1

INTRODUCTION

em·bed *— to make something an integral part of*

Merriam-Webster Online

From smart phones to medical equipment, from microwaves to antilock braking systems — modern embedded systems projects develop computing machines that have become an integral part of our society. To develop these products, computer engineers employ a wide range of tools and technology to assemble embedded systems from hardware and software components. One component — the Field-Programmable Gate Array (FPGA) — is becoming increasingly important. Informally, an FPGA can be thought of as a "blank slate" on which any digital circuit can be configured (Figure 1.1). Moreover, the desired functionality can be configured in the field — that is, after the device has been manufactured, installed in a product, or, in some cases, even after the product has been shipped to the consumer. This makes the FPGA device fundamentally different from other Integrated Circuit (IC) devices. In short, an FPGA provides programmable "hardware" to the embedded systems developer.

The role of FPGA devices has evolved over the years. Previously, the most common use of the technology was to replace a handful of individual small- and medium-scale IC devices, such as the ubiquitous 7400 series logic, with a single FPGA device. With well-known improvements and refinements in semiconductor technology, the number of transistors per IC chip has increased exponentially. This has been a boon for FPGA devices, which have increased dramatically in both programmable logic *capacity* and functional *capability*. By capacity, we are referring to the number of equivalent logic gates available; by capability we are referring to a variety of fixed, special-purpose blocks that have been introduced, both of which are discussed in more detail throughout this book.

As a result of this growth, modern FPGAs are able to support processors, buses, memory controllers, and network interfaces, as

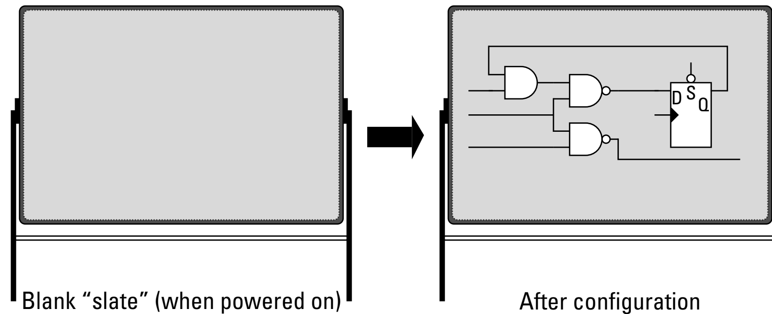


Figure 1.1. A 10,000-meter view of an FPGA; a blank slate that can be configured to implement digital circuits after the chip has been fabricated.

well as a continually increasing number of common peripherals, all on a single device. With the addition of a modern operating system, such as Linux, these FPGAs begin to appear more like a desktop PC in terms of functionality and capability, albeit on a single IC chip. Furthermore, with Open Source operating systems comes a plethora of Open Source software that can be leveraged by FPGA based designs.

We use the term *Platform FPGA* to describe an FPGA device that includes sufficient resources and functionality to host an entire system on a single device. The distinction is somewhat arbitrary, as there is no physical difference between a large FPGA device and a Platform FPGA device. Rather it is a matter of perspective: how the developer intends to use the device. We would say that an ordinary FPGA is generally used as a peripheral and plays a supporting role in a computing system. In contrast, a Platform FPGA has a central role in the computing system.

Overall, FPGAs offer a great deal to the embedded systems designer. Beyond simply reducing the numbers of chips, the Platform FPGA offers an enormous degree of flexibility. As it turns out, this flexibility is extremely valuable when designing systems to meet the complex and demanding requirements of embedded computing systems that are becoming so universal today. Often it is this flexibility that makes FPGA technology more appealing than traditional microprocessor-based solutions, Structured ASIC solutions, or other System-on-a-Chip (SoC) solutions.

Along with its advantages, FPGA technology also comes with a new set of challenges. Previously, one of the most important questions for an embedded systems designer was the choice of the processor. That choice usually dictated much of the remaining architecture (or at least limited the range of design choices). With Platform FPGAs, in order to fully utilize their capabilities and cost savings, a much wider pool of system developers needs to be able to combine computer engineering, programming, system analysis, and technical skills.

The aim of this book is to draw this necessary information together into a single text so as to provide the reader with a solid and complete foundation for building embedded systems on Platform FPGAs. This includes the underlying engineering science needed to make system-level decisions, as well as the practical skills needed to deploy an assembled hardware and software system.

The organization of this book reflects that twofold mission: each chapter consists of a set of white pages followed by a set of gray pages. The white pages emphasize the more theoretical concepts and slow-changing science. The gray pages provide descriptions of state-of-the-art technology and emphasize practical solutions to common problems.

The specific learning objectives of this chapter are straightforward.

- We begin with an abstract view of a computing machine and define an embedded system by distinguishing it from a general-purpose computing system. Since we are using a Platform FPGA device to develop application-specific, custom computing machines, it is important to revisit the traditional concepts of hardware and software and, in particular, describe the two very different compute models used by each.
- Next, we consider some of the specific challenges that embedded systems designers face today. These include short product lifetimes leading to tight project development schedules, increased complexity, new requirements, and performance metrics that often define the degree of success. The complex interplay of these challenges presents one of the most important problems facing the next generation of embedded systems.
- The white pages of this chapter end with a section describing Platform FPGA characteristics and why these devices are well suited to meet these complex demands facing modern embedded systems designers today.

  In the gray pages, a practical example using a fictitious scenario is presented; we briefly describe how to set up the software tools needed throughout the book and show to how to create the obligatory “Hello, World!” on a Platform FPGA development board. Overall, the goals of this chapter are to establish a solid motivation for using Platform FPGAs to build embedded systems and to initiate the reader on how to use the development tools.

1.1. Embedded Systems

In the simplest sense, an embedded system is a specialized computing machine. A *computing machine* (or just *computer*) is

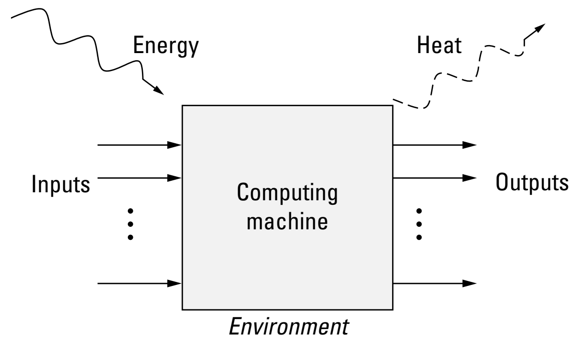


Figure 1.2. An abstract view of a computing system.

frequently modeled as a system that includes inputs, outputs, and a computation unit. The machine exists in some sort of *environment* that provides the energy to propel the machine. In addition to the manipulation of information, the computing machine produces heat that is returned to the environment. This organization is illustrated in Figure 1.2. (This very abstract figure is fleshed out in the next chapter.)

When the machine is being used, it is said to be executing and that period of time is called "run-time." When the machine is not executing, it is said to be "off-line." The inputs, which come from the environment, determine the outputs, which are conveyed from the machine to the environment. The inputs and outputs are often physical signals and as such are assigned meaning depending on the context of the problem, which is the point of building computing machines: to solve problems.

The personal computer (or desktop) is a well-known example of a computing machine but there are a multitude of others as well. The slide rule and the abacus are old-fashioned mechanical machines that were used to do arithmetic. These have since been replaced by the electronic calculator, which is an electronic computing machine.

1.1.1. Embedded and General-Purpose

Depending on how the computing machine is used, we classify it as either embedded or general-purpose. An *embedded computing system* (or simply an embedded system) is a computing machine that is generally a component of some larger product and its purpose is narrowly focused on supporting that product. In other words, it is a computing machine dedicated to a specific purpose. In terms of the abstract machine in Figure 1.2, the environment of an embedded computing system is the product it is part of. The

end user of the product typically does not directly interact with the embedded system, or interacts with only a limited interface, such as a remote control. Even though the computer may be capable of more, the embedded system is typically restricted to perform a limited role within the enclosing product, such as controlling the behavior of the product.

There are numerous examples of embedded systems. They are ubiquitous and have permeated most aspects of modern life: from the assortment of consumer electronics (DVD players, MP3 players, game consoles) we interact with almost daily, to energy-efficient refrigerators and hotel key-card activated door handles. Even large, heavy earth-moving equipment typically has hundreds of sensors and actuators connected to one or more embedded systems.

In contrast, a *general-purpose computing system* is a product itself and, as such, the end user directly interacts with it. Another way to put it, the end user explicitly knows the product they are buying is a computer. General-purpose systems are characterized by having relatively few, standardized inputs and outputs. These include peripherals, such as keyboards, mice, network connections, video monitors, and printers. Embedded systems often have some of these standard peripherals as well, but also include much more specialized ones. For example, an embedded system might receive data from a wide range of special-purpose sensors (accelerometers, temperature probes, magnetometers, push button contact switches, and more). It may signal output in a variety of ways (lamps and LEDs, actuators, TTL electrical signals, LCD displays, and so on). These types of inputs and outputs are not typically found in general-purpose computers. Furthermore, how the inputs and outputs are encoded may be device specific in an embedded system. In contrast, general-purpose computers use standard encodings. For example, the movement of a mouse is typically transmitted using low-speed serial communication and many manufacturers' mice are interchangeable. Another example is each key on a keyboard has fixed standard ASCII encoding. While an embedded system may also use these standards, it is just as likely that it will receive its input encoded in the form of a pulse frequency. Both may look similar electrically but how meaning is attached to the signals is very different.

Ultimately, such a precise definition of an embedded system can be difficult to agree with depending on what an individual considers a "larger product." For example, is a handheld computer an embedded system? If so, what is its enclosing product? It is not used to control anything. Furthermore, the computing system is exposed to the user: the user can download

general-purpose applications. So by the criterion just described, it is not an embedded system. A handheld computer shares many characteristics of embedded systems. (For example, like an embedded system, handhelds are very sensitive to size, weight, and power constraints.) For this reason some might be inclined to call it an embedded system. Now add a mobile phone application to the system and most would agree that this is an embedded system. With the addition of some specialized inputs and outputs, a general-purpose system can be viewed as an embedded system. So, the exact boundary between general-purpose and embedded is not black and white but rather a spectrum.

General-purpose systems strive to strike a balance, compromising performance in some areas in order to perform well over a broad range of tasks. More to the point, general-purpose computers are engineered to make the common case fast. Since embedded systems commonly have a more narrowly defined purpose, designers have the benefit of more precise domain information. This information can be used during the product development to improve performance. Often, this can lead to very substantial improvements.

For example, a general-purpose computer may be used to process 50,000 frames of video from a home camcorder and then later the same computer may need to handle spreadsheet calculations while serving World Wide Web content. For a single user, this computer has to handle these functions (and many others) reasonably well. As long as other issues, such as energy use, are reasonable, the user is happy. However, an embedded system designer might know that their system will never need to do spreadsheet calculations and the engineer can exploit this knowledge to better utilize the available resources.

Most computer courses target general-purpose computing systems. Operating systems, programming languages, and computer architecture are all taught with general-purpose computing machines in mind. That is appropriate because — embedded or not — many of the principles are the same. However, given the enormous presence of embedded computing systems in our day-to-day lives, it is worthwhile to accentuate the unique characteristics of embedded systems and focus a program of study on those.

1.1.2. Hardware, Software, and FPGAs

One of the differences between general-purpose computing and embedded computing is that embedded systems often require special-purpose hardware and software. It is worthwhile to define these terms before we proceed. Furthermore, Platform FPGAs

begin to blur the distinction between hardware and software. Before we get into the details of FPGA technology, though, let's revisit the definitions of hardware and software.

Hardware refers to the physical implementation of a computing machine. This is usually a collection of electronic circuits and perhaps some mechanical components. Simply put, it is made of matter and is tangible. For example, consider a computing machine implemented as a combinational circuit of logic gates (AND, OR, NOT) and assembled on a breadboard. The hardware is visible and the design physically occupies space. Another characteristic of hardware is that all of the components are active concurrently. When inputs are varied, changes in the circuit propagate through the machine in a predictable but not necessarily synchronized manner.

