Making Everything Easier!™

FPGAs

FOR DUMMIES®

A Wiley Brand

Learn:

- How FPGAs work

- The differences between FPGAs, ASSPs, and ASICs

- To use FPGAs as functional blocks in a system

Brought to you by

(intel®)

Andrew Moore
with Ron Wilson

FPGAs

FOR

DUMMIES

A Wiley Brand

2nd Intel® Special Edition

by Andrew Moore

with Ron Wilson, Editor-in-Chief, Intel® Programmable Solutions Group

FOR

DUMMIES

A Wiley Brand

FPGAs For Dummies,® 2nd Intel® Special Edition

Publisher's Acknowledgments

Table of Contents

Introduction

Field programmable gate arrays (FPGAs) are integrated circuits that enable designers to program customized digital logic in the field. FPGAs have been around since the 1980s and were originally conceived to give all design teams the ability to create custom logic. In the early days, using an FPGA in your design meant you had to do a lot of programming just to get your FPGA to perform simple functions, so most designers avoided them. If you haven't looked into FPGAs since your university studies way back when, you'll want to take another look at them.

The FPGA has evolved from a useful but humble interface device into a system-level integrated circuit (IC) with its own microprocessors, memory blocks, and interfaces. It's the next big thing.

Now would be a great time to get an inexpensive development kit, download free tools, and begin to explore this world for yourself. And this book will help you understand the practical uses of FPGAs.

About This Book

This book is for you if you're a system designer, an experienced engineer, or someone who hasn't seen FPGAs since your days in engineering school.

This book was written with cooperation from Altera Corporation, now a part of Intel Corporation.

Icons Used in This Book

Throughout this book, I occasionally use special icons to call attention to important information. You won't see the typical smiley faces or emoticons, but you'll definitely want to stop and pay attention! Here's what you can expect.

This icon points out information that you'll want to put into your cache, your memory, or whatever that thing is you use to keep information, like birthdays and phone numbers, for later use!

Who knows? You might come away with some neat information you can use to impress your friends at parties!

No, I'm not asking you to take care of your servers and bartenders! You should stop and take notice because these are bits and pieces of knowledge that could save you aggravation later on.

Beyond the Book

Although this book is stuffed with information, I can only cover so much in 48 pages! So, if you find yourself wanting more information about FPGAs, just go to `www.intel.com/altera`. There you can find more information about Intel FPGAs. You can also view videos and webinars, download demos, read data sheets and white papers, and much more!

Chapter 1

FPGAs for Everyone

*W*elcome! If you're reading this chapter, it's a good bet that you're an engineer who may have learned about field programmable gate arrays (FPGAs) in the past, but you now want to know more. Maybe you're wondering how you can use them with your designs. In this chapter, I introduce you to FPGAs and explain what problems they solve. I also go over how they do what they do and discuss design tradeoffs and the truth about FPGA operation.

Compared to other ways of building hardware, FPGAs have two huge things going for them. First, they enable you to build exactly the hardware you need, instead of having to use the same application-specific standard product (ASSP) all your competitors are using, or having to undertake the time, cost, and risk of an application-specific integrated circuit (ASIC) design.

But just as important, that ability to customize the FPGA means that often, in an FPGA, you can do operations in a simpler, faster, more energy-efficient way than they could be done in the microprocessor cores of an ASSP.

So Why Would You Need an FPGA, Anyway?

I'm glad you asked! An *FPGA* is a semiconductor device on which the function can be defined after manufacturing. An FPGA enables you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field — hence the term *field programmable*. And *gate arrays* are two-dimensional arrays of logic gates. If you get enough of these things put together, you can make those simple calculations add up to do something meaningful.

In less technical terms, an FPGA allows you flexibility in your designs and is a way to change how parts of a system work without introducing a large amount of cost and risk of delays into the design schedule.

A simple example is a rear-view camera designed for a car. If your camera system takes 250 milliseconds from the time the image sensor sees the image until the image frame actually appears on the display, and a change in government regulations requires that this delay or latency be no more than 100 milliseconds, you could find ways to adjust the image signal processing pipeline in an FPGA to comply with the new latency requirements. This would be almost impossible to do with a microprocessor-based system. In this example, a company can gain a big advantage using an FPGA because it doesn't have to redesign parts or buy all new processors.

In the early days, FPGA circuits were very large and you couldn't fit many onto a single chip. All a designer could do was build an interface with an FPGA and customers could reprogram that interface to do something different (for example, changing an interface that operates on a keyboard input to one that handles input from a touchscreen device). Soon, however, designers realized they could build entire subsystems out of FPGAs, which meant designers were no longer restricted to only using ASICs to implement these subsystems. Because circuit components keep getting smaller and smaller, designers can put many more devices on the same

chip — allowing for more sophisticated functionality and faster arithmetic, which in turn leads to faster computation and less power consumption.

Modern FPGAs consist of mixes of configurable static random access memory (SRAM), high-speed input/output pins (I/Os), logic blocks, and routing. More specifically, an FPGA contains programmable logic elements called, naturally, logic elements (LEs), as well as a hierarchy of reconfigurable interconnects that allow the LEs to be physically connected to one another. You can configure LEs to do complex functions or simply perform basic logic gates, such as AND and OR. Most FPGAs also contain memory blocks (for more on these topics, see the section "The building blocks of an FPGA").

ASICs and ASSPs

An *application-specific integrated circuit (ASIC)* is an integrated circuit composed of electrical components, such as transistors, capacitors, and resistors, fabricated on a wafer composed of silicon or other semiconductor material that is customized for a particular use. Two examples of ASICs are a voice recorder and a high-efficiency Bitcoin miner. Over the years, the size of components used in ICs has shrunk, meaning that more complex circuits can be created using the same space. Because of this shrinking of components, some ASICs have now become large enough to contain multiple microprocessors and other complex subsystems.

Application-specific standard products (ASSPs), on the other hand, are ICs that are dedicated to a specific application market and sold to more than one user (and hence, standard) in contrast to ASICs, which are designed for, and sold to, a single customer. Some examples of ASSPs are microcontrollers and the system chips at the hearts of many smartphones and tablets.

ASICs and ASSPs are specifically designed for dedicated functionality. Because of the tight control of their configuration, ASICs and ASSPs are very compact, inexpensive, fast, and low-power, which are all highly desirable traits in electronics design. Because their function is hard-wired at the time of manufacture, it isn't easy to change the functionality of even a small part of the circuit. In fact, because these circuits are permanently fabricated on silicon wafers, you simply can't take apart the circuitry and replace it with something else. If you need to change something in the design, you have to scrap the whole chip and start again.

Today, hard intellectual property (IP) can be built into the FPGA fabric to provide rich functionality while reducing power and lowering cost. Some examples of the hard IP included in today's FPGAs are memory blocks, calculating circuits, transceivers, protocol controllers, and even central processing units (CPUs). It is important to remember, however, that this hard IP isn't customizable like the rest of the FPGA! Integrating hard IP — for example, a digital signal processor (DSP) — frees designers from reinventing the wheel each time they need to add these common pieces to their systems. FPGA manufacturers can get away with weaving hard IP components into FPGAs because these functionalities have become commodities, and, therefore, quite uniform throughout most electronic systems.

Examining an FPGA

The following sections discuss FPGA components in detail and explain what an FPGA is in simpler language, in case it has been a long time since you first learned about FPGAs and have forgotten some of the details.

The building blocks of an FPGA

You can build anything digital from three simple pieces: a wire, a logic gate, and a register (see Figure 1-1). A *register* remembers a piece of information until it is told to remember something else. A *logic gate* performs simple logic operations on signals, and a *wire* connects these other pieces.

– A wire

– A gate

– A register
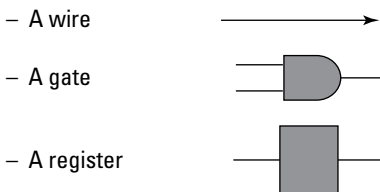
Courtesy of Intel Corporation.

Figure 1-1: The building blocks of a digital system.

