How Programmable Logic Works

by Michael Barr

In recent years, the line between hardware and software has blurred. Hardware engineers create the bulk of their new digital circuitry in programming languages such as VHDL and Verilog and often target it to CPLDs and FPGAs. What are these devices and how are they changing the way embedded systems are designed? This article will help you make sense of programmable logic.

A quiet revolution is taking place. Over the past few years, the density of the average programmable logic device has begun to skyrocket. The maximum number of gates in an FPGA is currently around 500,000 and doubling every 18 months. Meanwhile, the price of these chips is dropping. What all of this means is that the price of an individual NAND or NOR is rapidly approaching zero! And the designers of embedded systems are taking note. Some system designers are buying processor cores and incorporating them into system-on-a-chip designs; others are eliminating the processor and software altogether, choosing an alternative hardware-only design.

As this trend continues, it becomes more difficult to separate hardware from software. After all, both hardware and software designers are now describing logic in high-level terms, albeit in different languages, and downloading the compiled result to a piece of silicon. Surely no one would claim that language choice alone marks a real distinction between the two fields. Turing's notion of machine-level equivalence and the existence of language-to-language translators have long ago taught us all that that kind of reasoning is foolish. There are even now products that allow designers to create their hardware designs in traditional programming languages like C. So language differences alone are not enough of a distinction.

Both hardware and software designs are compiled from a human-readable form into a machine-readable one. And both designs are ultimately loaded into some piece of silicon. Does it matter that one chip is a memory device and the other a piece of programmable logic? If not, how else can we distinguish hardware from software?

I'm not convinced that an unambiguous distinction between hardware and software can ever be found, but I don't think that matters all that much. Regardless of where the line is drawn, there will continue to be engineers like you and me who cross the boundary in our work. So rather than try to nail down a precise boundary between hardware and software design, we must assume that there will be overlap in the two fields. And we must all learn about new things. Hardware designers must learn how to write better programs, and software developers must learn how to utilize programmable logic.

Types of programmable logic

Many types of programmable logic are available. The current range of offerings includes everything from small devices capable of implementing only a handful of logic equations to huge FPGAs that can hold an entire processor core (plus peripherals!). In addition to this incredible difference in size there is also much variation in architecture. In this section, I'll introduce you to the most common types of programmable logic and highlight the most important features of each type.

PLDs

At the low end of the spectrum are the original Programmable Logic Devices (PLDs). These were the first chips that could be used to implement a flexible digital logic design in hardware. In other words, you could remove a couple of the 7400-series TTL parts (ANDs, ORs, and NOTs) from your board and replace them with a single PLD. Other names you might encounter for this class of device are Programmable Logic Array (PLA), Programmable Array Logic (PAL), and Generic Array Logic (GAL).

PLDs are often used for address decoding, where they have several clear advantages over the 7400-series TTL parts that they replaced. First, of course, is that one chip requires less board area, power, and wiring than several do. Another advantage is that the design inside the chip is flexible, so a change in the logic doesn't require any rewiring of the board. Rather, the decoding logic can be altered by simply replacing that one PLD with another part that has been programmed with the new design.

Inside each PLD is a set of fully connected macrocells. These macrocells are typically comprised of some amount of combinatorial logic (AND and OR gates, for example) and a flip-flop. In other words, a small Boolean logic equation can be built within each macrocell. This equation will combine the state of some number of binary inputs into a binary output and, if necessary, store that output in the flip-flop until the next clock edge. Of course, the particulars of the available logic gates and flip-flops are specific to each manufacturer and product family. But the general idea is always the same.

Because these chips are pretty small, they don't have much relevance to the remainder of this discussion. But you do need to understand the origin of programmable logic chips before we can go on to talk about the larger devices. Hardware designs for these simple PLDs are generally written in languages like ABEL or PALASM (the hardware equivalents of assembly) or drawn with the help of a schematic capture tool.

CPLDs

As chip densities increased, it was natural for the PLD manufacturers to evolve their products into larger (logically, but not necessarily physically) parts called Complex Programmable Logic Devices (CPLDs). For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip. The larger size of a CPLD allows you to implement either more logic equations or a more complicated design. In fact, these chips are large enough to replace dozens of those pesky 7400-series parts.

