device. Additional computational resources are provided in dedicated 18×18-bit multiplier blocks.

Logic architecture

The smallest piece of logic from the perspective of the interconnect structure is the CLB. Shown in Figure 1.16, it consists of four equivalent *slices* organized into two columns of two slices each with independent carry chains and a common shift chain. Each slice connects to the general routing fabric through a configurable switch matrix and to each other in the CLB through a fast local interconnect.

Each slice comprises primarily two 4-LUT function generators, two programmable registers for state holding, and fast carry logic. The slice also contains extra multiplexers (MUXFx and MUXF5) to allow a single slice to be configured for wide logic functions of up to eight inputs. A handful of other gates provide extra functionality in the slice, including an XOR gate to complete a 2-bit full adder in a single slice, an AND gate to improve multiplier implementations in the logic fabric, and an OR gate to facilitate implementation of sum-of-products chains.

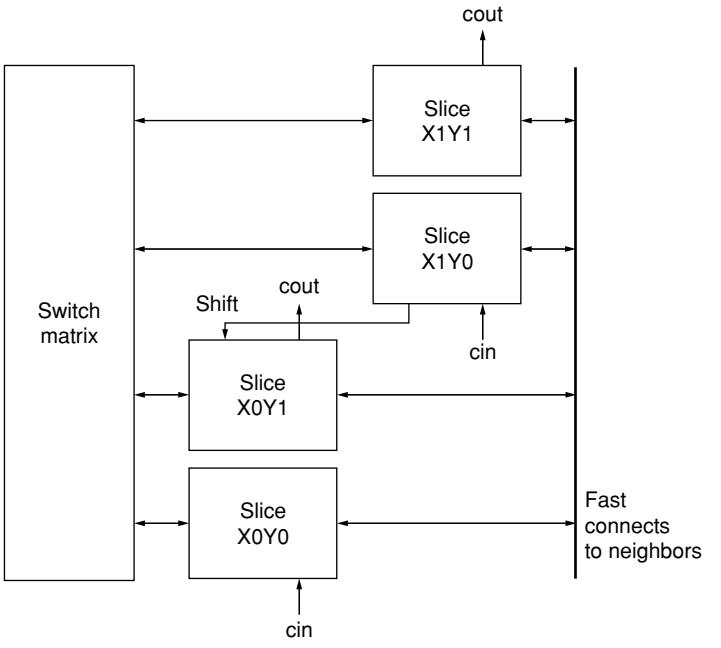

FIGURE 1.16 ■ Xilinx Virtex-II Pro configurable CLB. (*Source:* Adapted from [8], Figure 32, p. 35.)

In the largest Virtex-II Pro device, the XC2VP100, there are 120 rows and 94 columns of CLBs. This translates into 44,096 individual slices and 88,192 4-LUTs—comparable to the largest Stratix device. In addition to these general configurable logic resources, the Virtex-II Pro provides dedicated RAM in the form of block SelectRAM+. Organized into multiple columns throughout the device, each block SelectRAM+ provides 18 Kb of independently clocked, true dual-port synchronous RAM. It supports a variety of configurations, including single- and dual-port access in various aspect ratios. In the largest device there are 444 blocks of block SelectRAM+ organized into 16 columns, yielding a total of 8,183,808 bits of memory.

Complementing the general logic resources are a number of 18×18-bit 2's complement signed multiplier blocks. Like the DSP blocks in the Altera Stratix, these multiplier structures are designed for DSP-type operations, including FIR, IIR, FFT, and others, which often require multiply-accumulate structures. As shown in Figure 1.17, each 18×18 multiplier block is closely associated with an 18Kb block SelectRAM+. The use of the multiplier/block SelectRAM+ memory, with an accumulator implemented in LUTs, allows the implementation of efficient multiply-accumulate structures. Again, in the largest device, just as with block SelectRAM+, there are 16 columns yielding a total of 444 18×18-bit multiplier blocks.

Finally, the Virtex-II Pro has one unique feature that has been carried into newer products and can also be found in competing Altera products. Embedded

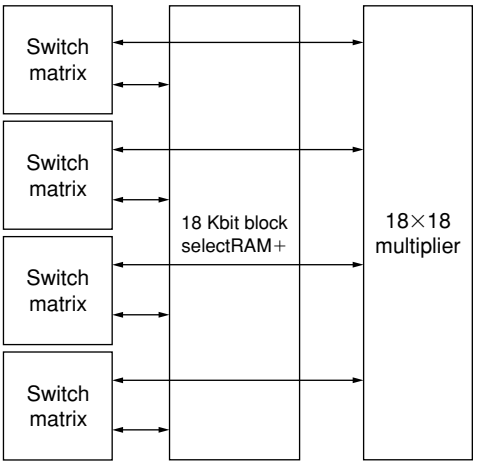

FIGURE 1.17 ■ Virtex-II Pro multiplier/block SelectRAM+ organization. (*Source:* Adapted from [8], Figure 53, p. 48.)

in the silicon of the FPGA, much like the multiplier and block SelectRAM+ structures, are up to four IBM PowerPC 405-D5 CPU cores. These cores can operate up to 300+ MHz and communicate with surrounding CLB fabric, block SelectRAM+, and general interconnect through dedicated interface logic. On-chip memory (OCM) controllers allow the PowerPC core to use block SelectRAM+ as small instruction and data memories if no off-chip memories are available.