Logic gates

Logic gates perform the core functionality of digital circuits, which means they perform simple logic on *inputs* — electrical

pulses that your computer uses to represent 0s and 1s (more on that later). On their own, these simple operations don't do much, but when you put thousands or even millions of these together you can do something really powerful. Your computer's CPU is made up of billions of logic gates that allow your computer to do all the cool stuff that it does.

In order to perform as functional circuits, logic gates use a type of arithmetic called *Boolean* algebra. Boolean algebra was first introduced in 1854 by George Boole. Unlike elementary algebra, where the values of variables are numbers and the main operations are addition and multiplication, the primary operations of Boolean algebra are the conjunction AND, the disjunction OR, and negation NOT. The values of variables in Boolean algebra are the *truth* values *true* and *false.*

The basics of Boolean algebra operations are as follows:

- ✔ **And** (conjunction) is denoted as x AND y, which yields a result of true if both x and y are true, and false otherwise.

- ✔ **Or** (disjunction) is denoted as x OR y, which yields true if either x or y is true, and false if neither x nor y is true.

- ✔ **Not** (negation) is denoted as NOT x, which yields true if x is false and false if x is true.

One of the most common uses of Boolean algebra happens to be in digital logic design. It turns out that Boolean algebra maps directly to digital circuits in which inputs are 0s and 1s or *false* and *true.* By connecting more and more of these logic gates that perform simple Boolean arithmetic on 0s and 1s, you get systems that can perform advanced functions. For instance, logic gates guide the Mars Rover, allow billions of devices to be connected to Global Positioning Systems (GPS), and even power your favorite game on your mobile device.

Registers

Registers are simple devices that store pieces of data for use in the future. Think of registers as a short-term spot for placing data that you can access quickly; this is where you would place a phone number given to you moments before dialing. As soon as you try to remember something else, like the time you have to be at an appointment, that telephone number you tried to remember earlier is replaced with the starting time of the appointment. Registers keep whatever information is given to them until they're told to forget it and keep new information.

Wire

The third piece of all things digital is the wire used to connect all the registers and logic gates. These elements must be connected for the entire system to do what you want, from simple tasks, such as adding 1 + 2, all the way to more complex tasks, such converting pulses from blue LEDs reading a Blu-ray disc into crisp, high-definition images on your television screen. You can build any digital system you want, with the right amount of logic gates and registers with wires to connect them all.

Explaining FPGAs creatively

But hold on a second! All this talk of logic gates, registers, and wires sounds really technical and is pretty abstract. Do you need to be an experienced electrical engineer to understand an FPGA? No! You can use two different metaphors to explain how FPGAs work. The first metaphor is beads and string. The second metaphor is the interlocking building block toys you used to play with as a kid.

It turns out that beads and interlocking building blocks illustrate two approaches to using logic elements to build electronic systems.

Beads and string

The method represented by beads and string gives the designer the finest control of the pattern by using small beads and thin string to connect them all. The result is a beautiful, very complex pattern. But this fine-grained control comes at a cost. It is very difficult, if not impossible, to change the pattern even slightly without undoing all your work and starting over. When you map the beads and threads to digital electronics design, you come up with a design that very much resembles ASICs or ASSPs.

Imagine a beadwork pattern created from many beads of different colors arranged in different patterns and connected by thread. Using these simple components, you can create almost any type of pattern from the most simple to the most complex depending on the number of beads, their colors, and how you choose to arrange them.

Now, imagine that the beads represent both registers and logic gates; and imagine the string as wire. Just like the beads and string, these elements produce a system — a system that

can perform all sorts of computations from the very simple all the way to the very complex. You can think of different bead colors as representing the different types of logic gates such as AND, OR, or NOT, and you see how these simple arithmetic operations can become very complex calculations much like the arrangement of simple colors of beads can become very intricate patterns once put together with string.

By arranging beads in patterns, you can create beautiful designs, but what happens when you want to change the patterns into something else by rearranging the beads or changing the colors of the beads? This is where things get complicated! In order to change the patterns, you have to untangle all the string in order to change things around. You'll soon find that the threads are so closely connected that you just can't undo part of the design. Very quickly, you'll see the whole design must come apart in order to change the pattern even slightly. Well, that's not very flexible, is it?

Building block toys

Building blocks are relatively big and chunky and can only fit together at certain points on the block. The building block design isn't quite as elegant and intricate as the pattern produced by beads and thread. You can, however, change parts of the design without ripping apart the whole thing and starting all over, which brings me to the second approach to digital electronic design: FPGAs.

Perhaps you've spent countless hours as a child or with your children using building blocks to build towers, firetrucks, and spaceships. Similarly, you can use building blocks to create a nice representation of digital systems by building a table and pretending that some bricks are logic gates, some are registers, and others are the wires used to connect them all.

Now imagine that someone tells you she wants to change the pattern on the lower-right side of the table and, perhaps, to change the colors of the blocks. Because the blocks are all interconnecting pieces, you can easily remove just those in the lower-right corner and replace them with a group of different pieces. The rest of the blocks on the table are left intact and you don't have to redo the entire design just to change a small part.

Comparing FPGAs and ASICs

FPGAs are generally more flexible and cost-effective than ASICs. In the following sections, I explain why.

Costs and flexibility

Using FPGAs, you can implement any logical function that an ASIC can do but with the distinct advantage of updating the functionality after chip manufacture, which is desirable for many applications. FPGAs are more cost-effective than ASICs because the customer can program FPGAs according to its requirements instead of contracting a vendor to design and construct an ASIC to meet its needs.

Design time risk reduction versus speed

If you set out to use the most advanced semiconductor process in the world, no matter the cost, you could always design an ASIC that would run faster than the fastest available FPGA. But almost no one uses the most advanced process: Doing so would be risky, very difficult, and witheringly expensive. In fact, only a handful of ASSP companies leap on a new process as soon as it's available. Everyone else uses a process that is one, two, or three generations old. And the fact is, the fastest FPGA you can get can compete directly with those older ASIC processes. And the FPGA brings reduced design work and far less risk.

If, for example, you're designing a system with specific power efficiency and performance requirements and are planning to use an older 65 nanometer (nm) ASIC, did you know you can achieve similar results with a current 20 nm FPGA?

And using the FPGA would shorten your design time, reduce your risk of design errors, and offer a lower total cost of ownership (TCO) than the ASIC. For most applications, the FPGA's power consumption will be acceptable for your needs. Therefore, due to their lower TCO and greater flexibility, FPGAs are often the best technological choice.

Nanometers (nm) are a measure of the size of the transistors on a chip. The transistors have been shrinking for decades. See Chapter 2.

Choosing an FPGA for a system offers the designer greater configurability as well as less risk of impact to the development schedule because, as demonstrated by the building blocks analogy, small parts of FPGAs can be modified without impact to the rest of the design.

FPGAs Are Surprisingly Easy to Use

Some designers have the false impression that building a system with a modern FPGA means you have to mess with millions of logic gates and massive amounts of connections just to do something useful. But if that were the case, FPGA use wouldn't be growing: Instead, there would only be about a half dozen FPGA users left.

The good news is that FPGA designers have done much of the heavy lifting of adding commonly needed components like clock generators, dynamic random access memory (DRAM) controllers, peripheral component interconnect Express (PCI) controllers, and even whole multicore microprocessors, so all that you have to concentrate on is customizing those functions that are specific to your application.

Hard IP

Earlier in the chapter, I mention that *hard IP* is intellectual property built into the FPGA, such as DRAM controllers, PCIe controllers, clock generators, and big blocks of memory. In fact, there is so much hard IP in today's FPGAs that they really have become systems on chips (SoCs).

Not only are common functions that most system designers need built into the hard IP of the FPGA, but even many less-commonly needed functions like high-speed serial transceivers for radar or communications, and digital signal processor (DSP) multiplier-accumulators for signal processing can be included. Today, even dual-core ARM (ARM is a brand of microprocessor designs) CPU subsystems may be built in. In

fact, today's high-end FPGAs may have programmable logic on only half the die area, the other half consisting of hard IP. Designers now commonly start with an FPGA that has the IP they need already built in and then use the programmable logic to customize the FPGA to their particular application.