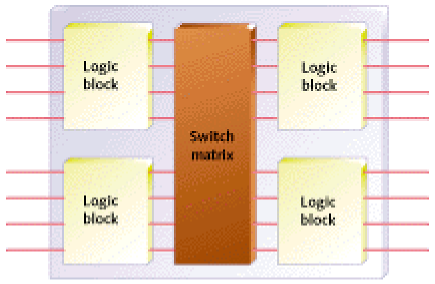

Figure 1. Internal structure of a CPLD

Figure 1 contains a block diagram of a hypothetical CPLD. Each of the four logic blocks shown there is the equivalent of one PLD. However, in an actual CPLD there may be more (or less) than four logic blocks. I've just drawn it that way for simplicity. Note also that these logic blocks are themselves comprised of macrocells and interconnect wiring, just like an ordinary PLD.

Unlike the programmable interconnect within a PLD, the switch matrix within a CPLD may or may not be fully connected. In other words, some of the theoretically possible connections between logic block outputs and inputs may not actually be supported within a given CPLD. The effect of this is most often to make 100% utilization of the macrocells very difficult to achieve. Some hardware designs simply won't fit within a given CPLD, even though there are sufficient logic gates and flip-flops available.

Because CPLDs can hold larger designs than PLDs, their potential uses are more varied. They are still sometimes used for simple applications like address decoding, but more often contain high-performance control-logic or complex finite state machines. At the high-end (in terms of numbers of gates), there is also a lot of overlap in potential applications with FPGAs. Traditionally, CPLDs have been chosen over FPGAs whenever high-performance logic is required. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter.

FPGAs

Field Programmable Gate Arrays (FPGAs) can be used to implement just about any hardware design. One common use is to prototype a lump of hardware that will eventually find its way into an ASIC. However, there is nothing to say that the FPGA can't remain in the final product. Whether or not it does will depend on the relative weights of development cost and production cost for a particular project. (It costs significantly more to develop an ASIC, but the cost per chip may be lower in the long run. The cost tradeoff involves expected number of chips to be produced and the expected likelihood of hardware bugs and/or changes. This makes for a rather complicated cost analysis, to say the least.)

The development of the FPGA was distinct from the PLD/CPLD evolution just described. This is apparent when you look at the structures inside. Figure 2 illustrates a typical FPGA architecture. There are three key parts of its structure: logic blocks, interconnect, and I/O blocks. The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable input, output, or bi-directional access to one of the general-purpose I/O pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of logic blocks. And connecting logic blocks to logic blocks and I/O blocks to logic blocks is the programmable interconnect wiring.

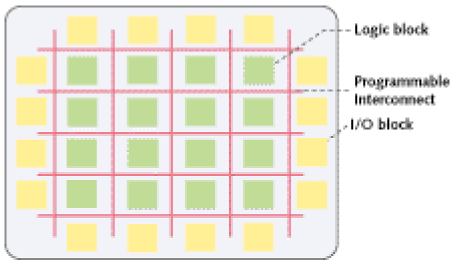

Figure 2. Internal structure of an FPGA

The logic blocks within an FPGA can be as small and simple as the macrocells in a PLD (a so-called fine-grained architecture) or larger and more complex (coarse-grained). However, they are never as large as an entire PLD, as the logic blocks of a CPLD are. Remember that the logic blocks of a CPLD contain multiple macrocells. But the logic blocks in an FPGA are generally nothing more than a couple of logic gates or a look-up table and a flip-flop.

Because of all the extra flip-flops, the architecture of an FPGA is much more flexible than that of a CPLD. This makes FPGAs better in register-heavy and pipelined applications. They are also often used in place of a processor-plus-software solution, particularly where the processing of input data streams must be performed at a very fast pace. In addition, FPGAs are usually denser (more gates in a given area) and cost less than their CPLD cousins, so they are the de facto choice for larger logic designs.

Hardware design and development

The process of creating digital logic is not unlike the embedded software development process you're already familiar with. A description of the hardware's structure and behavior is written in a high-level hardware description language (usually VHDL or Verilog) and that code is then compiled and downloaded prior to execution. Of course, schematic capture is also an option for design entry, but it has become less popular as designs have become more complex and the language-based tools have improved. The overall process of hardware development for programmable logic is shown in Figure 3 and described in the paragraphs that follow.

Perhaps the most striking difference between hardware and software design is the way a developer must think about the problem. Software developers tend to think sequentially, even when they are developing a multithreaded application. The lines of source code that they write are always executed in that order, at least within a given thread. If there is an operating system it is used to create the appearance of parallelism, but there is still just one execution engine. During design entry, hardware designers must think-and program-in parallel. All of the input signals are processed in parallel, as they travel through a set of execution engines-each one a series of macrocells and interconnections-toward their destination output signals. Therefore, the statements of a hardware description language create structures, all of which are "executed" at the very same time. (Note, however, that the transference from macrocell to macrocell is usually synchronized to some other signal, like a clock.)