Software, however, is information and as such does not manifest itself in the physical world. *Software* is a specification that describes the behavior of the machine. It is generally written in a programming language (such as C, MATLAB, or Java) and this representation of desired machine behavior is called a *program*. While it is possible to print a program on paper, it is a representation of the program that exists in the physical world. The software is information expressed on the printout and is inherently intangible. (Note that a person who is composing software is called a *programmer* and that the act is called *programming*.)

These traditional definitions of hardware and software have served us well over the years, but with the advent of FPGAs and other programmable logic devices in general, these definitions and the distinction of what is hardware and what is software become less clear.

1.1.3. Execution Models

One of the most challenging aspects of embedded systems design is the fact that one has both hardware and software components in the design. These have fundamentally different execution models, which are highlighted.

Several programming paradigms exist for High-Level Languages (functional, object-oriented, etc.). However, at the machine level, all commercial processors use an imperative, fetch-execute model with an addressable memory. In software, this sequential execution model serializes the operations, eliminating any explicit parallelism. As a result, operations are completed in order, alleviating the software programmer from the arduous task of synchronizing the execution of the program. In contrast, hardware is naturally parallel and computer engineers must explicitly include synchronization in their design. This concept of

sequential, implicitly ordered operations versus the unrestricted compute model is especially important in FPGA designs and is a topic that we will continue to address throughout this book.

A subtle point is that it is commonplace to refer to some components of our Platform FPGA system as “hardware” when, in fact, these components are written like software. That is, the component is a specification of how the device is to be configured. The real, physical hardware is the FPGA device itself. Thus, in place of the conventional definition of hardware given earlier, we will distinguish hardware and software components of our system based on their model of execution. Earlier we said that software is characterized by a sequential execution model and that the hardware execution model is distinctly nonsequential. In other words, software will refer to specifications intended to be executed by a processor, whereas hardware will refer to specifications intended to be executed on the fabric of an FPGA and, on occasion, in a broader sense, the physical components of the computing system. To make this important distinction more concrete, consider the following computation:

```
R0+R1+R2 → Acc
```

Assume that all four names are registers. The computation implemented in the sequential model is illustrated in Figure 1.3. It shows the traditional fetch-execute machine. The dashed box highlights on the main mechanism. The ALU circuit is time-multiplexed to perform each addition sequentially. A substantial part of the circuit is used to decode and control the time steps. This is commonly referred to as the von Neumann stored-program compute model. We will continue to refer to it as the sequential execution model. In contrast, a nonsequential execution model can take a wide variety of forms, but is clearly distinguished from sequential execution by the absence of a general-purpose controller that sequences operations and the explicit time-multiplexing of named, fixed circuits. A nonsequential execution implementation of this computation is illustrated in Figure 1.4. This is generally known as a *data-flow model* of execution, as it is the direct implementation of a data-flow graph. (We use the broader sense of the term throughout this book; the data-flow architectures studied in the 1980s and 1990s incorporate more detail than this.)

In terms of how these two computing models work, consider their operation over time. In Figure 1.5, time advances from top to bottom. Each “slot” delineated by the dotted lines indicates what happens during one clock cycle. It is important to note that the speed of the two computing models cannot be compared by counting the number of cycles — the clock frequencies for each

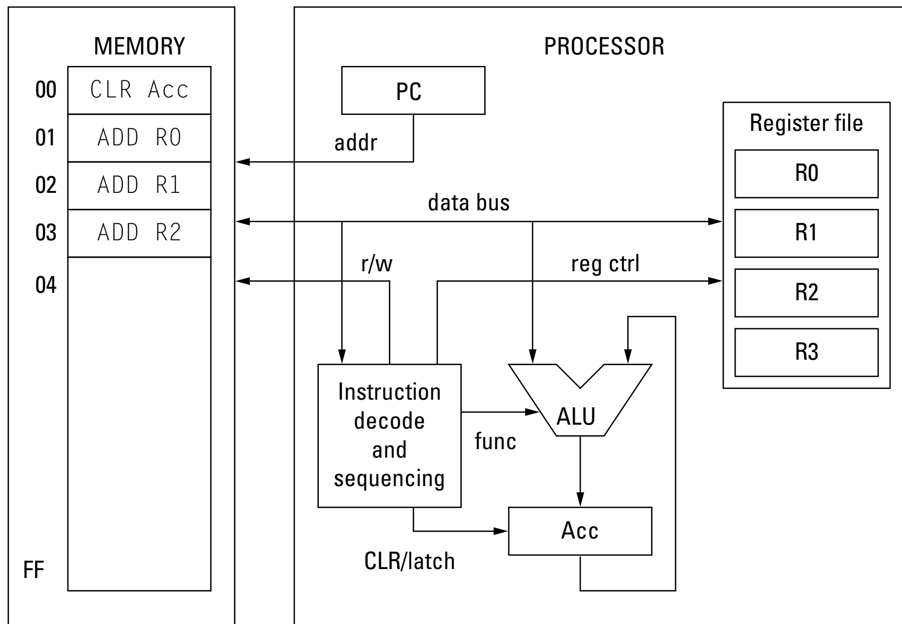


Figure 1.3. A sequential model.

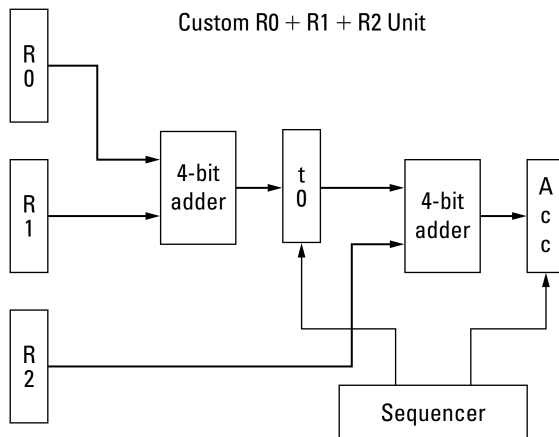


Figure 1.4. A nonsequential model.

are different. A typical FPGA clock period might be 5× to 10× longer than a processor implemented in the same technology. Also, modern processors typically operate on multiple instructions simultaneously, which has become crucial to their performance.

With this understanding, we can say that software runs on a processor where a *processor* is hardware that implements the

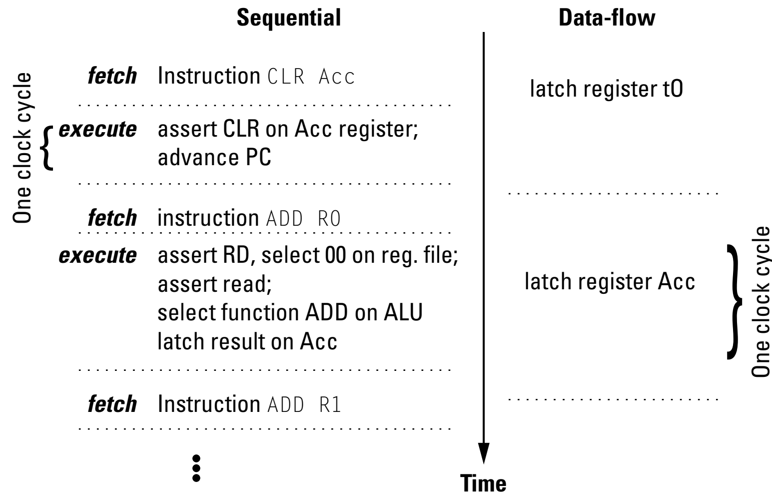


Figure 1.5. A cycle-by-cycle operation of sequential and data-flow models.

sequential execution model. Note that the terms Central Processing Unit (CPU) and microprocessor are common synonyms for processor. Hardware will be the specification that we use to configure the FPGA and does not use the sequential execution model.

1.2. Design Challenges

Now that we have a better understanding of what an embedded system is, it is worthwhile to understand a little bit about how embedded systems projects work. This section discusses the life cycle of a typical project, describes typical measures of embedded system success, and closes with a summary of costs.

1.2.1. Design Life Cycle

There are many books on how to manage an engineering project. Our goal in this section is simply to highlight a couple of terms and describe the design life cycle of a project to provide context for remaining sections. This life cycle, illustrated in Figure 1.6, is sometimes referred to as the “waterfall model.” It is meant to suggest that going upstream (i.e., returning to a previous stage because of a bug or mistake in the design) is considerably more challenging than going downstream.

Whether initiated by a marketing department or by the end user, the first stage is called *requirements*. As the name suggests, the step is to figure out what the product is required to do. Often this will start as a very broad, abstract statement, and it is the developer’s job to establish a concrete list of user measurable

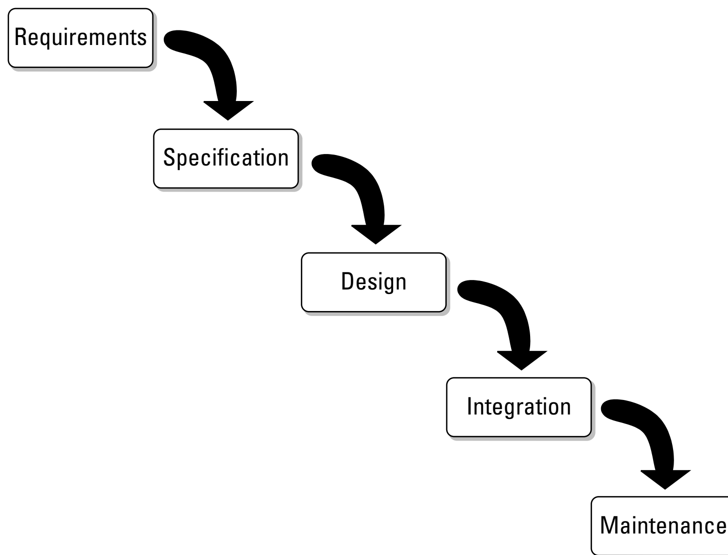


Figure 1.6. A"waterfall model" description of a project life cycle.

capabilities that, if met, the end user would agree the project was a success.

The next stage in is called *specifications.* In contrast to requirements, specifications are usually written for the developer, not the end user. Created from the requirements, they are used to judge whether a particular piece of code or hardware is correct. Often a specification comes from a particular requirement, but some specifications may result as the combination of several requirements or as the result of preplanning the design of the product. It is reasonable to develop a set of metrics at this stage that define the correct behavior (in terms of implementation).

The *design* stage follows next. Usually a developer is contemplating a working design throughout all of the previous stages; this is where the design is formalized. Formats, division of functionality into modules, and algorithm selection are all part of this stage. There are numerous methodologies for how to design software and some include developing a prototype, whereas others call for reiterating over specifications and requirements. Others espouse coding a solution as fast as possible under the adage that "get your first design finished as soon as possible because you always throw out the first one." For FPGA designs, a developer is typically designing the physical components, electrical and mechanical, at this stage as well.

The fourth stage — called *integration* (or in some methodologies, "testing") — begins when the physical components become available. All of the components are combined, and the

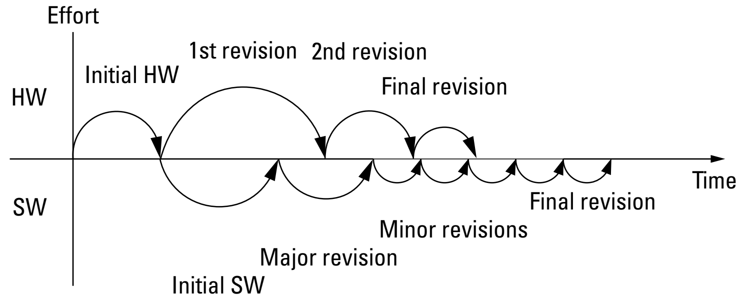


Figure 1.7. Rate of revisions in developing hardware and software.

functionality tests developed during the specification stage are applied. For custom Printed Circuit Boards (PCBs), there is usually an incremental process of (1) testing the electrical connectivity, (2) applying low-voltage power and verifying the digital components, (3) testing the high-voltage components (if any), and then finally loading the software. The process is called "bringing up a board," and the idea is to minimize potential damage due to any mistakes in the design or fabrication process. By testing incrementally, it also helps locate the mistakes that inevitably will exists.