Parallel operation and order reduction

Modern designers have become quite good at making tools that are smart enough to reduce more complicated operations into simpler ones (this is known as *order reduction*) and also have the ability to perform a complex operation in a series of instructions that operate simultaneously, which is known as *parallel operation*.

So this all sounds great, but what does it really mean? A microprocessor can do just about anything simply by executing its instructions. If the program says to multiply, the microprocessor loads the instruction from memory, decodes it, loads each number, multiplies them, and stores the result. Each one of those steps takes time and energy. But what if all you wanted to do was multiply a number by 2? That is just a shift operation. In general, if you have a multiplication operation and you know one of the coefficients is a constant value, you can reduce this complex operation to a simpler one, saving you processing time and power. Microcontrollers, unlike FPGAs, don't have the smarts to reduce multiplication to addition when possible, so they have to perform the multiplication operation, resulting in slower execution speed and more power use.

FPGAs really shine when it comes to doing things like vector mathematics computations. Vector math isn't just for physics class: Programmers use it whenever they have to perform the same operation on each one of a large set of numbers. The big advantage of FPGAs here is that although a microprocessor would have to treat each number separately — or at best, take a few numbers at a time — you can program the FPGA to do many operations simultaneously (in parallel). If you have a 128-element matrix, you can build 128 arithmetic "pipelines" so all these operations can execute simultaneously, giving you huge gains in performance and power usage.

Often an ASIC or ASSP isn't the right answer!

Chapter 2

What's in an FPGA, Anyway?

In This Chapter

▶ Investigating the programmable fabric and I/O

▶ Taking a look at upward scaling

▶ Embedding hard IP and integrated CPUs

▶ Tackling the modern design flow

*I*f you're reading this chapter, it's a safe bet that you may know what an FPGA is. But if it's been some time since you learned about FPGAs, you will be surprised by how they've evolved in recent years to be more than an array of programmable logic gates. They now include built-in hardware to perform many common functions right out of the box. This chapter takes you through what's really in an FPGA, discusses the future of upward scaling, and walks you through how a design flow works.

The Basics — Programmable Fabric and I/O

As the name field-programmable gate array (FPGA) suggests, FPGAs are, at their core, simply integrated circuits that contain a bunch of logic gates and I/O circuitry. The I/O circuitry takes in data from a source and spits out data at the other end into some other system or subsystem.

In Chapter 1, I discuss the building blocks of an electrical system: logic gates, wires, and registers. At the core of an electrical system is a flat rectangle of silicon with wires and

transistors that are etched into its surface. These bits of silicon are known as *integrated circuits* (ICs).

TECHNICAL STUFF

Transistors are semiconductor devices usually made from silicon. Transistors are used to switch and/or amplify electrical signals and have at least three connectors (or terminals) that form a connection to the circuit. Transistors switch electrical signals by applying an electrical potential energy (voltage) across two of the terminals, and then applying another potential at the third terminal, which causes the current to flow from one end of the transistor to the other across the first two terminals. When this potential is removed, no current flows through the circuit. Transistors can also amplify the input power by applying a voltage or current that causes the power at the output of the transistor to be greater than that supplied at the input.

Transistors are similar to the spring-loaded valves used in plumbing. When you apply force to the valve, water flows through a pipe (a wire in this analogy). When you remove the force on the valve, it closes and water stops flowing through the pipe. The same is true for transistors. When you apply electrical potential (force) to the transistor, electricity flows through it and out the other end onto the wires connecting the circuit. When you remove this electrical potential, electricity stops flowing.

History of transistors

Historians credit the development of the transistor in 1947 to John Bardeen, Walter Brattain, and William Shockley. What makes the transistor such a remarkable invention is that it ushered in the age of *solid-state* electronics and the integrated circuit. Transistors are much, much smaller and use much less power than the vacuum tubes they replaced. The reduction in size allowed for smaller devices, which ultimately means you can carry a phone around with you that plays the latest cat video from YouTube while getting driving directions to your favorite French-Vietnamese fusion restaurant! If you are feeling really nerdy, you can delight and amaze your friends at the next party by explaining to them that the word *transistor* was coined by John R. Pierce as a combination of the term *transfer resistor.* This tidbit may also come in handy for fabulous cash and prizes on *Jeopardy!*

Boolean algebra is really numeric operations on input values of *true* or *false* or expressed numerically as 1 or 0. *Logic gates* are the devices that are used to perform the various Boolean algebra operations on input values of 0 or 1. Because transistors switch electrical signals by applying or removing an electrical potential, you can arrange a group of transistors in such a manner as to create a logic gate that can perform one of the Boolean algebra operations like AND, OR, or NOT (see Chapter 1 for a discussion of Boolean algebra operations).

The core of an FPGA is simply an array of these logic gates and wires etched into an integrated circuit in a way that allows you to reconfigure them. Or, if you prefer, take a look at the analogy I present in Chapter 1 that compares FPGAs to building block pieces arranged on a table.

So really, an FPGA, in its simplest form, is a big array of colorful rectangles placed on a rectangular table that can be arranged in the manner desired by its owner (or in the case of FPGAs, its programmer).

TECHNICAL STUFF

Moore's Law

Moore's Law originated from an article in the April 19, 1965, issue of *Electronics Magazine* called "Cramming More Components onto Integrated Circuits." In this article, Moore accurately predicted that circuit complexity would double every 2 years while the *wafer* (the flat silicon disk on which the integrated circuit is built) would remain constant. By the end of the 1970s, the most popular formulation of Moore's Law became the limit for the number of transistors on the most complex chips. Amazingly, Moore's prediction still holds true nearly 50 years after its initial publication!

Simply put, Moore's Law states that because you can shrink the size of the features you inscribe on the integrated circuit by 30 percent every 18 to 24 months, you can double the transistor count. Moore's Law does make things more complex and difficult. For example, adding all these wires and transistors to a smaller surface area of silicon makes it very difficult to maintain the placement integrity of the original circuit design, after processing, into the etched wafer of silicon. The projected images appear with oddities such as lines wider or narrower than designed, or distortions such as rounded corners on the wafer.

In the past, designers found only simple uses for FPGAs, such as simple interfacing to computers and to create basic logic functions. Beyond their use as devices for field programming interfaces, designers didn't do much else with them.

Scaling Upward

Designers have it covered when it comes to eliminating the problems that occur when etching circuits with what may add up to be millions of wires and transistors on a small wafer of silicon.

Another problem that occurs when scaling chip design to add millions, or even billions, of transistors is that it becomes very difficult to *pinch off* the transistors (cut off the electrical potential) because the transistors are so small. As chip design continues to double the number of transistors on a wafer, the design becomes more complex and the transistors leak more power, meaning that the chip even burns power when sitting around not doing any computations. The transistors also get weaker, so more work has to be done just to keep the chip from running slower.

Could the end of scaling be near? Designers already pack so much power into such a small area on a chip that it is possible for a chip to melt its own wires! Designers are at the point where they'll have to fundamentally change how they design transistors from a *planar* arrangement to a *FinFET*. A FinFET is a nonplanar mutigate field-effect transistor (FET) built on a silicon wafer designed to reduce the surface area a transistor occupies. The FinFET gets its name from the thin silicon "fin" that forms the conducting channel of the transistor.

What does the future hold for chip designers? In the not-so-distant past, a chip had about 20 transistors. In the next few years, chips will be built containing tens of billions of transistors. Before these chips can be designed, designers will have to start making transistors that stand up on their edges, different kinds of wires, and many other changes. Chip design will be on the edge of unexplored territory — chips will be able to diagnose, compensate, and heal themselves. Indeed, these are exciting times for chip designers as they continue to extend Moore's Law into the future!

REMEMBER

The application of Moore's Law has certainly been a winner for integrated circuits. Today's integrated circuits may contain millions or even billions of transistors and can perform very complex operations such as high-speed data networking, advanced 3D graphics computations, or streaming and playing movies across the Internet in high definition! Moore's Law doesn't only apply to integrated circuits — its application has revolutionized FPGAs as well.