The presence of a complete, standard microprocessor that has the ability to interface at a very low level with general FPGA resources allows unique, system-on-a-chip designs to be implemented with only a single FPGA device. For example, the CPU core can execute housekeeping tasks that are neither time-critical nor well suited to implementation in LUTs.

Routing architecture

The Xilinx Virtex-II Pro provides a segmented, hierarchical routing structure that connects to the heterogeneous fabric of elements through a switch matrix block. The routing resources (dubbed Active Interconnect) are physically located in horizontal and vertical routing channels between each switch matrix and look quite different from the Altera Stratix interconnect structures.

The routing resources available between any two adjacent switch matrix rows or columns are shown in Figure 1.18, with the switch matrix block shown in black. These resources include, from top to bottom, the following:

- 24 long lines that span the full height and width of the device.
- 120 hex lines that route to every third or sixth block away in all four directions.

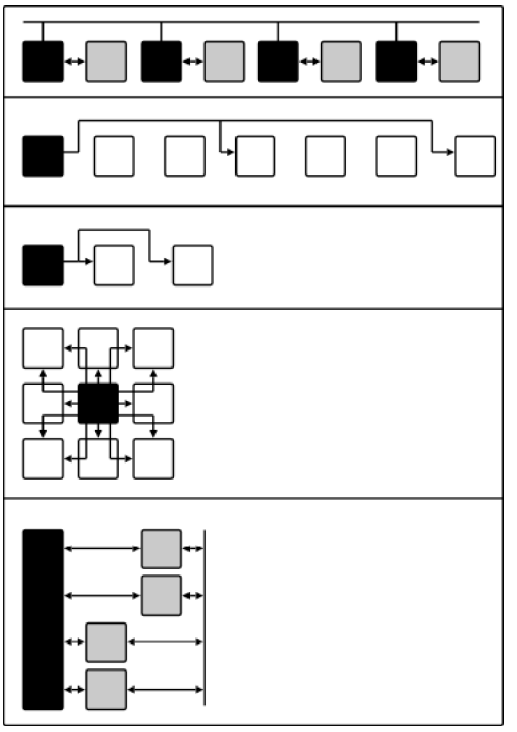

FIGURE 1.18 ■ Xilinx Virtex-II Pro routing resources. (*Source:* Adapted from [7], Figure 54, p. 45.)

- 40 double lines that route to every first or second block away in all four directions.
- 16 direct connect routes that route to all immediate neighbors.
- 8 fast-connect lines in each CLB that connect LUT inputs and outputs.

1.6 SUMMARY

This chapter presented the basic inner workings of FPGAs. We introduced the basic idea of lookup table computation, explained the need for dedicated computational blocks, and described common interconnection strategies. We learned how these devices maintain generality and programmability while providing performance through dedicated hardware blocks. We investigated a number of ways to program and maintain user-defined configuration information. Finally, we tied it all together with brief overviews of two popular commercial architectures, the Altera Stratix and the Xilinx Virtex-II Pro.

Now that we have introduced the basic technology that serves as the foundation of reconfigurable computing, we will begin to build on the FPGA to create

reconfigurable devices and systems. The following chapters will discuss how to efficiently conceptualize computations spatially rather than procedurally, and the algorithms necessary to go from a user-specified design to configuration data. Finally, we'll look into some application domains that have successfully exploited the power of reconfigurable computing.

References

[1] J. Rose, A. E. Gamal, A Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE* 81(7), July 1993.

[2] P. Chow, et al. The design of an SRAM-based field-programmable gate array—Part 1: Architecture. *IEEE Transactions on VLSI Systems* 7(2), June 1999.

[3] H. Fan, J. Liu, Y. L. Wu, C. C. Cheung. On optimum switch box designs for 2-D FPGAs. *Proceedings of the 38th ACM/SIGDA Design Automation Conference (DAC)*, June 2001.

[4] ———-. On optimal hyperuniversal and rearrangeable switch box designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(12), December 2003.

[5] H. Schmidt, V. Chandra. FPGA switch block layout and evaluation. *IEEE International Symposium on Field-Programmable Gate Arrays*, February 2002.

[6] Xilinx, Inc. *Xilinx XC4000E and XC4000X Series Field-Programmable Gate Arrays, Product Specification* (Version 1.6), May 1999.

[7] Altera Corp. *Altera Stratix™ Device Handbook*, July 2005.

[8] Xilinx, Inc. *Xilinx Virtex-II Pro™ and Virtex-II Pro™ Platform FPGA Handbook*, October 2005.