After integration, the product is generally ready to be delivered (if it is a one-off custom design) or manufactured in mass. The final stage, *maintenance*, begins once the product has left integration. As the name suggests, it deals with fixing problems that were discovered after the product begins to ship. For FPGA-based designs, there is much that can be done in this stage because, after all, the device is programmable in the field! More to come on that later.

An alternative view of the design and integration stages is shown in Figure 1.7. The point of this illustration is to show the hardware (top) and software (bottom) revisions of the designs. Often, before software development can begin, some initial design effort is needed to produce the first hardware (first silicon). This can also be the time waiting for a development board to be fabricated. Once the hardware arrives, the software development can swing into full gear. Each arc represents another revision, where the large area under the arc indicates a major revision and smaller arcs represent minor revisions. A project may only have so much time to revise hardware, so the relatively few major revisions are shown. Software may be able to make many small revisions, especially toward the end of the design process.

1.2.2. Measures of Success

For the last several decades in the general-purpose computer world, the top design consideration has been speed (rate of

execution). Computing machines were compared based on how fast they completed a task. In the embedded systems world, success is a combination of *requirements* and *performance metrics*. The requirements are capabilities that the system must meet to be functionally correct. Metrics are measured on a continuum. Also, an embedded systems design is, by definition, constrained by its environment (i.e., the larger product it is part of). This necessarily means that success is a multifaceted measure.

Clearly, the requirements and specifications guide the design of an embedded system. The requirements will define capabilities that must be met. Often there exist additional criteria (which may or may not be explicitly provided) that help judge the success of the product. We call these extra criteria performance metrics. Metrics are generally measured on a scale, such as “faster is better” or “lighter is better.” For example, a six-week battery life might be a requirement but a product may also be judged by its battery life metric. A product that typically runs for eight weeks between charges is generally considered better than one that simply meets its six-week requirement.

Of course, a developer does not want to blindly “over engineer” a product because there are real costs associated with many decisions. In order to make sensible decisions given a fixed amount of resources, a developer needs quantifiable metrics as a guide. There are a number of quantifiable metrics that embedded systems developers may use; three common metrics are considered here: speed, energy, and packaging.

Speed

Throughout most computer-related undergraduate curricula, the idea of speed is theoretical. That is to say an algorithm is evaluated in terms of the number of time steps it takes to complete. Usually this is written in terms of the size of its input, n. This is a very effective tool for comparing the relative merits of algorithms, but for embedded systems, it is not as useful. Embedded systems, for example, typically run until the power is turned off — so picking a fixed n as an input is often impossible! Instead, it is more useful to measure the speed of the individual components.

If the computing machine has a dedicated purpose, then the speed may be characterized as a *minimum speed* requirement. In this case, the speed can be thought of as a “cliff” function: if you are heading toward a cliff, being able to stop in time is sufficient but not stopping in time has dramatic consequences. For many dedicated systems, there are limited benefits in exceeding the target speed, but falling short of the target does not gradually degrade the performance of the system.

For scientific applications, the speed — that is, the rate of computation — is usually the metric that trumps all others. Heat and packaging are important; however, as long as the machine can be physically built and the heat generated by the machine removed is at a reasonable cost, then the key factor that distinguishes one system from another is how fast it completes a given task. Likewise, for many business applications, reducing the amount of time that an end user spends waiting for a result provides a competitive edge over another machine. This is sometimes characterized as "how fast the time piece goes away" in reference to a mouse pointer icon that indicates the machine is working. In these and other general-purpose cases, performance increases as the speed of the machine increases. For such systems, an *unbounded speed* requirement means that faster is better.

Unlike many general-purpose systems, embedded systems incorporate both speed requirements. When the computing machine is used for controlling the product and the speed is not directly observable by the end user, then there is usually just a minimum speed requirement. The machine just has to be "fast enough" in order for the product to function correctly. If the end user *can* observe the speed, then an unbounded speed requirement may be in effect. For example, suppose the task is to skip forward to the next song in the playlist. An unbounded speed requirement is reasonable here because if the target is two seconds, then exceeding this results in a better end-user experience. Faster response times can translate into a measurable advantage. If the embedded system is designed to increase the energy efficiency of a household appliance, then exceeding the minimum speed actually degrades the overall performance (because a faster system presumably consumes more energy).

Speed requirements are usually specified for each use case. This often results in multiple tasks that, to some extent, share resources in the system. Operating systems can be used to manage those shared resources by creating a virtual environment for each task and then time-multiplexing the tasks. Ordinary operating systems schedule these tasks in real time with an assumption that the tasks have unbounded speed requirements. However, embedded system applications often need a mechanism to relate tasks' progress in their individual virtual times to the events in real time. An operating system that provides these mechanisms is called a real-time operating system.

Scheduling in real-time operating systems is challenging because, for a variety of reasons, the amount of time a particular task is going to take is not always known. Hence, before we leave

the speed component of performance, it is worthwhile to note that *predictability* is valuable. Predictability can improve the scheduler of a real-time operating system, which allows us to more carefully specify minimum speed requirements.

Energy and Power

All machines — whether they are mechanical or solid state — are propelled by energy and it is hard to imagine any human activity in our 21st-century society that does not consume energy. Twenty years ago, small (slow) stationary computing machines (desktops) did not consume much energy. Hence, power — the rate at which energy is consumed — was a relatively insignificant issue. However, with rising energy prices and ever-faster microprocessors, power has become a first-class design metric. Moreover, all of the energy that goes toward computation is eventually converted into heat. For low-power systems, the ambient environment may be sufficient to dissipate the heat. As power consumption increases, active thermal measures need to be taken, which further increases the total power used by the system. In some very large high-end computing installations today, 2 MWatts are required to power the computing machine and 1 MWatt is needed to run the chillers! For embedded systems, there are two energy-related components to this issue. First is the total energy used to complete some task or application. The second is power.

Total Energy

Many computing systems are application specific; that is, there is a well-defined lifetime for the machine or there is a duty cycle. Examples of the former might include disposable greeting cards that play a musical tune when the card is opened or a wireless sensor that is built into a bridge to take structural integrity measurements. Examples of the latter include most rechargeable portable devices — mobile phones, personal digital assistants, and so on.

In both cases, the computing system must include some energy storage unit — a battery. The functional specification will probably be expressed in terms of time (i.e., a cell phone must operate for two days between charges). Battery technology is improving; however, it has not kept up with increased power demands of general-purpose microprocessors. Further complicating this picture is the fact that batteries tend to be an expensive and physically large component of the system. Consequently, for many projects the range of battery capacities is relatively restricted and it is up to the engineer to design the computational unit to fit a total energy budget.

For some applications, the total energy over the lifetime of the product is all that matters, since operating system cost is proportional to energy. If the product has a very short lifetime (imagine a greeting card with a small battery-powered audible message) or if the product is wired (like your desktop computer), then the analysis ends there. Ultimately, most embedded systems don't fall into either of these situations and so managing the total energy consumed is an important performance metric.

Power and Heat

While total capacity of a battery constrains the lifetime (or charge cycle) of a computing system, there is another dimension to the problem. Specifically, most energy storage units are limited in terms of how quickly they can release their stored energy. In other terms, energy may be available but the battery cannot meet the power demands at a particular moment in time.

To make these ideas more concrete, a quick review of electrical properties is in order. Starting with some power source, such as a battery, there is a voltage difference between its two terminals. This difference, measured in volts (v), is the potential energy that will be dissipated in an electrical circuit connected to the battery. A circuit provides a path for electricity to flow from one terminal to the other, and the amount of electrical charge that flows is current, measured in amps (i). Depending on how the path from one terminal to another is constructed, the circuit will limit the current. This is the resistance r of the circuit. These three variables are related by Ohm's law, which is $v = ir$. Once the circuit is connected and current flows, the circuit begins to convert the potential energy of the battery into heat, which is the rate of energy consumption, which we already know is power. Power (measured in Watts, p) is a product of the change in voltage and current, that is, $p = vi$. (For computing devices, this only is only part of the picture — without taking switching into account this only measures the static power.)

The rate at which energy is consumed by the circuit is proportional to the rate at which heat is released into the environment. Passive thermal management techniques are limited in how fast they can remove heat. If heat is not removed, then the temperature rises, which has a series of problems, including device failure and discomfort for the end user, who may have the device on their person. To keep the temperature low, we have to resort to removing the heat with active thermal management (fans and more exotic techniques). These, of course, require power themselves, complicate the design, and raise the cost.

To bring this discussion to the point: power is dictated by the circuit technology we use and by how we organize our machine.

While the total energy used is one factor, the rate of energy consumption is much more significant because it affects so many of our performance metrics, such as size and mass. Poor power budgeting can raise the cost significantly.

Size and Packaging

Often size (and mass) is a requirement: the product has to fit within some space or cannot weigh more than a certain amount, but once met, they can also be metrics. For example, thinner mobile phones and lighter earpieces are better. For embedded systems design, size matters. If a designed circuit board is too large to fit in the packaging, it may result in a redesign of the circuit or a reorder of the packaging. Both are likely to be expensive mistakes. Therefore, it is important to consider what packaging or size constraints exist prior to circuit layout or fabrication. Clearly, these size and packaging issues are easy to quantify, so we will not belabor it here.

1.2.3. Costs

The previous two subsections are, strictly speaking, performance related. Cost is usually held in opposition to performance. An implicit goal of nearly every engineering project is maximizing performance while minimizing cost. Furthermore, cost constrains all of the metrics discussed previously, thus cost fundamentally limits the range of potential solutions. The cost of developing a product and selling it includes development and Nonrecurring Engineering (NRE) costs and then manufacturing and distributing costs.

While the business aspects of bringing an embedded systems product to market are better left to people schooled in business, it is important for the developer to at least be aware of the business aspects. In particular, there are two points we want to emphasize. The first is related to delivery dates, market demand, and how projects are financed. The second point is how technology — especially Platform FPGAs — can be used to reduce development/ NRE costs and the risk of escalating development costs.

Costs Incurred

Personnel

Perhaps the most obvious cost to an engineer is the development cost. These costs can be divided into two categories: direct and indirect. Direct costs include things such as salaries and hourly wages for employees developing the product, consumable materials and supplies, and education costs. Special equipment needed for development is also a direct cost.

Indirect costs are sometimes called overhead and are basically "the cost of doing business." More precisely, overhead pays for infrastructure. This cost is determined as a percentage of the direct costs. Assuming the project is part of some bigger organization, the project will be charged indirect costs based on what the institution provides. For example, a research project housed in an educational institution might be charged anywhere from 40 to 100% of the indirect costs. Typically, large industrial organizations charge upward of 100 to 200% overhead. (That's correct — if the direct costs are $50,000, total budget might be $150,000!) The justification for indirect cost depends on the type of institution as well. Organizations charge indirect costs to cover building capital costs and staff salaries.

It may not be immediately obvious why indirect cost is charged as a percentage. Why not simply pay for all services directly? The answer lies in the fact that most organizations have multiple ongoing projects and it is inefficient to dedicate a resource (such as an administrative assistant) to a single project. Moreover, the demands on that resource are likely to vary over time. The idea is that an active project would require more services, so it generates more overhead for the organization. By treating expenditures (direct costs) as a reasonable approximation of project activity, different projects can be charged based on their approximate demands on the infrastructure.

The general cost formula is:

$$TOTAL = DIRECT + RATE \times (DIRECT - EQUIPMENT)$$

Because the cost of large equipment purchases would generally skew the indirect costs, it is usually not included. For example, a project with $100,000 direct costs (of which $25,000 is a special-purpose milling machine) would be charged overhead on $75,000.