Hard IP and Integrated CPUs

Today, implementing a design in an FPGA can often provide the same energy efficiency and speed as the same design implemented in an ASIC on the same size of hardware. This is true in part because FPGA vendors have embedded large blocks of predefined hardware into the FPGA to implement frequently needed functions, from standard interfaces to whole microcomputers.

The fact that FPGAs are *field programmable* gives them an edge over ASICs because the same hardware can be reprogrammed in response to changes in design. If the design uses an ASIC, changing the design requires the hardware to be scrapped and new hardware has to be built to reflect the design changes. More designers are choosing FPGAs over ASICs for their designs. If you're thinking about using FPGAs in your designs, read on to find out more about the design flow for designing with FPGAs while taking full advantage of their embedded hardware.

Modern Design Flow of FPGAs

FPGA designs often start with what are called *reference designs,* which represent a technical blueprint of a system that is intended for others to copy. Reference designs contain the essential elements of the system. Reference designs are typically done by applications engineers as part of a sales support effort, but the nature of the customer has changed. Increasingly, reference designs aren't sales tools — they're the product itself!

Creating a functional block diagram

So what does a system design flow look like? Figure 2-1 shows a simple block diagram of a high-level system design flow.
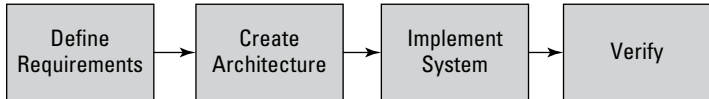
Courtesy of Intel Corporation.

Figure 2-1: System design flow.

The system design flow appears in the form that you'd expect. First you define the requirements, and then create the architecture of the system you define. Here, you determine the components you need to implement your design. Next, you implement the system using the architecture you planned out. Finally, you verify that the system meets all the requirements.

Figure 2-1 shows a simplistic view of the system design flow. The *Create Architecture* and *Implement System* steps are where the action is. Here is where you'll determine what the architecture of the system looks like and build the hardware and software applications required to implement the system design. You can further break out the steps between *Define Requirements* and *Verify* into a separate flow that can be called the software *application flow*. Figure 2-2 adds the steps of the *software application flow* to the system design flow in Figure 2-1.
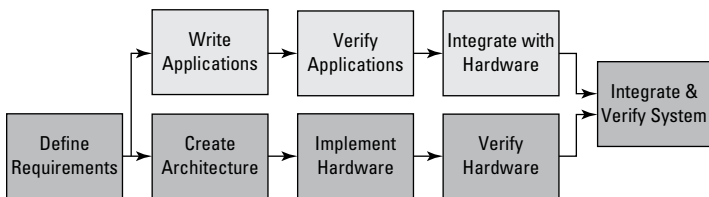
Courtesy of Intel Corporation.

Figure 2-2: System design flow with the software application flow.

The lighter colored blocks between *Define Requirements* and *Integrate & Verify System* make up the application flow of the design. In this step, you write and verify the applications

software and then integrate those applications with the hardware. After the applications are integrated with the hardware, you integrate and verify that the system meets the design requirements.

The application flow is all about developing the applications needed to implement the system. Designers often must consider how their systems will run on different platforms depending on what type of application the system will be deployed into (for example, automotive, communications, and so on). Often, different application domains have established software and hardware standards to ensure that applications developed for these systems perform common functionality and can work with each other.

Think about an Android-based phone, for example. The Android operating system contains common features that can be used by all applications developed for it and uses a platform standard for such things as accessing the camera and sharing data among applications. Designers often include what is typically called *middleware* into their applications. Middleware is a layer of software that doesn't implement the core functionality of the product, but instead, provides a layer that implements an industry standard or protocol. Designers create middleware to isolate their application logic from logic that is specific to a particular standard or development platform (for instance, Android or Apple iOS). Middleware can often be reused in many other applications. Another important feature of adding middleware to applications is that it makes it possible to incorporate future standards and platforms easily into your application. Figure 2-3 shows how middleware development fits into the system design flow.
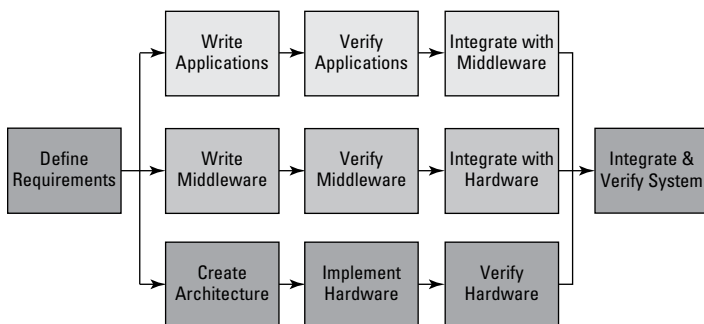
Courtesy of Intel Corporation.

Figure 2-3: Middleware in the system design flow.

Replacing functional blocks with existing IP

Looking at the block diagrams, it seems like your work is cut out for you when it comes to building and integrating your applications into the system. But FPGA manufacturers have learned over the years that most systems require many of the same types of functionality. Functionality such as network data I/O, graphics processing, and microprocessors is commonly needed, so it doesn't make sense for each system designer to design and build these components. It makes much more sense for these types of functionality to be available *out of the box*. In recent years, FPGA manufacturers have been including such common functionality or intellectual property (IP) into their products. This IP can be in the form of hardware built into the chip, software provided to the user, or — just for FPGAs — hardware designs that the user can drop into the programmable logic. Now, you can replace parts of the blocks in the block diagram with existing IP — the work that has already been done for you. Figure 2-4 shows an illustration of where existing IP fits into the design diagram.
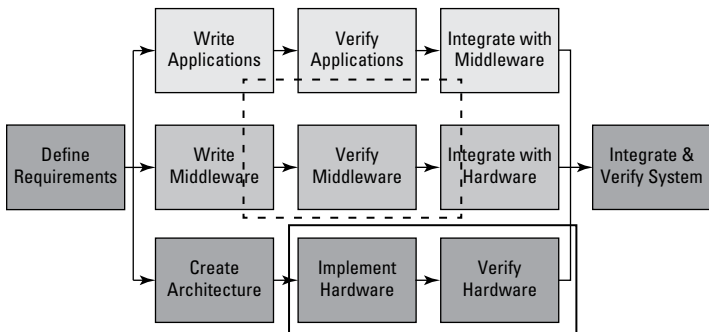
Courtesy of Intel Corporation.

Figure 2-4: Replacing functional blocks with existing IP.

The solid rectangle shows the effect of using hardware and programmable-logic IP: Many of the Implement Hardware and Verify Hardware steps are done for you. In addition, the dashed rectangle in Figure 2-4 shows where the existing IP fits into the design diagram. Here it replaces part of the work in the writing of applications and middleware with functionality implemented for you.

The real bug in bug

If you're really into word origins, then you'll be fascinated to know that the first use of the word "bug" in software was attributed to computer pioneer Grace Hopper. In 1947, she uncovered a moth trapped in a relay of an electromechanical computer and referred to the resulting glitch in the program execution as a bug!

Coding the missing blocks

Existing IP can only go so far by implementing some common functionality, for example, accessing GPS data for an on-board navigation system. The rest of the field programming work of the FPGA is left up to the designer. After all, you want it to exactly fit your applications, right?

Programming modern FPGAs is much easier than you might think. The steps to programming an FPGA include identifying any blocks of the design that you actually want to design yourself, choosing a high-level or hardware description language (HDL), writing the code in a text editor, synthesizing (more on that later) the design, placing and routing the design, then loading the design onto the FPGA itself. After the design is loaded onto the FPGA, it may require a cycle of *debugging* to fix errors in functionality.

You'll come across the word *bug* in the technical jargon of software and hardware development. A bug is an inexplicable defect in computer software that produces an incorrect or unexpected result. The term *debugging* refers to eliminating defects until the whole design runs according to its required functionality.