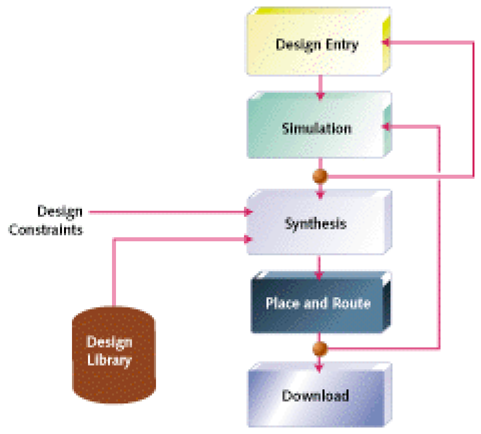

Figure 3. Programmable logic design process

Typically, the design entry step is followed or interspersed with periods of functional simulation. That's where a simulator is used to execute the design and confirm that the correct outputs are produced for a given set of test inputs. Although problems with the size or timing of the hardware may still crop up later, the designer can at least be sure that his logic is functionally correct before going on to the next stage of development.

Compilation only begins after a functionally correct representation of the hardware exists. This hardware compilation consists of two distinct steps. First, an intermediate representation of the hardware design is produced. This step is called synthesis and the result is a representation called a netlist. The netlist is device independent, so its contents do not depend on the particulars of the FPGA or CPLD; it is usually stored in a standard format called the Electronic Design Interchange Format (EDIF).

The second step in the translation process is called place & route. This step involves mapping the logical structures described in the netlist onto actual macrocells, interconnections, and input and output pins. This process is similar to the equivalent step in the development of a printed circuit board, and it may likewise allow for either automatic or manual layout optimizations. The result of the place & route process is a bitstream. This name is used generically, despite the fact that each CPLD or FPGA (or family) has its own, usually proprietary, bitstream format. Suffice it to say that the bitstream is the binary data that must be loaded into the FPGA or CPLD to cause that chip to execute a particular hardware design.

Increasingly there are also debuggers available that at least allow for single-stepping the hardware design as it executes in the programmable logic device. But those only complement a simulation environment that is able to use some of the information generated during the place & route step to provide gate-level simulation. Obviously, this type of integration of device-specific information into a generic simulator requires a good working relationship between the chip and simulation tool vendors.

Device programming

Once you've created a bitstream for a particular FPGA or CPLD, you'll need to somehow download it to the device. The details of this process are dependent upon the chip's underlying process technology. Programmable logic devices are like non-volatile memories in that there are multiple underlying technologies. In fact, exactly the same set of names is used: PROM (for one-time programmables), EPROM, EEPROM, and Flash.

Just like their memory counterparts, PROM and EPROM-based logic devices can only be programmed with the help of a separate piece of lab equipment called a device programmer. On the other hand, many of the devices based on EEPROM or Flash technology are in-circuit programmable. In other words, the additional circuitry that's required to perform device (re)programming is provided within the FPGA or CPLD silicon as well. This makes it possible to erase and reprogram the device internals via a JTAG interface or from an on-board embedded processor. (Note, however, that because this additional circuitry takes up space and increases overall chip costs, a few of the programmable logic devices based on EEPROM or Flash still require insertion into a device programmer.)

In addition to non-volatile technologies, there are also programmable logic devices based on SRAM technology. In such cases, the contents of the device are volatile. This has both advantages and disadvantages. The obvious disadvantage is that the internal logic must be reloaded after every system or chip reset. That means you'll need an additional memory chip of some sort in which to hold the bitstream. But it also means that the contents of the logic device can be manipulated on-the-fly. In fact, you could imagine a scenario in which the actual bitstream is reloaded from a remote source (via a network of some sort?), so that the hardware design could be upgraded as easily as software.

Applications

Now that you understand the technology, you're probably wondering what all of these FPGAs and CPLDs are doing within the embedded systems. However, their uses are so varied that it's impossible to generalize. Rather, I'll just touch on some of the emerging trends. This should hopefully answer your question, though admittedly indirectly.

Prototyping

Many times a CPLD or FPGA will be used in a prototype system. A small device may be present to allow the designers to change a board's glue logic more easily during product development and testing. Or a large device may be included to allow prototyping of a system-on-a-chip design that will eventually find its way into an ASIC. Either way, the basic idea is the same: allow the hardware to be flexible during product development. When the product is ready to ship in large quantities, the programmable device will be replaced with a less expensive, though functionally equivalent, hard-wired alternative.