Nonrecurring Engineering Costs

After the product has been developed, there may be a prototype or just a construction plan. The next step is manufacture, distribution, and marketing. While this is primarily a business concern, it is important for an engineer to understand how products are manufactured, as the design decisions can have a significant impact on these costs. Generally, these costs are divided into two components. The first, NRE costs, are upfront charges that are required to build the first unit. The second are material and labor charges, which are basically the per-unit cost for some lot size.

Nonrecurring engineering costs are usually some sort of tooling charge or one-time software development costs, for example, the production of ASICs (Application-Specific Integrated

Circuits). This process was originally designed to make inexpensive product-specific integrated circuits widely available. The idea is to keep many parts of the process identical, but leave some layers customizable by the designer. The customizable parts are used to generate a mask that can be used to make silicon wafers quickly and easily. Generating a mask is a steep NRE cost that is proportional to the IC technology used. While individual IC devices might cost less than $1 per unit to manufacture, the initial mask cost might be over $250,000!

Finance and Consumer Demand

Most consumer electronic devices have an imaginary "consumer demand curve." This curve represents the total number of potential sales of a product over a period of time. Unlike older technology (such as hammers), consumer electronics change rapidly and newer devices replace older devices (such as game consoles). Whereas a consumer will use a hammer until it breaks, many consumers will stop using a functional mobile phone and replace it with a newer phone. Depending on when your product arrives relative to the customer demand curve and how quickly it is adopted dictate the total income derived from the product. We adopt Vahid and Givargis' model (Vahid & Givargis, 2002) and extend it to explain how development delays hurt the potential income but also bear significant costs that may not be immediately obvious.

The on-time model is illustrated in Figure 1.8. Time is on the x axis and the expected delivery date is at the origin. Everything to the left is development, manufacture, and delivery. To the right of the origin is after an on-time delivery. The y axis represents an account balance — below the x axis, the project costs have exceeded its income, above the x axis indicates that the project is profitable. In an ideal world, the consumer demand curve begins at the origin, will peak, and then asymptotically approach the x axis again. (It is easiest to think of "demand" as the number of potential units sold per time period. If there are no discounts and all of the costs are carefully accounted for, the number of units can be translated in potential profit.)

With an on-time delivery, we can see that a certain amount of monthly development costs will accrue between the start of the project (point A) and the origin. Once the embedded systems product has been been completed, a factory is contracted to produce a lot. This lot is then distributed to retailers at point B. If the product has no competition (and we contracted for a big enough lot), then the income will follow the consumer demand curve and, at some point, our account balance comes out of the

Figure 1.8. The importance of time-to-market in terms of break even and profitability.

"red" and the project becomes profitable. What we want to learn from this illustration is: what is the impact of delays during development? Delays during development that change the delivery date doubly hurt profitability. Suppose the actual delivery date is at point C. We continue to incur monthly development costs, and the consumer demand between points B and C is lost income that will never be recovered. (Furthermore, the financing charge continues on the original development cost estimate plus the additional costs.) Thus, missing a deadline for a consumer electronic device is not just delayed profits. In fact, it means that a diminished potential income will be available to recover a larger amount of debt.

While this section could continue into a variety of different product/project management and finance discussions, it is not the intention of this book to dwell deeply into these topics. Likewise, the brevity of the discussion is not to diminish the significance of the role they play in embedded systems design. In short, embedded systems design is a multifaceted endeavor and it is in the best interest for the reader to become familiar with as many aspects to the design process beyond the engineering solution. Meanwhile, this book will continue with a more engineering, science, and technology focus of embedded systems design.

1.3. Platform FPGAs

As highlighted earlier, embedded systems designers have increasingly turned to the use of FPGAs in their designs. While the remainder of this book addresses the specifics of "how" to use FPGAs, we

first must explain "what" are and "why" use FPGAs. Before FPGAs there were programmable logic devices (PLD). Early PLDs were designed to implement arbitrary digital (combinational) logic circuits. They were implemented as ICs with an array of gates (inverters, AND, and OR) and wires. Various mechanisms were used to make connections between the wires and the gates. The key feature of these devices was their ability to be configured after the manufacture of the device. A single PLD package is able to replace several packages of small-scale integrated circuits (such as the 7400-series parts). Hardware engineers began using the verb *program* to describe the process of setting the configuration. Hardware engineers would typically figure out the settings, and only the last step of transferring the configuration was the act of programming the device.

The FPGA is an advancement on these earlier programmable logic devices and is generally distinguished by two features. First, the gates are no longer physically implemented, but rather the logic is implemented with function generators and discrete memories. Second, most FPGAs use static RAM cells to hold the configuration information as opposed to more permanent ways of earlier devices (such as antifuses). This allows the FPGA to be configured and reconfigured after the device has been installed in a product. That is, an FPGA can be reconfigured in the field.

Field-Programmable Gate Arrays permit an arbitrary digital circuit to be realized after the physical silicon chip has been manufactured, tested, and installed. The digital circuit that is programmed (or configured) into an FPGA is called a *design*. Design entry is a generic term for creating a machine-readable representation of the desired digital circuit. We will return to this in the next chapter. A *Hardware Description Language* (an HDL) is a programming language used to describe the behavior of hardware and is the most common form of design entry today. Originally, HDLs were only intended to simply describe hardware for documentation purposes. Later, it was recognized that the HDL could be used to simulate hardware for testing the design before implementing it. Before long, engineers began to refer to the intangible HDL *design* as hardware because everyone understood that eventually it would be physically implemented. The definition of hardware was further co-opted when HDLs began to be used as a synthesis language. At this point, the common practice of referring to the specification as hardware becomes misleading: in an FPGA those gates described are never physically realized. The FPGA device is the hardware, and programming it does not manifest any physical change. (As shown in Chapter 2, the FPGA device is simply constructed to mimic the desired digital circuits.) Thus, even though

the design is, strictly speaking, intangible like software and the hardware is the physical FPGA device, the current convention is to refer to the design as “hardware.” This broadens the definition of hardware given earlier to include not only the tangible physical components of the computing system, but also the intangible specifications that use a nonsequential execution model.

We are now ready to give a more refined definition of a Platform FPGA. In 2002, Xilinx introduced the Virtex-II family of FPGA devices. The Virtex-II Pro members of this family incorporated one or more PowerPC processors as diffused intellectual property (IP) cores, to be explained shortly. Coupled with other diffused IP, such as RAM and high-speed transceivers, this meant that it is possible to deploy a complete, fully functional computing system on a single FPGA package. Moreover, with the large amount of configurable resources on the die, standard and nonstandard peripherals can be readily incorporated. Because it is configurable, the resources can be used to efficiently implement application-specific architectures. Naturally, this makes an excellent foundation (or platform) upon which to build an embedded system. We use Platform FPGA to designate an FPGA device that is deployed as the central component of a computing system. Indeed we will see that the presence of a diffused processor core is not always strictly required: with large FPGAs, the processor can be implemented in the reconfigurable resources of the FPGA.

It is useful to define some of these “hardware” components more carefully now. First, we begin with an Intellectual Property core. An IP core (or simply a *core*) refers to a hardware specification that, depending on how it is expressed, can be used to either physically manufacture an integrated circuit or configure the resources of an FPGA. For example, a pipelined multiplier could be packaged as a core. A *diffused core* in an FPGA means that the hardware is physically part of the silicon device and its functionality is realized in transistors when the device is manufactured. Because it is, by the traditional definition, truly hardware, it is more often called a *hard core*. In fact, the term hard core has come to replace diffused IP so much so that we will go forward with this text using hard core in place of diffused IP core.

In contrast, a *soft core* is implemented in the reconfigurable resources of an FPGA. Finally, some communities refer to cores as either *blocks* or *modules*. A hard block is a core that has been implemented in CMOS transistors, and a soft block is a core that has been implemented in the function generators and memories of an FPGA device. The design automation community often refers to modules, and modules are always destined to be hard blocks on an IC chip. Details regarding synthesis, mapping, and routing are

covered in detail in Chapter 2 where the inner workings of an FPGA are discussed in detail.

In summary, the Platform FPGA is simply an FPGA that an engineer uses to deploy a custom computing system. Instead of its role as a support device, it becomes the central compute device. In our case, we will be using it to develop custom computing machines for embedded systems designs. The rest of this book is spent covering details regarding Platform FPGAs and embedded systems design; this section is meant to introduce and motivate their use as an alternative to a microprocessor-based system.

Chapter in Review

The overall aim of this chapter was to set the stage for building embedded systems with Platform FPGAs. To accomplish this, we addressed three key questions.

- *What is an embedded system?*
- *Why are embedded systems different?*
- *How do Platform FPGAs help?*

We answered the first question by revisiting the concept of a computing machine and then distinguishing an embedded computing machine from a general-purpose computing machine. We answered the next question by showing the complex constraints that embedded systems impose on the developers. Finally, we gave a quick overview of the characteristics of a Platform FPGA and described how this technology can be useful to embedded systems developers.

In the gray pages that follow, we present an embedded systems scenario and contrast two approaches. We also describe the setup and installation of a popular set of commercial tools and then do a simple demonstration.

Practical Expansion

Every chapter of this book is divided into two sections: white pages describe the *science* of the chapter's topic and gray pages focus on specific *technology*. The white pages contain information that has been true for twenty years and will probably be true for the next decade. The gray pages (these pages) contain more details about state-of-the-art technology and examples introduced by the white pages. The reader might ask, "why not put it at the end of the book?" In short, the gray pages are specifically relevant to the material covered in each chapter, which covers practical exercises and examples that, if placed at the end of the book, the reader may miss. While the white pages refer to the science that is considered to be "timeless," the gray pages contain specific details on software and hardware that do age with time and, as is common with any technology, can become dated quickly. We have selected materials for these gray pages that are toward the beginning of their shelf life, but it may be evident that these technologies are less relevant as the book ages. However, the concepts of embedded systems development on an FPGA are far less vendor and hardware specific. Being too abstract would leave the reader without any solid, tangible examples, just as covering every FPGA vendor and family would result in far too much information for a reader to digest in a timely manner. Therefore, we have chosen the Xilinx ML-510 development (Xilinx, Inc., 2009c) board for its available resources that an embedded systems designer may use in their design. A photo of the ML-510 is shown in Figure 1.9. The reader is by no means required to purchase this board; there are less expensive development boards with the same FPGA available. As a result, we will do our best to present the material with an emphasis on FPGA generality and highlight what is ML-510 specific. Overall, the material we wish to cover is more specific to FPGA design and should be applicable over a range of FPGAs.

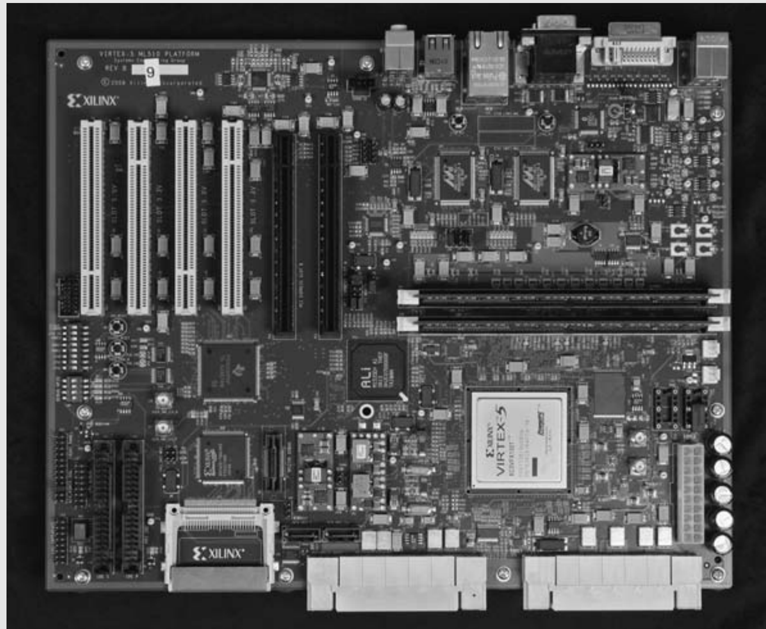


Figure 1.9. A photo of the Xilinx ML-510 development board.