Once you're happy with how the design works, the next steps are to document the program and finally ship it to the customer.

So, how do you program those blocks you can't just import from a library? In the early days of FPGAs, you had one choice: an HDL that, as the name suggests, described the blocks in hardware terms — wires and gates. This limited FPGA programming to hardware engineers and determined

enthusiasts. But today, high-level software-programming tools like OpenCL allow software designers without special hardware skills to specify the function of a subsystem, compile their code, and create a hardware-description file that they can feed into the normal FPGA design flow. Such tools allow software designers to create hardware accelerators with only minimal — or in some cases, no — support from hardware engineers, opening the power of modern FPGAs to a much wider audience.

For blocks that still require more detailed design, designers will use an HDL. *Verilog* is a common HDL used in creating designs for FPGAs. Verilog has a syntax very similar to the commonly used, general-purpose programming language called C. But instead of defining a program to run on a computer, Verilog, VHDL, and other hardware description languages describe the hardware — the interconnected network of gates, registers, and wires — that the designer wants to create in the FPGA. You write your Verilog programs in the proper syntax using a simple text editor.

After you write the HDL design, the next step is to compile the HDL design. In FPGA programming, a synthesis tool takes the HDL design as input and converts it into a network of gates, registers, and wires configured to implement the functions the HDL describes. Then additional processes select which particular gates, registers, and wires to use in the FPGA and create a programming file that will configure the FPGA when it powers up.

So your HDL code gets mapped directly into the physical hardware elements available on the selected FPGA device. In microprocessor programming, program logic gets mapped into a list of processor instructions that the processor must execute. So it is quite a different — and wonderful — feature that you can convert your logic directly to silicon gates for execution.

During this process, the design tools may also link into the design of the *hard IP* — the predefined blocks of hardware already embedded in the FPGA. In the modern tool flow, you only have to specify whether you want to use hard or soft IP blocks and how you want them connected. You only need to write HDL code for any blocks that aren't already available as IP.

Verifying the system design

After you compile your code, test it before deploying onto your FPGA. In the old days, designers tested their designs for much simpler programmable-logic chips by simply trying them out to see if they worked. But because of the complexity of modern FPGAs, plugging-and-trying as an early debug tool isn't feasible.

Debugging an FPGA design is typically done in a simulation environment. Simulators are software applications that (as you might expect) simulate the behavior of your design. But the simulation is done using software where you can see what the individual registers are doing before you put the design into the FPGA.

Debugging and verifying code is typically repeated until you're certain that the HDL code works as intended. Most developers use what is called a *testbench* as a tool to verify that the FPGA will work when attached to the real world. A testbench can be a mix of software simulation and actual hardware — designed by you — that makes up a model of the system that will contain your FPGA. Most FPGAs contain tens of thousands or hundreds of thousands of gates, so you can't test all of them. Instead, a testbench focuses on the most meaningful gates that contain the critical areas of your design. Simulation environments help you to isolate particular areas and add debugging aids in those areas so you can get your design working the way you want.

TIP

Any good software application requires extensive documentation that tells the customer or end-user exactly how the application is defined, and brings up any caveats, warnings, and so on. The documentation requirements are the same for FPGAs as in any microcontroller-based programming; of course, the contents will differ greatly.

Mapping the system into FPGA hardware

In the end, the bits that have been synthesized must be loaded into the FPGA to implement the gates of the system.

Like any system, if the hardware is correct, the design can evolve to include bug fixes and feature enhancements. The ability to edit the HDL code allows for design, debug, and

verification in the same environment, which helps you get a faster time-to-market using an FPGA.

Trying out the design in the system

Once the design is programmed into the hardware, ensure that everything works as it is supposed to. What does *working* really mean for an FPGA? This stage is sometimes called *closure*. And, as with any hardware device, certain performance criteria are expected. In many applications, power consumption is an important design criterion. Think of your smartphone, for example. Smartphones have strict power requirements so that they can maintain an acceptable battery life. You wouldn't want your smartphone to be a power hog; otherwise you'll find your battery dead after a short period of use. Speed is another important criterion. Test to ensure each *net* (wire connection between gates) meets its timing limit. Finally, ensure that every clock and power pin is connected on your FPGA.

In the FPGA design-tool environment, you can enter your design, select blocks of IP to include, and convert the design into the hardware elements that actually exist in the FPGA. Then, while the design is still in software and easy to test, you can verify that it works as expected, that it works at the required speed, and you can even estimate how much power the design will consume.

Now comes the magic: You can load your tested design into the target FPGA on your prototype board, power up the system, and verify that everything is working as expected. You now have custom hardware to exactly fit your requirements — months before you would receive the first sample chips of an ASIC.

Chapter 3

FPGAs as Systems

In This Chapter

▶ Understanding FPGAs as functional building blocks

▶ Absorbing the system into the FPGA with SoCs

*T*his chapter shows how FPGAs are used in the real world. An FPGA is really a functional building block of a system, and as FPGAs grow larger, the whole digital system can be absorbed into an FPGA, making it a system on a chip (SoC).

In this chapter, I examine the concept of an electronic system. Then I take you through SoC FPGAs in the bewildering complexity of a high-end car.

FPGAs in System Design

This section gives you a deeper look into the system design process and how FPGAs play an important role (I talk about some of the basics in Chapter 2).

Figure 3-1 shows the traditional model of system design. The diamond shapes you see between each block represent decision points in the process.
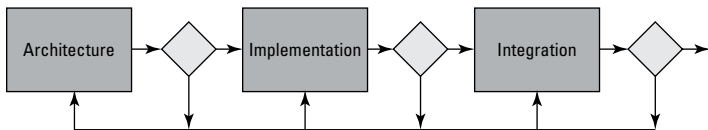
Courtesy of Intel Corporation.

Figure 3-1: System design with decision points.

The decision points in the system design are the points where you have to ask some questions such as:

✔ **What does the system have to do?** This question appears during the requirements definition phase and is the fundamental question. The answer to this question is often provided by the product manager in coordination with the customer and drives the requirements gathering phase.

✔ **Can I use my existing design with changes?** Often, an existing system design is in place that, with some changes, may meet the requirements of the system.

✔ **How much of the system can I leave in software?** This is an important question asked during the design and implementation phase. The amount of the system that can be left in software determines what types of hardware can be used. FPGAs and microcontrollers can be used to program the software.

✔ **How much hardware can I buy off the shelf?** Many times, a functional block of your system design may already be implemented in a commercially available hardware device (known as *off-the-shelf*). If that's the case, it may be more economical to purchase this hardware, or license it as IP, rather than implement the design in software or design custom hardware.

✔ **Does it work yet?** This is the fundamental question during the integration stage and must be answered "Yes" before the system can be deployed. If the answer is "No" then you must keep iterating over the design and implementation until you get it right.

In Figure 3-2, you can see that during the requirements definition, designers must consider the constraints on their design, such as performance, power consumption, and size. The functions of the system are also important — including which functions are visible, invisible, or locked. Finally, designers run experiments on their design. This process leads to early system estimation, which indicates the actual size and scope of the system and what it will take to implement.
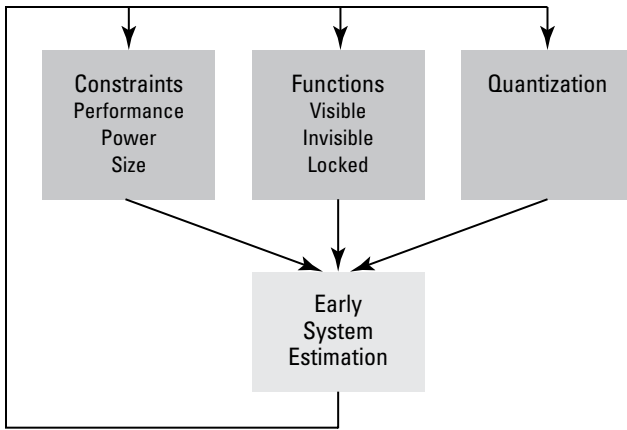
Courtesy of Intel Corporation.