Embedded cores

More and more vendors are selling or giving away their processors and peripherals in a form that is ready to be integrated into a programmable logic-based design. They either recognize the potential for growth in the system-on-a-chip area and want a piece of the royalties or want to promote the use of their particular FPGA or CPLD by providing libraries of ready-to-use building blocks. Either way, you will gain with lower system costs and faster time-to-market. Why develop your own hardware when you can buy an equivalent piece of virtual silicon?

The Intellectual Property (IP) market is growing rapidly. It's common to find microprocessors and microcontrollers for sale in this form, as well as complex peripherals like PCI controllers. Many of the IP cores are even configurable. Would you like a 16-bit bus or a 32-bit bus? Do you need the floating-point portion of the processor? And, of course, you'll find all of the usual supporting cast of simple peripherals like serial controllers and timer/counter units are available as well.

Hybrid chips

There's also been some movement in the direction of hybrid chips, which combine a dedicated processor core with an area of programmable logic. The vendors of hybrid chips are betting that a processor core embedded within a programmable logic device will require far too many gates for typical applications. So they've created hybrid chips that are part fixed logic and part programmable logic. The fixed logic contains a fully functional processor and perhaps even some on-chip memory. This part of the chip also interfaces to dedicated address and data bus pins on the outside of the chip. Application-specific peripherals can be inserted into the programmable logic portion of the chip, either from a library of IP cores or the customer's own designs.

Reconfigurable computing

As mentioned earlier, an SRAM-based programmable device can have its internal design altered on-the-fly. This practice is known as reconfigurable computing. Though originally proposed in the late 1960's by a researcher at UCLA, this is still a relatively new field of study. The decades-long delay had mostly to do with a lack of acceptable reconfigurable hardware. On-the-fly reprogrammable logic chips have only recently reached gate densities making them suitable for anything more than academic research. But the future of reconfigurable computing is bright and it is already finding a niche in high-end communications, military, and intelligence applications.

Gate Count
The gate count by itself is almost useless. Different vendors use different measures: number of available gates, equivalent number of NAND gates, equivalent number of gates in a PLD, equivalent number of gates in an ASIC, etc. You simply can't compare these numbers across vendors. A better comparison can be made in terms of numbers of registers (flip-flops) and I/O pins.
Number of I/O Pins
Are there adequate inputs and outputs for your design? This is often a more limiting constraint than gate count, and it very much affects the cost of the chip. As a result, many manufacturers offer the same part with different numbers of I/O pins.
Cost per Chip
Obviously, cost is a factor if you'll be including a CPLD or FPGA in your production system. Would it be cheaper in the long run to develop a fixed ASIC design and produce a large quantity of them? If you stick with the programmable device, you'll want to use the smallest part with adequate resources for your design.
Available Tools
The most popular Verilog and VHDL simulation and synthesis tools are sold by third party tool vendors. These tools generally have support for a laundry list of common FPGAs and CPLDs. This means that the tools understand the constraints of your particular chip and also understand the timing-relating information that comes out of the place and route tool.
Performance
Generally speaking, CPLDs are faster and introduce more predictable delays than FPGAs. However, that's because their internal structure is less flexible. So you have to give something up for the extra speed. What's typically lost is density. The larger your design, the more likely it is that you'll have to use a slower part. When using an FPGA, the actual performance of your design won't really be known until the final place and

route process is complete, since the routing specifics will play a role.
Power Consumption
Power consumption can be an important consideration in any system. EEPROM and Flash-based devices usually require more power than those based on PROM, EPROM, or SRAM technologies.
Packaging
Programmable logic devices are available in all sorts of packages. Your choice of a package will most likely be driven by your need to reduce power consumption, heat dissipation, size, and/or cost.

Table 1. Checklist for programmable logic selection

What's it to ya?

Hopefully, you now have a better understanding of this new kind of software that is really hardware in disguise. (Or is it a new kind of hardware that is really software in disguise?) This should give you a better basis for communicating with hardware designers on partitioning issues like: What functions on your next project should be implemented in dedicated logic, programmable logic, and/or software? I've found that there are valid reasons for choosing all three of these implementation techniques, and that you must pay close attention to the requirements of the particular application. As software and hardware continue to simultaneously expand and overlap, we must all broaden our perspectives and be willing to learn new things.

This article was published in the June 1999 issue of *Embedded Systems Programming*. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Barr, Michael. "Programmable Logic: What's it to Ya?," Embedded Systems Programming, June 1999, pp. 75-84.