Embedded Systems developers wear many hats. As Section 1.2 detailed, much more than just designing operational hardware or functionally correct software goes into a successful project. So far in the white pages we have discussed some of the benefits of Platform FPGAs, but the reader may still be asking "why use Platform FPGAs over commodity off-the-shelf components?" So let us start these gray pages with an instructive end-to-end practical example to motivate the use of Platform FPGAs. Then, we will discuss the technical details necessary to begin embedded systems design with Platform FPGAs.

1.A. Spectrometer Example

First, the general scenario and key requirements are defined. Next, two designs are described — one based strictly on commodity components and the other uses Platform FPGAs. This section concludes with a discussion of strengths and weaknesses of each approach.

1.A.1. Scenario

Suppose a hospital is preparing to conduct a large study involving a large number of experiments and thousands of volunteer subjects over a three-year period. This means every subject will make routine visits to the hospital to have various specimens collected depending on the subject. In order for the study to be conclusive, subjects have a specific time frame in which they have to report for various experiments.

Based on the large amount of data being collected and the risk of logistical mistakes corrupting the massive study, the principal investigators have budgeted money for an automated information system. Over the three years it is probable that new staff members will be hired and others will leave. As a result, the system needs to be simple enough to train new employees quickly. Also, the subject's experience needs to be pleasant and not take up much additional time. Once at the hospital it is important that subjects are not frustrated by long lines. (Every subject that fails to complete the study represents a loss of investment by the study; if enough subjects leave the study, the results of the whole study are at risk.)

Based on this, a systems analyst has designed a system where each volunteer carries an RFID tag with them while at the hospital. Many of the experiments will be custom-built so that when a subject's specimen is taken and analyzed, the subject and data are automatically associated. The results are transmitted over the campus network back to a central server. The study bears the cost of such an elaborate system because they are very interested in features that will allow them to track their subjects and eliminate paperwork transcription errors.

For our Platform FPGA case study, we will focus on one experiment within this larger context. The project requires that a specimen be taken and analyzed with a spectrograph. The embedded system needs to read the subject's RFID tag and process the spectrograph's output to determine the presence of various particular chemical elements. Together with the date and time, a database record is constructed and transmitted to the server. A minimal user interface is needed for the professional collecting the specimen and for station diagnostics. This is the minimum functionality required, but as it is anticipated that additional projects for this study are under consideration, it is worthwhile to spend some time designing the system for reuse.

1.A.2. Two Solutions

Commodity Components Solution Many of the components needed for this station are available as commodity off-the-shelf components, one easy solution is to buy a commodity desktop machine (monitor, main board, disk, keyboard/mouse, network card, off-the-shelf OS). Spectrographs come in a wide variety of packages. The one we need comes in two flavors — one is bare-bones hardware with an electrical interface and the other is hardware plus a built-in computer with

preinstalled software. The hardware itself is not very expensive, but for many stand-alone lab users, the all-in-one solution is well worth the price. Finally, a USB-based RFID wand, with ergonomic handle and software that makes recording an ID “as easy as cutting and pasting,” can be purchased.

From this information, we could buy the RFID wand, all-in-one spectrograph, and the desktop computer with a network interface. For this approach, what remains is simply a large software integration problem. The spectrograph has a graphical user interface, but processed data can be exported to an ASCII file. This file can be transmitted over a serial port to the desktop computer.

A graphical user interface on the desktop computer could have a window with database fields for the subject’s ID (which can be pasted in with RFID wand), date, time, and a button that initiates a transfer of the data file exported by the spectrograph. Our software application can then combine data and transmit it over the network. A short user’s manual can be written for the staff person, explaining how to operate the spectrograph, how to export data, how to launch the desktop application, and how to use the RFID wand to fill in the the subject ID’s field. We’ll call this the off-the-shelf solution.

Platform FPGA Solution Alternatively, we might consider using a Platform FPGA approach. For a cost-effective solution it will still be necessary to leverage off-the-shelf components; however, in this case, we will use simpler subsystems with a higher degree of integration. At the heart of the system we have a Platform FPGA that can be purchased as part of an evaluation or development board. (A custom-printed circuit board is possible, but the initial layout of the board may not be cost-effective for a few boards. We’ll explore the relationship between the number of units and the cost in the next chapter.) Interfaced to the FPGA is the bare-bones spectrometer device. Likewise, small RFID-printed circuit boards (about the size of a quarter) can also be purchased. The interface of the spectrometer is fairly complex, requiring that a sequence of digitally encoded amplitudes be correlated to known frequencies, and other electrical signals are used to control the behavior of the device. The RFID has a simple RS-232 serial interface. A case, power supply, and an inexpensive two-line LCD display round out our hardware needs.

Note that we do not need a monitor, a video adapter, or even a system disk. The programmable logic in the FPGA can be used to build a custom hardware solution that will correlate frequencies coming from the spectrometers. Likewise, three standard UART cores are easily instantiated in the FPGA’s programmable logic (one for the RFID, one for the LCD display, and one that we’ll reserve for development testing). A simple keyboard interface and a standard Ethernet network interface can be realized in programmable logic as well. Because modern FPGAs have transceivers integrated, neither an external MAC or a PHY chip are necessary — the whole network interface is on-chip. FPGA development boards typically use flash to store the FPGA configuration and most allow the flash to store the OS and application as well. With this hardware, the main tasks of the software are to (i) recognize the presence of a subject by their RFID, (ii) interact with the staff and operate the spectrometer, and (iii) communicate with the central server over the network.

1.A.3. Discussion

Both solutions are reasonable approaches and both meet the functional requirements of the problem, but which one is better? To answer this question, we will consider a number of factors, including quantitative measures (such as cost) and qualitative measures (such as how well does it serve the customer’s goals). Indeed, we’ll see that this complex evaluation is typical for embedded systems.

We begin with cost. Cost is the price of all of the hardware and intellectual property plus the development cost. In accounting the total cost, we can discount solutions that provide components that realistically can be reused in the

Commodity Costs		Platform FPGA Costs	
Desktop PC	$400	FPGA, Flash, PCB	$500
Spectrometer Unit	$2 500	Bare Bones Spectrometer	$500
RFID Wand	$200	RFID Device	$100
TOTAL	$3 100	TOTAL	$1100
(a)		(b)	

Figure 1.10. (a) All-commodity parts solution and (b) custom FPGA solution.

future. No adjustment will be made for the learning experience (i.e., the benefit of learning a new technology that will improve a future project). The hardware costs for each solution are shown in Figure 1.10. Even though commodity components are much less expensive than custom components, the all-commodity solution uses more components. Furthermore, packaging and application software associated with each subsystem contribute to the hardware cost of solution. Of course, hardware is only one part of the total cost. The software/development costs for the solutions are heavily in favor of the commodity solution. Rather than estimating an exact cost, we might simply compare ratios. Chances are that for most software/applications programmers, the time it takes to learn and debug hardware development will eat through the $2400. However, experienced computer engineers might break even if they can integrate the system in a week.

There are two other factors that may be included in the cost. First is the risk of failure. Of the two solutions, the all-commodity approach probably has a lower risk of failure, as each subsystem is already an independent entity. For example, the all-in-one spectrograph and data it exports are warrantied. The only risk is in making sure it is calibrated properly. In contrast, the hardware solution has more unknowns. Correlating frequencies in hardware is fairly straightforward in an academic setting. However, measurements from physical devices can often have “messy” data with invalid samples and skewed constants. These are normally corrected with software, but, if unanticipated, this can lead to longer development times.

Now consider scalability. With the cost of the all-commodity solution over 2.5$\times$ greater, as the number of data collection stations grows, the Platform FPGA solution becomes the clear winner. Of course, a careful reader may point out that these numbers are suspect and that a case could be made to argue that a commodity solution would be better. Our goal with this example is to show how these two technologies can both be used to implement a viable solution. A reader familiar with the commodity parts can begin to connect the dots between how a similar solution is constructed from Platform FPGAs. We can then extend the solution to highlight some of the unique features the Platform FPGA can support, such as reduced chip count and flexible scalability.

Of course this is only one specific scenario and it is impossible to make an argument that Platform FPGAs will always be the practical solution. What is important is understanding that Platform FPGAs do provide much of the same functionality as all-commodity solutions, but often at a reduced total cost. Therefore, it is important to consider the custom solution when designing systems.

1.B. Introducing the Platform FPGA Tool Chain

With the practical example complete, it is time to shift our focus away from the theory and science of Platform FPGA development and instead begin to understand the necessary technology behind developing embedded computing systems with Platform FPGAs. This section exposes the reader to a number of graphical user interfaces, as well as their corresponding

command line tools to generate complete Platform FPGA designs. The end goal of this first practical example is to build a basic "hello world" system and run it on the FPGA. It may not be technically challenging, but there is something gratifying about running a design on the hardware within the first chapter.

An important aside, this book focuses on the use of Linux for the development environment as well as the run-time environment on the Platform FPGA. Our goal is not to force the reader to switch from their current development environment, but to describe what the authors and many research institutions and companies have been or are beginning to use. (The Xilinx tools are supported under Microsoft Windows, and we highlight any applications, tools, or commands that are specific to Linux development.)

So we begin by using a graphical user interface that makes FPGA development exceedingly easy. Here we will look at Xilinx Platform Studio, (Xilinx, Inc. 2009d) (or XPS for short). Other FPGA vendors have similar tools with slightly different features and operating modes. XPS provides a multitude of wizards, tools, and premade design templates that greatly simplify the initial design process. Our focus for this technical section is to begin to familiarize the reader with the tool chain through a simple Platform FPGA design.

1.B.1. Getting Started with Xilinx Platform Studio

The first step is to acquire and install the Xilinx software package: Integrated Software Environment (ISE), (Xilinx, Inc., 2009b). The ISE install should consist of at least the ISE Design Suite, Embedded Development Kit (EDK), (Xilinx, Inc. 2009a), Software Development Kit (SDK), and ChipScope Pro. We will be focusing our attention initially on the EDK and SDK tools (which are part of the XPS tool chain). Many digital logic courses use ISE; therefore, we will spend less time discussing ISE until later chapters. ChipScope is a very useful hardware development tool that we will use in future chapters. If you are installing the tools yourself on a Linux system, the directory structure shown in Figure 1.11 is a convenient way to keep the multiple versions organized. (The directory `/opt` is often used to install packages that have a custom organization; i.e., they don't follow the standard GNU package format and filesystem hierarchy standard discussed in the next chapter.)

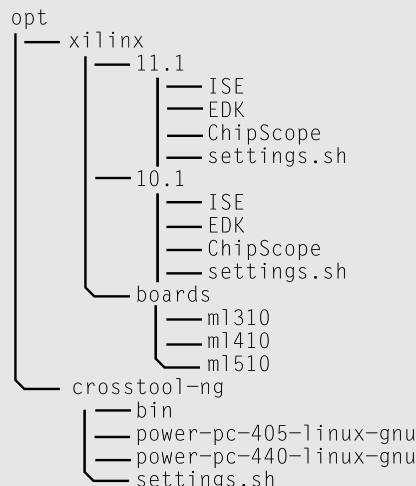


Figure 1.11. A directory structure for installing Xilinx ISE.

As can be seen, under the directory `/opt/xilinx` there is a subdirectory for each version of the tools. At the time of this writing, 11.1 is currently the latest release. (There have been minor updates available at the time of this writing, which would make this release technically 11.4, but major release version number 11 is the important identifier.) In that directory, there are two subdirectories: `ISE` and `EDK`. This is where the Integrated Software Environment and Embedded Development Kit were installed (respectively). (Other tools, such as ChipScope and PlanAhead, can also be installed here.)