Figure 3-2: Defining system requirements.

The design process for FPGAs is an iterative one — you start with an idea of the system and then refine the idea into definitions of *transactions*. A transaction would include input, processing, and output. You can think of a transaction just like a bank transaction where you hand money over to the teller, the teller takes the money, and then adds the money to your account. Basically, a transaction is anywhere in the system where information is shared between two components of the system.

After you define the transactions, you implement them, and then verify that they work and meet the functions and constraints you established at the start. This is another iterative process that requires deciding which functions will be done in hardware and which in software, which can be done with existing IP, and which will have to be newly written.

Automotive Electronics Systems with FPGAs

This section examines a real-world example of a system. In fact, you may have driven this system to work this morning. I show how SoCs — many of which could be FPGAs — fit into a modern car.

Drivetrain

Consider the components that make up a car's drivetrain, and how under regulatory, safety, cost, and feature pressures they have all become electronic:

- ✔ **Engine:** Electronics in the engine control the fuel, ignition, and valves based on power demand, emissions, smoothness, starting cycle, and strategy.

- ✔ **Transmission:** Modern transmissions include electrical systems to control gear ratio, shifting sequence, signals based on speed, power demand, and engine rotations measured in revolutions per minute (RPMs).

- ✔ **Brakes:** For safety, electronic systems — not just the pedal on the floor — control the braking force.

- ✔ **Steering:** High-end automobiles have sophisticated power steering features that control the ratio, feedback, and angle of steering based on many inputs, and the electronics may be able to take over control of the steering.

- ✔ **Tires:** In recent years, advancements in automobiles include electronic sensors that monitor tire pressure so that drivers know when to inflate their tires if needed — improving both fuel economy and tire life.

Infotainment

Infotainment is a nifty word used to describe the information and entertainment systems in automobiles. Many cars have sophisticated electronic infotainment features such as:

- ✔ **Displays and controls:** Today's cars have electronically controlled speedometers and other readouts.

- ✔ **Entertainment:** Modern cars have advanced features such as digital AM/FM radio, satellite radio, CDs, and digital audio players where you can store your entire music library. There is sometimes also a digital video system that will keep your kids happy during those long road trips!

- ✔ **Comfort:** Now, the driver and passengers each have access to the lighting and multizone climate control systems so they can be as hot or cool as they want without impacting others in the same car.

✔ **Access control:** Cars today come equipped with power locks, doors, windows, and security systems. There are also common safety features like window and door lock controls that keep your children from opening doors and windows while the car is in motion.

✔ **Passive safety:** Safety systems know how many occupants are where in the car, and make appropriate preparations if they sense an impending collision.

Driver assistance

Driver assistance includes some of the coolest technology that has come along in automotive design in recent years. It makes cars safer than ever! Driver assistance systems include:

✔ **Lights, back-up, lane-exit, and collision avoidance:** Cars often come equipped with advanced lighting systems, indicators, and heads-up displays that warn the driver when the car swerves outside of the lane or is about to collide with another vehicle or object.

✔ **Sensors including cameras, lasers, and radar:** These sensors are used to allow drivers to see in their blind spot when backing up or changing lanes, which drastically reduces the chances of an accident.

Importance of FPGAs

Today, most automotive systems depend on low-cost micro-controllers that operate at the point of sensing or action. The trend in automotive design is for consolidation of systems and for systems to become more autonomous. As systems become more sophisticated, their processing and memory requirements skyrocket. Consider sensor fusion with Kalman filters.

Sensor fusion is the combining of sensory data from disparate sources so that the resulting information is better than what would be obtained from these sources individually, such as stereoscopic vision (the calculation of depth information by combining two-dimensional images from two cameras at slightly different viewpoints). A Kalman filter is an algorithm that uses a series of measurements, often from different kinds of sensors, observed over time that contain *noise* (random variations) and produces estimates that are more precise than

those based on a single measurement alone. Kalman filters are commonly used for guidance, navigation, and control of vehicles.

As automobile systems consolidate, microcontrollers are being absorbed into SoC implementations. As these systems get smarter and more autonomous, SoCs are evolving into multicore processor/DSP clusters. In order to control the explosive growth in the number of models, changes during the model year, evolution in bus architectures, and the continuous demand for better security, the trend is moving toward SoC FPGAs being the only viable answer to solve these design challenges and needs for frequent updates.

REMEMBER

The car is only one example of how systems become more dependent on electronics, the electronics become more complex and change more rapidly, and the need grows for SoCs that can change even during a model year. This same pattern shows up across a huge range of products with complicated behavior, from aircraft and trains to electric power grids to even home appliances. Yes, even your toaster.

Chapter 4

The Future: Heterogeneous Computing and OpenCL

In This Chapter

▶ Looking at heterogeneous computing

▶ Examining OpenCL

*I*ndustry trends are driving FPGAs toward playing a big part in the heterogeneous computing paradigm. Open Computing Language (OpenCL) is an industry standard development platform used to program FPGAs in a heterogeneous environment.

This chapter walks you through why heterogeneous computing is necessary and the emergence of new languages for creating software to execute on them.

Heterogeneous Computing

Inside data centers, one of the major trends is a shift in the computing architecture: from multicore CPUs to heterogeneous computing. *Heterogeneous computing* refers to systems that use more than one type of processor to perform specialized processing capabilities. An example of a heterogeneous computing system is a graphics rendering system that uses a CPU and a graphics processing unit (GPU) to render 3D graphics on a computer. GPUs are especially adept at rendering 3D scenes and performing mathematically intensive computations on large datasets. CPUs are used in the background to perform operating system

and data networking tasks. Heterogeneous computing is becoming more of the standard as systems consolidate and must incorporate several different processor architectures.

Parallel computing is the capability of computers to perform many calculations simultaneously based on the principle that large problems can be broken down into smaller problems and then solved concurrently *(in parallel)*. Parallel computing comes in many different forms: bit-level, instruction level, data, and task-based. Parallel computing is no longer just the domain of high-performance computing like IBM's chess-mastering Big Blue. As power consumption has become more of a design factor in embedded electronics, parallel computing has become the dominant paradigm in computer architecture, most commonly seen in the form of multicore processors.

Data parallelism focuses on the idea of separating data across multiple processors so that it can execute in parallel. Multicore processors often do this by farming out multiple instances of a program to each of the processors to execute these instructions simultaneously. *Task parallelism* has to do with a processor farming out computer code blocks known as *threads* across different processors to execute in parallel.

Why Use OpenCL on FPGAs?

The need for heterogeneous computing is leading to new programming languages to exploit the new hardware. One example is OpenCL first developed by Apple, Inc. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs, and other types of processors. OpenCL includes a language for developing *kernels* (functions that execute on hardware devices) as well as application programming interfaces (APIs) that allow a main program to control the kernels. OpenCL allows for parallel computing using task-based and data-based parallelism.

In the last decade or so, processor hardware frequencies have hit a so-called *power wall,* which prevents higher frequencies from being achieved on processors. Instead of dramatically increasing clock frequencies, CPU manufacturers have been busy adding more processing *cores* to their CPUs and

enhancing their instruction sets so several instructions can execute at the same time, speeding up program execution without requiring faster clock frequencies to do so. Software companies have been busy as well, developing software that allows chunks of computer code known as *threads* to execute in true parallel fashion. The threads may execute on separate processor cores instead of the pseudo-parallelism of the past where threads weren't executed on separate cores but time-sliced by the operating system to appear to be running in parallel.

REMEMBER

FPGAs are inherently parallel, so they're a perfect fit with OpenCL's parallel computing capabilities. FPGAs give you an alternative to the typical data or task parallelism by also offering pipeline parallelism where tasks can be spawned in a push-pull configuration with each task taking data from the previous task with or without host interaction. OpenCL allows you to develop your code in the familiar C programming language. Then, using the additional capabilities provided by OpenCL, you can separate your code into normal software and kernels that can execute in parallel. These kernels can be sent to the FPGAs without you having to learn the low-level HDL coding practices of FPGA designers. Generally, there are several benefits for software developers and system designers to use OpenCL to develop code for FPGAs:

- ✔ **Simplicity and ease of development:** Most software developers are familiar with the C programming language, but not low-level HDL languages. OpenCL keeps you at a higher level of programming, making your system open to more software developers.

