ED KLINGMAN

FPGA Programming Step by Step

FPGAs and microprocessors are more similar than you may think. Here's a primer on how to program an FPGA and some reasons why you'd want to.

Small processors are, by far, the largest selling class of computers and form the basis of many embedded systems. The first single-chip microprocessors contained approximately 10,000 gates of logic and 10,000 bits of memory. Today, field programmable gate arrays (FPGAs) provide single chips approaching 10 million gates of logic and 10 million bits of memory as shown in Figure 1.

Powerful tools exist to program these powerful chips. Unlike microprocessors, not only the memory bits, but also the logical gates are under your control as the programmer. This article will show the programming process used for FPGA design.

As an embedded systems programmer, you're aware of the development processes used with microprocessors. The development process for FPGAs is similar enough that you'll have no problem understanding it but sufficiently different that you'll have to think differently to use it well. We'll use the similarities to understand the basics, then discuss the differences and how to think about them.

Similarities

Table 1 on the next page shows the steps involved in designing embedded systems with a microprocessor and an FPGA. This side-by-side comparison lets you quickly assess the two processes and see how similar they are.

Powerful tools exist to program these powerful chips. Unlike microprocessors, not only the memory bits, but also the logical gates are under your control as the programmer.

Architectural design

The architectural-design phase is surprisingly similar. Although people will argue design philosophies, it's not unusual to perform a "first cut" at programming in a pseudo-language that can be translated into and refined as a specific language, say assembly, C++, or JAVA. I describe my first FPGA architectural design in a pseudo-C language then translate it to Verilog for an FPGA. Thus the ability to express yourself in C is a good start in learning Verilog. Architectural issues could fill a book; therefore we'll focus on development issues.

Because you understand editing, compiling, assembling, linking, and loading in microprocessor programming, you can relate this to editing, compiling, synthesizing, placing, routing, and loading in FPGA programming.

Editing

Not only is Verilog syntax C-like, but, since it's 100% ASCII, you can use any editor to prepare fpga.v files. Nothing new here.

Compiling

The process of compiling a program for the microprocessor combines the edited files and builds a logically correct sequence of bits that are used to control the sequencing of logical gates. These gates write data onto buses, into latches and registers, out ports, and across channels. The gates have fixed relationships designed to accomplish fixed functions. The assembly-language instructions represent these functions. Thus microprocessor compilers either produce assembly-language programs that are then assembled into bit patterns or directly produce the bits to drive the gates and fill the registers and the memories.

FIGURE 1 Comparison of first microprocessors to current FPGAs

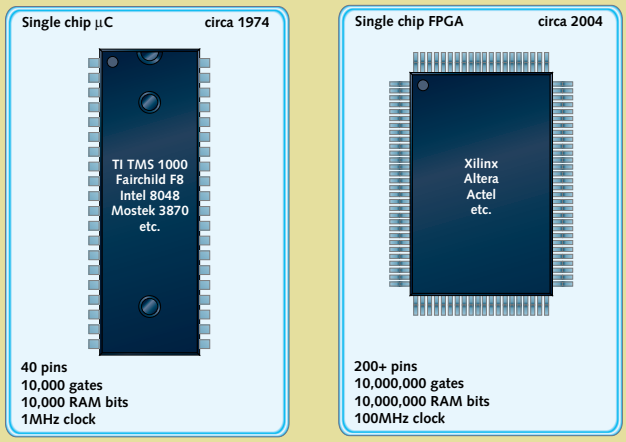


TABLE 1 Step-by-step design process for microprocessors and FPGAs

Microprocessor	FPGA
Architectural design	Architectural design
Choice of language (C, JAVA)	Choice of language (Verilog, VHDL)
Editing programs	Editing programs
Compiling programs (.DLL, .OBJ)	Compiling programs
	Synthesizing programs (.EDIF)
Linking programs (.EXE)	Placing and routing programs (.VO, .SDF, .TTF)
Loading programs to ROM	Loading programs to FPGA
Debugging µP programs	Debugging FPGA programs
Documenting programs	Documenting programs
Delivering programs	Delivering programs

The analogous operation in FPGA programming is the compilation of Verilog into register transfer logic (RTL) netlists. As the name implies, data is transferred into registers, subject to some clocking condition. At this stage FPGA programming departs from microprocessor programming in that an additional synthesis process is required to produce bits (or intermediate objects that can be converted to bits) that will control gates and fill registers and memories on an FPGA. This level is called *gate-level logic*, since it describes the logical gates of which the system will be composed. The output format is typically an Electronic Design Interchange Format (EDIF) file.