Now that the install locations have been identified, the second step is to update your user environment variables. This depends on which of two families of command line shells you use. For GNU/Linux systems, most distributions default to BASH, which is part of the Bourne-shell family of command line shells (others include the original Bourne shell `sh` as well as Korn shell `ksh` and others). The second family of shells includes C shell `csh` and `tcsh`. If you are working on a Solaris Unix workstation, C shell is common.

There are several ways to determine which shell you are using, perhaps the easiest way is to type:

```
echo $SHELL
```

If the result is `/bin/csh` or `/bin/tcsh`, then you set your environment variables by typing:

```
source /opt/xilinx/11.1/ise/settings.csh
source/opt/xilinx/11.1/edk/settings.csh
```

Note that this *only* changes the current shell and will be lost when you logout.

For the rest of the book, we will use the syntax for the Bourne-shell family of command line shells and, specifically, we will assume that the reader is using BASH. (If you use C shell, you'll need to occasionally adjust the examples.) Assuming that you are using BASH, then the commands are:

```
source /opt/xilinx/11.1/edk/settings.sh
source/opt/xilinx/11.1/ise/settings.sh
```

1.B.2. Using Xilinx Platform Studio

Now we are ready to launch the graphical user interface for XPS. Type `xps` at the command line shown in Figure 1.12 (or navigate to the Xilinx Platform Studio icon in the Window's Start Menu). A brief Xilinx splash screen is displayed before the application starts and an initial dialog pops up (shown in Figure 1.13).

Base System Builder Wizard We will begin by creating a new project and using the Base System Builder (BSB) wizard to create a basic template design for the Xilinx ML-510 development board. If you have a different board that is supported by the BSB wizard, select it and then following the rest of these directions.

The BSB wizard's welcome screen (Figure 1.14) allows you to either create a new design or load an existing BSB configuration file. We will start with a new design.

```
% xps
Xilinx Platform Studio
Xilinx EDK 11.1 Build EDK_LS2.6
Copyright (c) 1995-2009 Xilinx, Inc.  All rights reserved.

Launching XPS GUI...
```

Figure 1.12. Launching XPS from the command line.

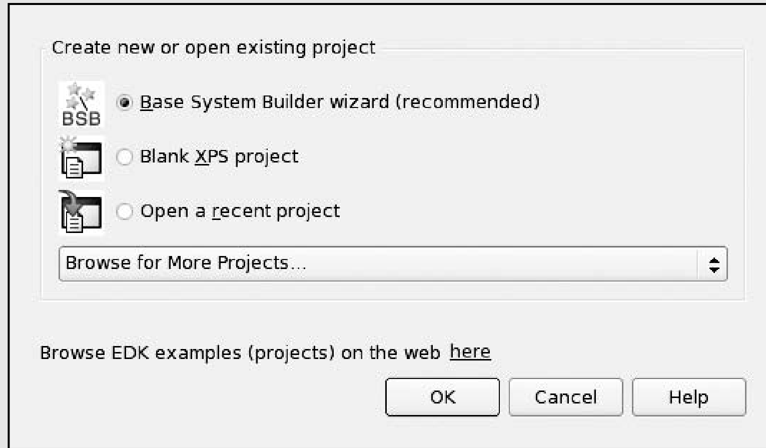


Figure 1.13. Initial XPS dialog.

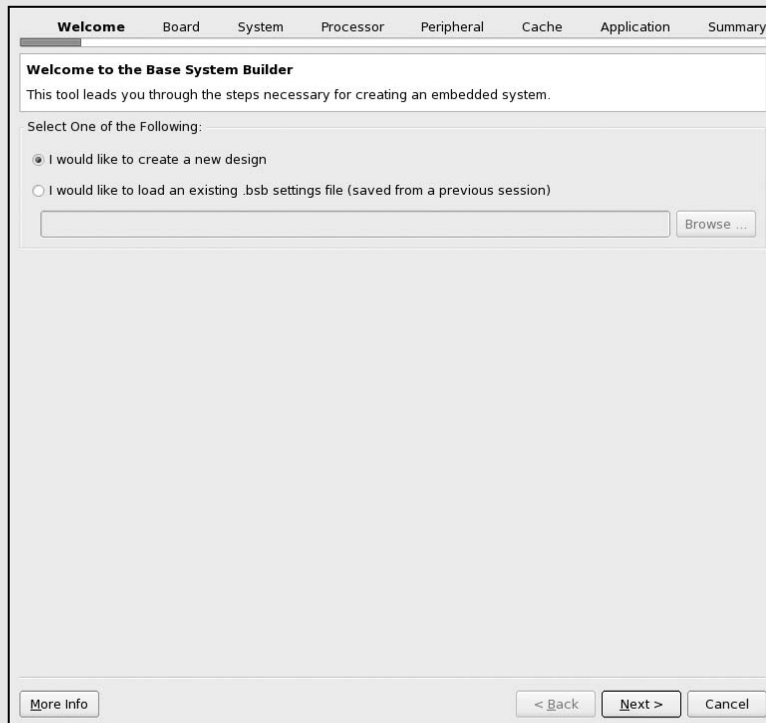


Figure 1.14. The base system builder wizard's welcome screen.

Figure 1.15. The base system builder wizard's board selection screen.

The BSB wizard's board selection screen (Figure 1.15) allows you to select which development board you will be designing for. This book uses the Xilinx Virtex 5 ML-510 Evaluation Platform.

The BSB wizard's system configuration screen (Figure 1.16) allows you to select a single-processor or dual-processor system. (Note: this option may not exist for all development boards.) For this first design we will select a single-processor system.

The BSB wizard's processor configuration screen (Figure 1.17) allows you to set the processor type and operating frequency for the processor and bus. Most of the designs presented in these gray pages focus on using the embedded PowerPC common in the Virtex series parts. The specific operating frequencies are less important for this initial design.

The BSB wizard's peripheral configuration screen (Figure 1.18) allows you to add or remove peripherals to the design. These are the basic peripherals that the particular development board supports. In this design, add only the *RS232_Uart* (xps_uartlite) and the *xps_bram_ctlr*. The UART will be used to connect the printed output from the development board to the control computer's terminal. The block RAM (BRAM) controller is on-chip memory used to store the application that will be written shortly.

The BSB wizard's cache configuration screen (Figure 1.19) allows you to add/configure the PowerPC's embedded cache. At this time we will not add cache.

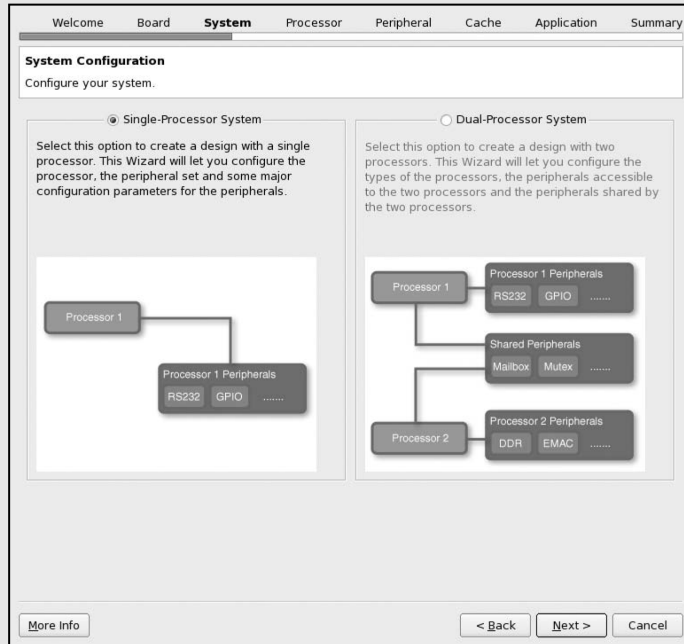


Figure 1.16. The base system builder wizard's system configuration screen.

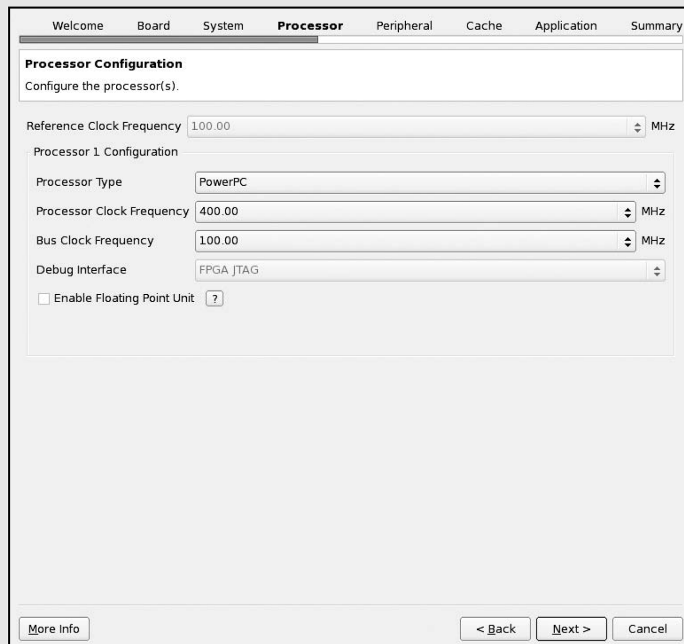


Figure 1.17. The base system builder wizard's processor configuration screen.

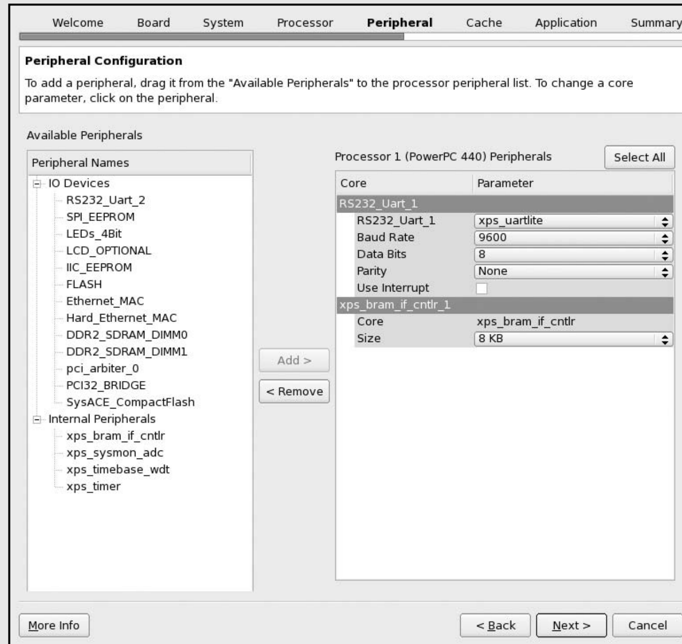


Figure 1.18. The base system builder wizard's peripheral configuration screen.

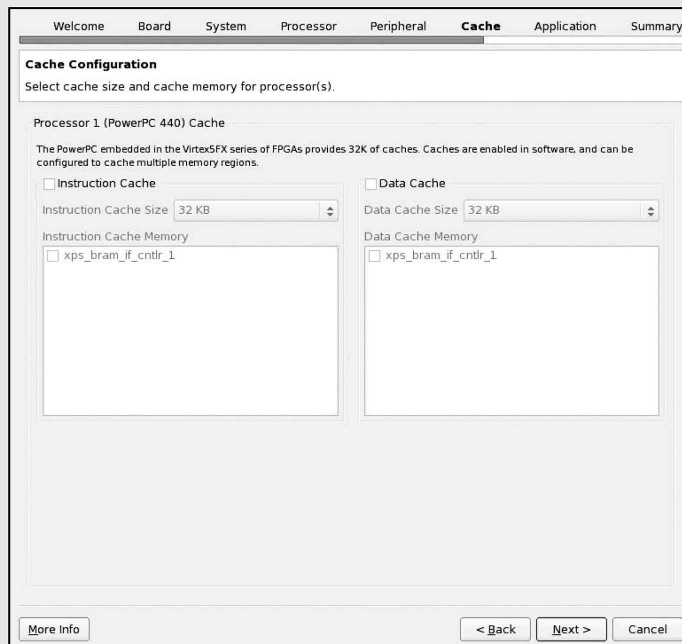


Figure 1.19. The base system builder wizard's cache configuration screen.