- ✔ **Code profiling:** Using OpenCL, you can profile your code and determine the performance-sensitive pieces that could be hardware accelerated as kernels in an FPGA.

- ✔ **Performance:** Performance per watt is the ultimate goal of system design. Using an FPGA, you're balancing high performance in an energy-efficient solution.

- ✔ **Efficiency:** The FPGA has a fine-grain parallelism architecture, and by using OpenCL you can generate only the logic you need to deliver high performance.

- ✔ **Heterogeneous systems:** With OpenCL, you can develop kernels that target FPGAs, CPUs, GPUs, and DSPs seamlessly to give you a truly heterogeneous system design.

✔ **Code reuse:** The holy grail of software development is achieving code reuse. Code reuse is often an elusive goal for software developers and system designers. OpenCL kernels allow for portable code that you can target for different families and generations of FPGAs from one project to the next, extending the life of your code.

Today, OpenCL is developed and maintained by the technology consortium Khronos Group.

Most FPGA manufacturers provide Software Development Kits (SDKs) for OpenCL development on FPGAs.

Chapter 5

Five Applications of FPGAs

*F*PGAs have come a long way since the old days, when implementing a complex system with an FPGA meant that you had to do lots of programming of the logic gates. Today's FPGAs come with built-in capabilities, such as network interfaces, memory blocks, and even ARM cores. Intel FPGAs with built-in ARM cores are known as SoC FPGAs, a recognition of the role these powerful chips are playing. The field-programmable part is now sometimes less than half of the chip area. This chapter highlights how FPGAs are used in several areas of industry and technology.

Single-Device Motor Control

Motors and motor control are commonplace in any industrial design. When you go to any factory or industrial, you'll find a variety of widely different machines with one thing in common — they're powered by motors. Most motor control systems are designed with microcontroller technology. However, microcontrollers can fall short of the performance demands of sophisticated motor-control algorithms such as direct torque control (DTC) or sensorless field oriented control (SFOC), for example. DSPs have been used in the past to get around that problem, but are usually unable to cost-effectively match an FPGA when it comes to high performance. You can build a flexible, scalable, and high-performance motor control system in a single SoC FPGA (see Figure 5-1).
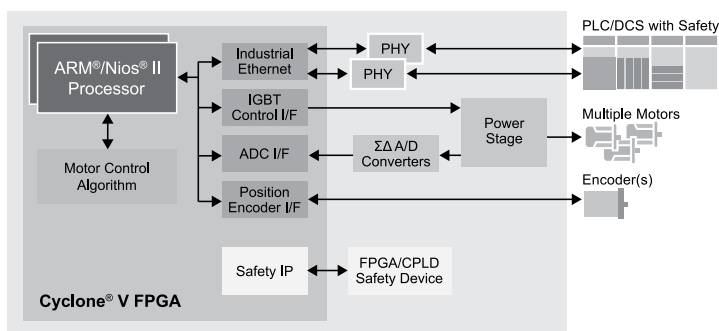
Courtesy of Intel Corporation.

Figure 5-1: Motor control.

One example is using an SoC FPGA to plug into a motor con-
trol module that comes with two independently controlled
DC motors and a simple optical feedback system. The SoC
FPGA includes a built-in processor that manages the feedback
and control signals so that the two motors can move inde-
pendently. The processor reads the data from the feedback
system and runs an algorithm to synchronize the movement
of the motors as well as control their rotation speeds. By
using an SoC FPGA, you can build your own IP that can be
easily customized to work on other motor controls. There are
several advantages to using an SoC FPGA for motor control
instead of a microcontroller:

- **System integration:** Fewer parts lead to less material costs,
  lower power requirements, and fewer reliability challenges
  by integrating industrial networking, safety, power stage
  interfaces, and DSP control algorithms on a single device.

- **Scalable performance:** You can use a single scalable
  platform across an entire product line. SoC FPGAs allow
  you to achieve higher performance with faster and more
  advanced control loops that can increase efficiency and
  machinery lifetime.

- **Functional safety:** As automation takes more of the
  responsibility for running potentially dangerous equip-
  ment, regulators are requiring the machine-control elec-
  tronics to guarantee that no harm can result. With an SoC
  FPGA and the right design flow, you can reduce time and
  effort complying with these government and industry
  safety regulations.

Television Broadcasting

Television broadcasters use a serial digital interface (SDI) standard to transmit uncompressed digital video on 75-ohm coaxial cable (the same kind that hooks your cable/satellite receiver or antenna to your television). With every improvement to video images, the standard has had to bump up its capacity. The latest standard is called the 3-Gbps (3G)-SDI, and is capable of moving 4K ultraHD signals around the studio. With all this change, here's another area where FPGAs really shine! FPGA solutions come with a core transceiver that can function on all three SDI rates (SD SDI, HD SDI, and 3G-SDI) on the same transceiver.

But much else has changed in the studio as well. New digital techniques help edit the video stream, improve or correct picture quality, and compress the image for transmission over cables or satellite links. The latest compression standard, H.265 (also known as the High-Efficiency Video CoDec) slashes the number of bits necessary to encode a movie or TV program. But it requires an enormous amount of computation. Many equipment vendors are finding that the best solution to pack the power into an SoC while responding to the pressure for rapid evolution — there is that combination again — is an FPGA. See Figure 5-2.
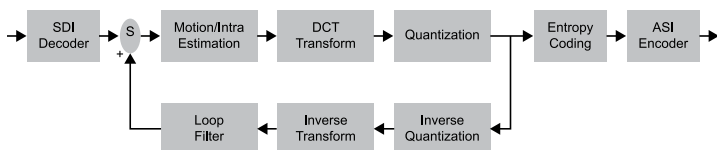
Courtesy of Intel Corporation.

**Figure 5-2:** Broadcast.

Wireless Data

Nothing has changed how people live and work more than the arrival of 4G wireless technology. It allows you to carry

around those phones that not only let you make calls from wherever you are but also allow you to browse the web and post updates from anywhere! The latest technology migration is from the 4G standards to 5G technologies.

REMEMBER

The key requirements for base stations and mobile operators are scalable form factor, low power consumption, low cost, and programmability as they strive to reduce their expenses while expanding and upgrading their networks. Manufacturers are also looking to increase productivity and time-to-differentiate as keys to introduce successful products and get a competitive edge.

Many FPGAs now come equipped with built-in low-latency intellectual property (IP) for advanced networks as well as productivity enhancing tools to allow manufacturers to leverage FPGAs advantages of performance, power, price, and productivity to focus their efforts on product differentiation and not on the mechanics of programming the nuts and bolts of the wireless infrastructure. See Figure 5-3.
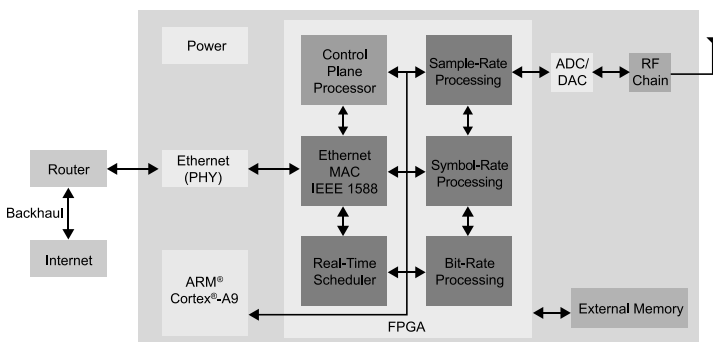
Courtesy of Intel Corporation.

Figure 5-3: Wireless infrastructure.

Automotive Driver Assistance Cameras

One of the big areas of growth in the automobile industry is the explosion of technology-driven features. Even lower-priced automobiles come equipped with fancy gadgets like navigation systems, video entertainment systems, and cameras.