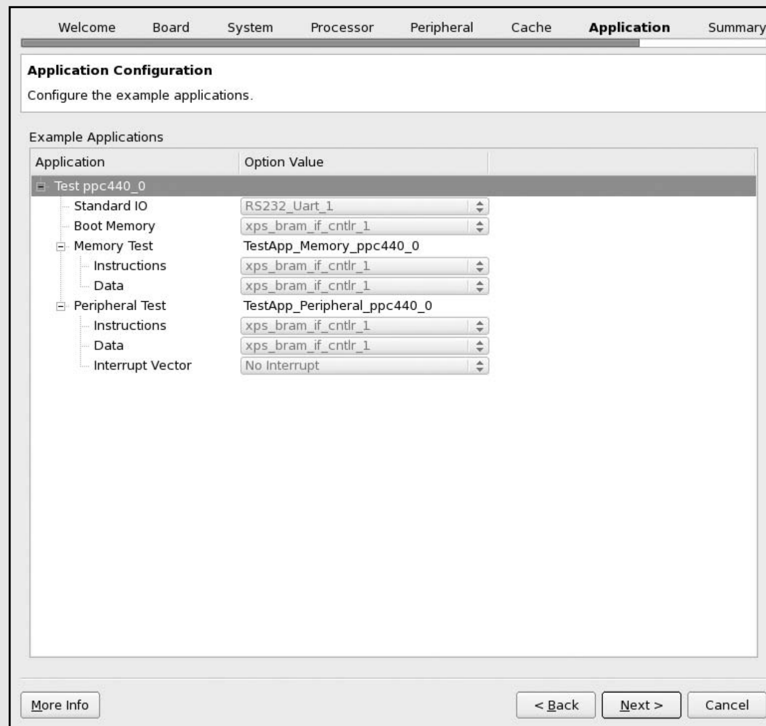


Figure 1.20. The base system builder wizard's application configuration screen.

The BSB wizard's application configuration screen (Figure 1.20) allows you to set the memory location for the Memory Test application and Peripheral Test application that are standard with the Xilinx BSB wizard. We will be designing our own application shortly so it is unnecessary to modify these configurations.

The BSB wizard's summary screen (Figure 1.21) presents a summary of the design that has been created with the wizard. It provides the designer with an opportunity to double-check that all configurations have been set correctly.

XPS Overview The BSB wizard simplifies the design process into a sequence of button clicks, but now XPS can be used to further customize the design. In this example we will hold off on the discussion of these customizations and instead focus on the XPS application. Future chapters will explore more customized designs. Details of the software and hardware flows appear in the next two chapters (respectively). It is worth noting that XPS does not do any of the actual work — it provides a graphical front end to command line tools that do the actual assembly and synthesis. Throughout the book we will cover these command line tools to provide the reader with a more thorough understanding of their role in FPGA development.

By default, the XPS GUI has four main areas, which are highlighted in Figure 1.22. The menu bar and buttons span across the top of the application. The menu system has every function that XPS can invoke, and the buttons are associated with specific menu items. Thus the buttons can be seen as shortcuts to navigating the menus. In the middle and on the left of the application screen is a tabbed project information area. This is where basic information about the current project is stored, such as options, location of various configuration files, and the IP Catalog. The IP Catalog provides a list (organized

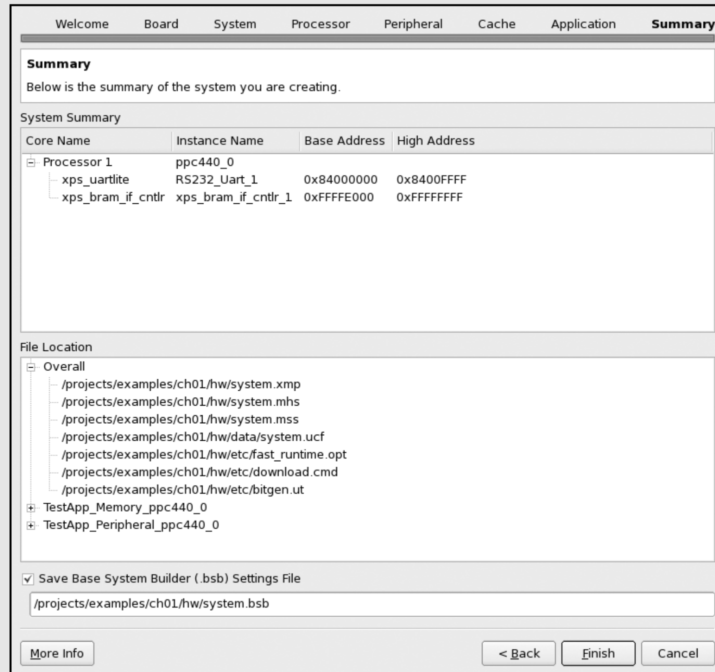


Figure 1.21. The base system builder wizard's summary screen.

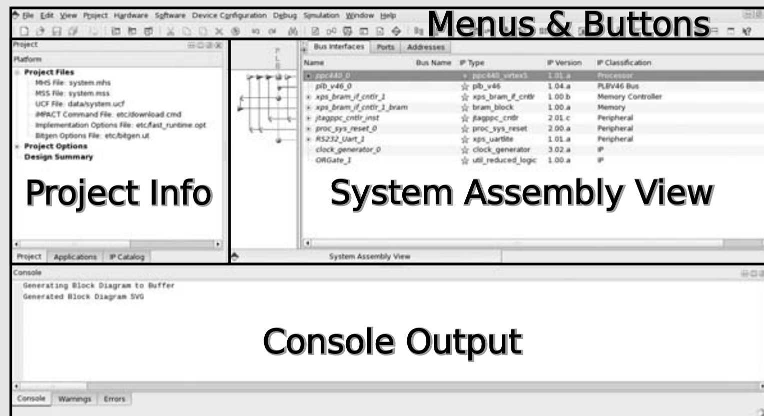


Figure 1.22. XPS menus, buttons, project information, and console areas.

in a tree) of IP cores available to the designer. By right-clicking on a leaf in the tree, one can add an instance of the IP core to their project. At the bottom of the application is the console output window. This window shows a filtered list of all output generated by the command line applications invoked by XPS. The command line tools generate a large amount of information while they process a design. One handy feature of this window is that by clicking on the "Errors" tab, only

error messages are displayed. After a long run, this is a quick way to see what, if any, error messages appeared in the copious output.

The fourth area is the system assembly window. This window allows the designer to connect and configure IP cores. For example, by connecting a DDR RAM controller to the Processor Local Bus, the designer is, in effect, putting memory on the bus. Double-clicking on an instance in the design will bring up an IP-specific configuration tool. For those familiar with VHDL already, these configuration tools allow the designer to assign values to all of the generics and parameters in the IP core.

Once all of the configuration changes have been made and the system is ready to be run through the toleration to produce a configuration file (bitstream) for the FPGA, we can go ahead and synthesize the design. This is accomplished through the `Hardware` menu. Select `Generate Bi‘tstream` to begin the synthesis process. This may take between 10 to 30 minutes depending on the speed of the computer running XPS. Once the bitstream has been generated, it is now time to switch from hardware development to software development. The hardware design must be exported to the software development kit (SDK). From the `Project` menu select `Export Hardware Design to SDK`.

Software Design Kit Overview The Xilinx Software Design Kit (SDK) is used to develop applications for the hardware specified within XPS. Xilinx has stated that beginning with the release of the 12.1 tools the SDK will replace the application's tab within the XPS. Meaning, Xilinx will no longer support software development in the hardware environment development kit. Abstracting the software development from the hardware development is important because, as mentioned earlier in this chapter, the distinction between hardware and software becomes less clear with FPGA development. For developers more familiar with the Eclipse C/C++ Development Toolkit (CDT), the SDK should be a welcome improvement for project management over the previous application process. This section aims to describe the SDK in sufficient detail to generate a stand-alone software platform and a simple “hello world” application. In future chapters, more details will be given regarding the specifics of the application development and among the stand-alone, xilkernel, and linux kernel platforms. The SDK can be launched from the command line with the `xps_sdk` command.

The SDK will prompt you to select and import a workspace from a previous design. Browse to the workspace created when you exported the design from XPS. (Note: this step can be skipped if the reader is continuing directly from the previous section.) With the SDK open, a few steps need to be taken to create a software platform and an application. We have chosen a simple stand-alone platform as we do not require any complex operating system support. All that is necessary for our application is to print “hello world!”

The first step is to create a stand-alone software platform. A software platform consists of the necessary libraries and drivers for the application to link to during compile time. The stand-alone software platform is the bottom layer of the software stack and provides a single-threaded environment with support for input and output streams. We will take advantage of the output (via `print`) support for the “hello world” example. This step can be accomplished in a variety of ways, but we will use the `File` menu option and select `new` followed by `software platform...` A new software platform project window opens, shown in Figure 1.23. Name the project “standalone_project” and set the platform type to be “standalone,” as shown in Figure 1.24.

The second step is to create an application. The application will use the stand-alone software platform. To create a new application, use the `File` menu option and select `new` followed by `Managed Make C Application Project`, as seen in Figure 1.25. Under this project, the Makefile will be updated by the SDK. This is one of the default options and requires the least amount of effort to maintain. Set the project name to “hello_world” and under `Sample Applications` select `Hello World`, shown in Figure 1.26. This will create a simple C hello world program.

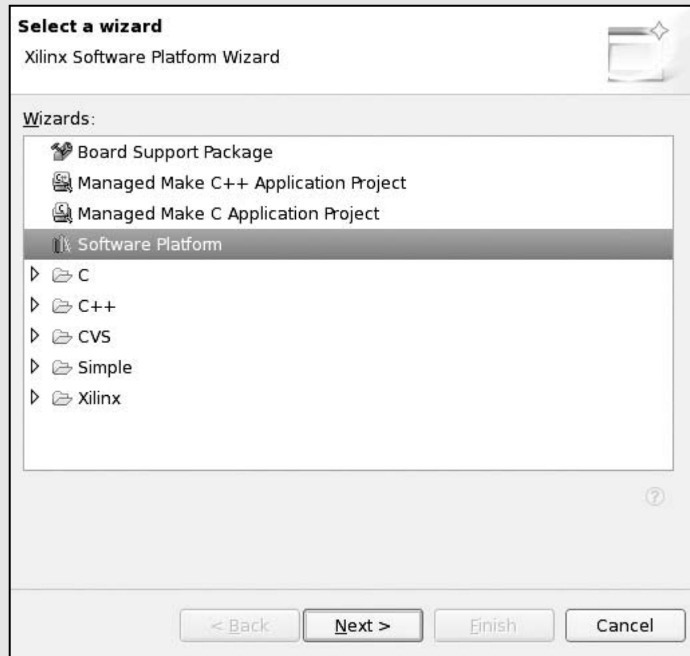


Figure 1.23. A new software platform project window.

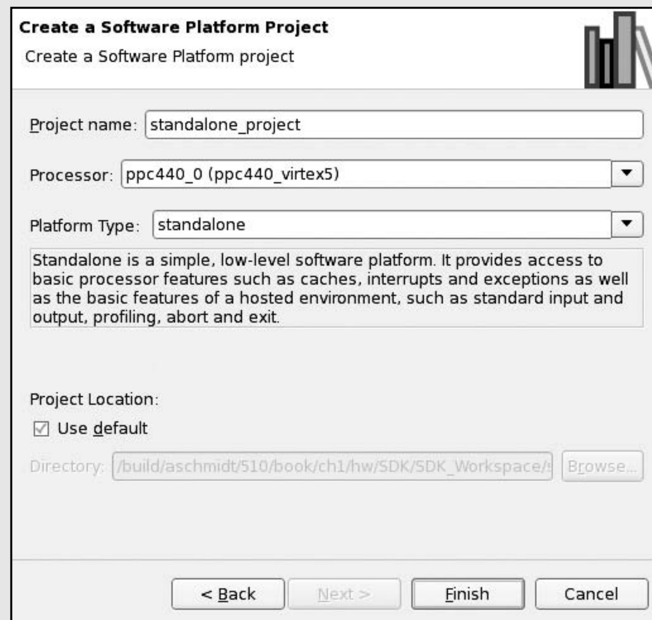


Figure 1.24. A software platform configuration window.

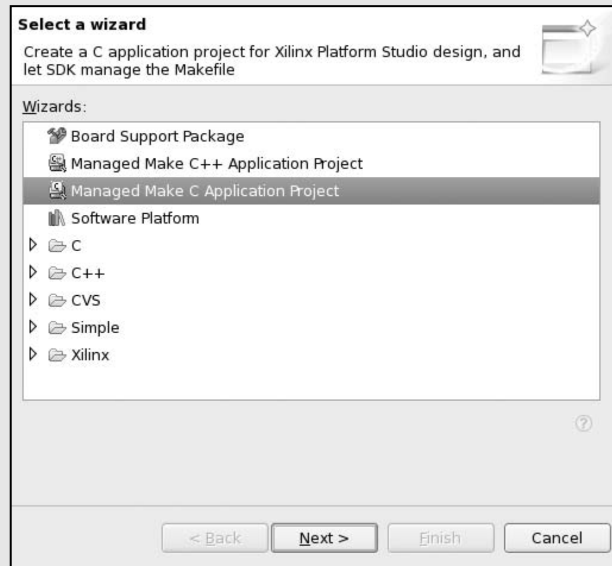


Figure 1.25. A new software application window.

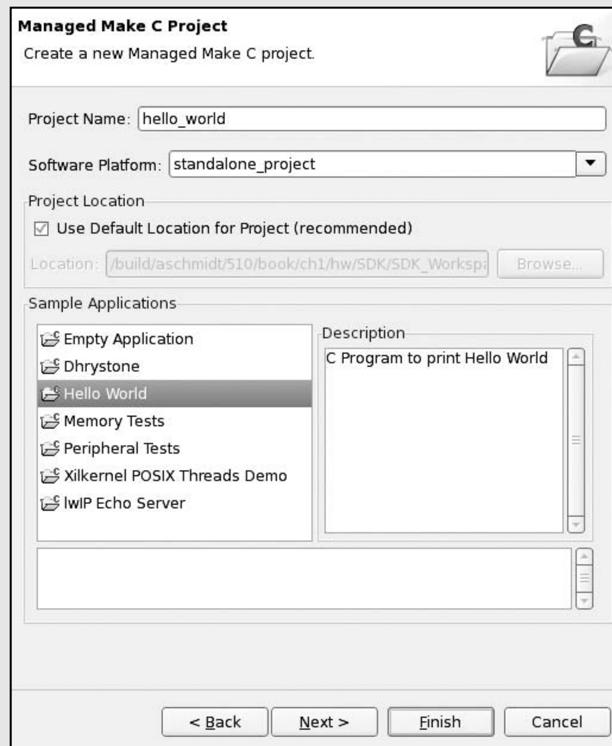


Figure 1.26. A new software configuration window.

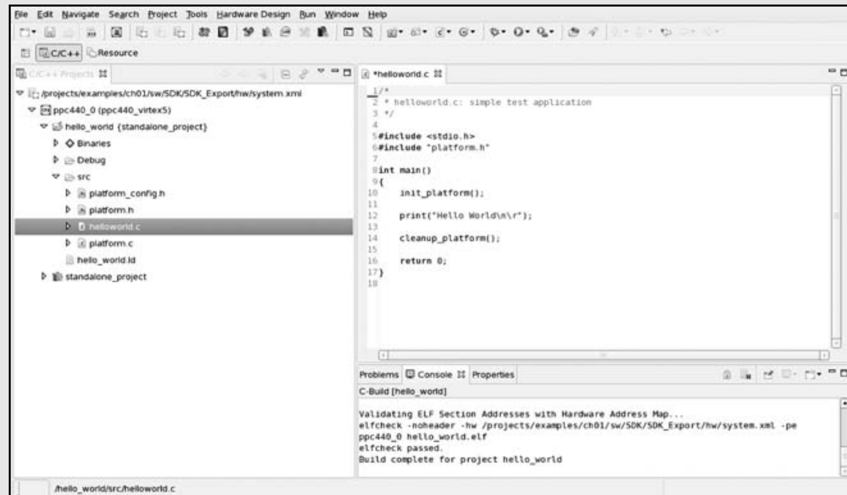


Figure 1.27. SDK GUI.

File	Location
genace.tcl	$XILINX_EDK/xmd/data/
download.bit	xps_project_directory/implementation/
hello_world.elf	sdk_workspace/hello_world/Debug/

Table 1.1 Files needed to generate ACE file.

To view the source code for the hello world application, expand the hello world arrow followed by the src arrow (as shown in Figure 1.27). Changes made to the source code will automatically be compiled when the file is saved. Alternatively, use the `Project` menu and select `Build All` to force a recompile.

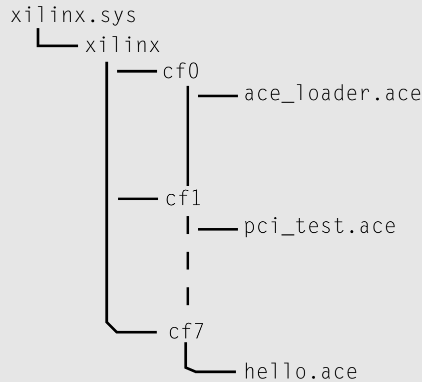
Once the application has been created and compiles without errors it is time to download hardware and software to the FPGA. This can be done in a variety of ways. The simplest is through the JTAG cable that may come with the development board (or can be purchased separately). The ML-510 development board includes a CompactFlash card to program the FPGA. We will create a Xilinx Advanced Configuration Environment (ACE) file. The ACE file is used on power-up to program the FPGA (both the hardware bitstream and application). These details are discussed in future chapters; for now we will focus on getting the design to run on the FPGA and "hello world" output.

To generate the ACE file you will need three files, located in the following directories, shown in Table 1.1.

Using the Xilinx Microprocessor Debugger (XMD), we will run the genace.tcl script to build the ACE file from the download.bit and hello_world.elf files.

```
% xmd -tcl genace.tcl -jprog -hw download.bit \
      -elf hello_world.elf -board ml510 -ace hello.ace
```

Next, to copy the `hello.ace` file to the development board, pull the CompactFlash card out of the development board (when the board is off!) and put it in any standard CompactFlash card reader connected to a computer. The directory structure is:



Each `cf`*n* directory holds a configuration, and the development board comes with a Compact Flash with slots 6 and 7 free for user designs. Simply remove any ACE file in the cf7 directory and copy the `hello.ace` into that directory on the CompactFlash. Eject/unmount the CompactFlash card, and return the card to the development board.

Finally, connect the development board to a computer over an RS-232 serial line. On Linux machines, many people use either `minicom` or `kermit` as a terminal emulator. Be sure to set the proper serial line and parameters (9600 baud, 8 bits, no parity, 1 stop bit). Start up the terminal emulator, turn on the development board, and, after the power-on self-test, type '7' (to boot the configuration in slot 7). The results are shown in Figure 1.28.

With the "Hello World" application running on the board, you have successfully completed your first Platform FPGA project! We will rely on some of this basic knowledge going forward when we begin to assemble more complex hardware cores and base systems. At this point, we recommend spending some time exploring these tools and becoming familiar with many of their options and functions as possible.

```
ML510 ACE-loader
----------------
 Enter Desired System ACE CF Configuration <0-7>.
   0: ACE-loader.
   1: Configuration 1.
   2: Configuration 2
   3: Configuration 3
   4: Configuration 4
   5: Configuration 5
   6: Configuration 6
   7: Configuration 7
Select <0-7>: Rebooting to System ACE Configuration Address 7...
Hello World!
```

Figure 1.28. Output displayed on terminal for the HelloWorld example.

Exercises

P1.1. Name three consumer electronics products that have embedded systems. Include a justification.

P1.2. Name three consumer electronics products that do not contain embedded computer systems. (You may need to consider very old products or products that are not consumer electronics.)

P1.3. In addition to the explicit inputs and outputs of a computing system, what is implicitly consumed and produced by (electronic) computing machines?

P1.4. How is the specification phase of a project different from the design phase?

P1.5. What is the difference between design and integration in the waterfall model?

P1.6. Is “access to a wireless network” a capability or a performance metric?

P1.7. Suppose two devices both have wireless access, but one is faster? Is this a new capability?

P1.8. Device A uses a network standard that is 100 Mb/s and device B uses a newer, revised standard that is 100 or 1000 Mb/s. Is this a capability or a functional improvement?

P1.9. Consider two solutions. One uses a 4-bit processor with a clock frequency of 1000 MHz; the other is a 64-bit processor with a clock rate of 100 MHz. Both are executing an infinite loop of 64-bit ADD operations. Contrast the two solutions in terms of:

- latency
- results/second
- energy
- development cost

P1.10. What is the advantage of using a standard — such as ASCII, binary, and BCD — versus an application-specific format?

P1.11. What is the disadvantage of using an application-specific format versus using a standard such as an IEEE 754 floating-point format?

P1.12. Suppose the dynamic range of an instrument flying on a satellite is -192 to $+191$ in discretized (integer) steps. How many bits are needed to represent a sample from the instrument?

P1.13. Assuming the same instrument just given: If a 32-bit processor (with a 100-MHz 32-bit system bus) was used to read data, one sample at a time, what is the peak

theoretical bandwidth of the information moved from the instrument to the processor?

P1.14. What is the bandwidth if multiple samples were buffered on the instrument and multiple samples could be read each time?

P1.15. What is the bandwidth if the bus was 64 bits?

P1.16. Suppose the NRE on the fabrication of an ASIC is $150,000 and the unit cost is $0.15. The unit cost of an FPGA is $15. All other costs going into the product are equal. If the anticipated demand is expected to be X units the first year and will decay by 50% each subsequent year ($X/2$ units the second, $X/4$ units the third, and so on), how many units have to be sold the first year for the ASIC to be more profitable than the FPGA?

P1.17. Explain the difference between hardware and software in terms of a traditional processor-based system.

P1.18. How does the "hardware" of a Platform FPGA-based system differ from the hardware of a traditional processor-based system?

P1.19. What is the difference between a soft IP core and a hard IP core in a Platform FPGA system?

References

Berger, A. (2002). *Embedded systems design: An introduction to processes, tools, and techniques*. San Francisco, CA, USA: CMP Books.

Catsoulis, J. (2003). *Designing embedded hardware*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Hollabaugh, C. (2003). *Embedded Linux: Hardware, software, and interfacing*. Boston, MA, USA: Addison/Wesley Publishing.

Vahid, F., & Givargis, T. (2002). *Embedded systems design: A unified hardware/software introduction*. New York, NY, USA: John Wiley & Sons, Inc.

Wolf, W. (2001). *Computers as components: Principles of embedded computing system design*. San Francisco, CA, USA: Morgan Kaufmann Publisher.

Xilinx, Inc. (2009a December). *EDK concepts, tools, and techniques: A hands-on guide to effective embedded system design*.

Xilinx, Inc. (2009b June). *ISE in-depth tutorial (UG695) v11.2*.

Xilinx, Inc. (2009c June). *ML510 reference design user guide (UG355) v1.2*.

Xilinx, Inc. (2009d June). *Platform specification format reference manual (UG642) v11.2*.

Yaghmour, K. (2003). *Building embedded Linux systems*. Sebastopol, CA, USA: O'Reilly & Associated, Inc.