Driver assistance and backup cameras are some of the most important safety innovations and help make cars safer than ever. The peace of mind that comes with knowing that all is clear behind your car when backing up is priceless.

Forward camera systems are made up of high-speed video processing, complex sensor fusion, and real-time data analysis that enable the automobile to perform corrective action in cases like when the driver nods off and veers into another lane. Forward cameras do their job by integrating with different sensors such as radar and laser sensors. Each type of sensor is different in how it provides data, posing a design challenge for multiple architectures.

Traditional DSP processors or microcontrollers don't have the power to do real-time video processing and analytics at the same time. Moreover, HDR or high dynamic range, which is a requirement for the camera to see equally well into both bright and dark areas of a scene, is a necessity for video analytics to be accurate. HDR processing can as much as triple the demand for video signal processing power as compared to a traditional non-HDR camera, taking the performance requirements out of reach for all but the most expensive DSPs. Instead of DSPs or microcontrollers, you can integrate the entire camera system in a single, low-cost SoC FPGA. You can optimize system performance by developing hardware parallel processing engines using FPGA logic and integrating with software algorithms running on the hard processor system of an SoC FPGA. Figure 5-4 shows a diagram of an SoC FPGA as part of an automobile vision system.
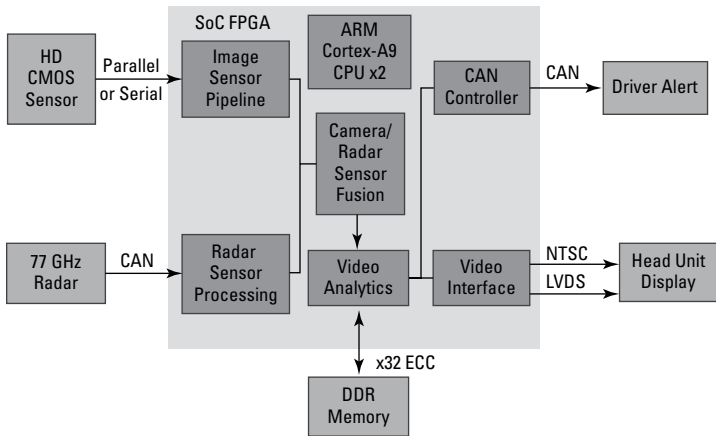
Courtesy of Intel Corporation.

Figure 5-4: The FPGA in an automobile vision system.

High-Performance Computing

The high-performance computing (HPC) market is one of the fastest growing areas of computing today. It is extremely important in many industries, such as financial, medical imaging, bioscience, military, and many others that can benefit from the logic and memory resources in FPGAs to develop application-specific coprocessors. Think about financial markets, for example, and the mind-boggling amounts of data that go from place to place for all those trades, forecasts, and price calculations. Fractions of a cent matter in these transactions, so high-speed, accurate floating-point arithmetic is absolutely essential.

In HPC, floating point is a numerical representation where a series of digits or bits represent real numbers. Applications require floating-point data types for more accurate results than integer calculations can produce. Floating-point operations require more processor logic and hence more power. Common floating-point applications include:

- Fast Fourier transform (FFT)
- Radar
- Bioscience
- Finite impulse response (FIR) filters
- Financial options trading
- Matrix math (used extensively in 3D graphics and image processing)
- Molecular dynamics
- Seismic and medical imaging

TECHNICAL STUFF A *coprocessor* is a computer processor that is used to supplement the functionality of the primary or central processor (CPU). Coprocessors are typically used to perform floating-point arithmetic, signal processing, string processing, encryption, or I/O interfacing to peripheral devices. Coprocessors take on computationally intensive operations, freeing the CPU to service the core functions of the computer.

All HPC markets require coprocessors to provide a productivity, performance, and power advantage.

The good news is that the latest Intel FPGAs build in not just DSP functions, but floating-point hardware, so that programmers don't have to convert their programs from floating-point format to integer format before running them on an FPGA-accelerated server. This capability is a huge boon in categories such as:
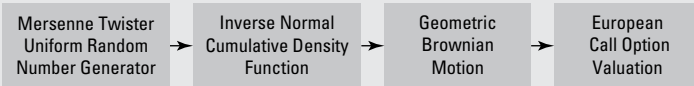
- **Data appliances:** Database and financial market acceleration
- **Functions:** Random number generators for financial markets, one-million point FFT for military and signal processing applications
- **Algorithms:** SRCs CARTE, Impulse, and AutoESL system generating algorithms

Case study: The Monte Carlo Black-Scholes method

One of the most important benchmarks in financial markets is the computation of option prices via the Monte Carlo Black-Scholes method. In financial terms, an *option* is a contract that gives the buyer the right, but not the obligation, to buy or sell an asset at a specified price on or before a given date. The Monte Carlo Black-Scholes technique is based on conducting random simulations of the underlying stock price and averaging the expected payoff over millions of different paths. The accompanying figure shows a graphical representation of this method.

of random numbers is fed to an Inverse Normal Cumulative Density Function (a probability function to specify the distribution of random numbers) in order to produce a normally distributed sequence of numbers. These random numbers are used to simulate the movement of stock prices using Geometric Brownian motion (an algorithm commonly used to predict stock prices). At the end of each simulation path, the call option payoff is recorded and averaged to produce an expected value for the payoff. The entire algorithm can be implemented in about

| Mersenne Twister Uniform Random Number Generator | → | Inverse Normal Cumulative Density Function | → | Geometric Brownian Motion | → | European Call Option Valuation |

Courtesy of Intel Corporation.

In most computer simulations, designers use some type of random number generator to simulate the data input into the simulation to model the randomness of the real system. In the Monte Carlo Black-Scholes method, modelers typically use a random generator known as a *Mersenne* twister. The *Mersenne* twister random number generator is a very fast and high-quality generator of pseudorandom numbers — it's ideal for simulations. This sequence

300 lines of OpenCL code that is portable from an FPGA to a CPU and GPU. The FPGA solution outperforms both the CPU and GPU in power, performance, and efficiency as shown in the accompanying figure.

In the figure, the FPGA is compared with a CPU and GPU in three criteria: power consumption, the number of simulations per second, and the rate of power-efficiency simulations per second. FPGAs are inherently

Platform	Power Watts (W)	Performance Simulations per Second (Bsims/s)	Efficiency Simulations per Second Watt (Msims/s/W)
CPU	130	0.032	0.0025
GPU	212	10.1	48
FPGA Solution	45	12.0	266

Courtesy of Intel Corporation.

parallel — meaning they can be coded to break complex calculations into computations that can be done in parallel. FPGAs can do more operations in parallel than can CPUs and GPUs, resulting in much faster execution and increased power efficiency, both of which are essential in electronic systems design.

Harness the power and flexibility of FPGAs in your system designs

Are you an engineer who has been hearing the buzz about FPGAs? Maybe you learned about FPGAs way back in college, but your knowledge is rusty? Relax! With this friendly reference by your side, you'll make informed choices about using FPGAs in your designs.

• *Understand FPGAs and how they work* — *discover what FPGAs are and how they do what they do*

• *Investigate programmable fabric* — *discover the programmable fabric of an FPGA and how you can design and develop your functionality with it*

• *Incorporate the FPGA as a functional block in your system* — *investigate how your system is absorbed into the FPGA as a system on a chip (SoC)*

• *Encounter the world of heterogeneous computing* — *find out how heterogeneous computing is changing the ways systems are developed and how they communicate*

Andrew Moore is a software engineer with over 15 years of software design experience. **Ron Wilson** is a messaging strategist at Intel. His background includes engineering and technology journalism.

Open the book and find:

• **The modern design flow of FPGAs**

• **The pros and cons of adding FPGAs to your design**

• **Uses for FPGAs in system designs in many industries including military and automotive applications**

• **The future of FPGAs in system design including heterogeneous computing and OpenCL**

Go to Dummies.com®
for videos, step-by-step examples, how-to articles, or to shop!

FOR DUMMIES®

A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.