There's a large difference between compiling and synthesizing, and you have to stretch some to encompass it. Whereas the compiler produces bits to control fixed-gate patterns (the micro-

The synthesis process . . . produces bit patterns, in an intermediate format. We compile Verilog to RTL netlists, then synthesize Verilog to EDIF, then place and route EDIF to produce HEX or TTF files that can be loaded into an FPGA.

History of Linking

Early computers had a "patch board" that looked somewhat like the telephone patch boards of the 1940s where "patch cords" were plugged into "sockets" to make connections between various buses and register inputs and outputs.

The patch boards of the '40s and '50s evolved into the bit-slice microprogramming of the 1970s, where, again, the focus was on control of logical gates, and the patch cords were implemented as "wire wrap" connections.

In FPGAs the patch cords are routing connections between gates.

Early FPGAs contained gates that numbered between the hundreds and the thousands. The size was not sufficient to implement systems, but was capable of implementing the types of circuitry built with Medium Scale Integrated Circuits (dozens of gates), which implemented the "glue logic" that held microprocessors and sophisticated peripheral chips together. Therefore, the first practical applications of FPGAs consisted of replacing many glue logic chips with one FPGA. Because these circuits were already completely described by circuit diagrams, the first FPGA editors used graphics capture; that is, the circuit diagrams were redrawn in the editor, and this was the input information. No high-level languages were used to program the FPGA in this process, simply graphics editing and time-based simulation of waveforms.

processor decoders, registers, arithmetic logic unit, and so on) the synthesizer defines gate patterns described by the logic of the program. That is, your program logic gets synthesized, or mapped into, logical gates, not into processor instructions that control multigate structures. This is absolutely amazing, and good FPGA programmers give thanks every day for living in the rare time in history (post 1990+) when you can design architectures with words and then synthesize your logic into (mostly silicon) gates that execute your logic. Not to get carried away, but it's absolutely wonderful.

Linking

Linking was a latecomer to programming—maybe 1950. Previous computers and programs simply put bits into console switches, and thereby into registers. (Read about the development of linking in the sidebar.)

The bit-based outputs of the microprocessor compilation process typically don't directly control gates but must be connected to other bit patterns. This is true because most programs run under the control of an operating system and must be connected, or linked, to the operating system. In fact, the location in memory of the actual compiled bits is usually unknown and not determined until linking and loading is completed. Further, there may be programs existing in a library that must also be linked to the compiled program before a useful product exists.

The synthesis process, as we have discussed, produces bit patterns, in an intermediate format. We compile Verilog to RTL netlists, then synthesize Verilog to EDIF, then place and route EDIF to produce HEX or TTF files that can be loaded into an FPGA. These bit patterns will end up controlling logic gates and filling memory and registers.

In the same way that C and other programs include objects defined in (possibly third-party) libraries, FPGA programs can include or import portions of systems from third-party intellectual property, in the form of FPGA-implementable programs or objects.

Also, in the same way that the linking and loading process of embedded systems design connects various system objects, subsystems, or super systems like the operating system, including library objects (and loads or places them into specific memory locations), the place and route function in FPGA design places the synthesized subsystems into FPGA locations and makes connections (microprocessor links ~ FPGA routes) between these subsystems, enabling their operation as an integrated system. The actual linking and loading of compiled bits is essentially a process of fitting, in one dimension, the bit patterns distributed over a set of available linear memory addresses. The FPGA place and route process fits, in two dimensions, the bit patterns (logic subsystems) over a two dimensional array of available logic gates and routes buses between these logic subsystems as necessary.

The similarity in the processes is obvious.

Loading

Finally, just as embedded programs are often embedded in physical ROM, flash, or downloaded live, FPGA programs (compiled, synthesized, placed, and routed) must be embedded in the physical FPGAs. The actual programming file may be a .HEX or similar. Programmers typically download or burn the bits from these files into the hardware. If nonvolatile, this is a one-time proposition. If not, it's a download-at-power-up proposition. Many variations exist with FPGAs as with microprocessor-based embedded systems, but in the end, in a functioning microprocessor-based product, the bits compiled, linked, and loaded must "get into" the physical memory to control the gates of the processor, and in an FPGA-based functioning product, the bits compiled, synthesized, placed, and routed, must "get into" the FPGA, to implement the gates of the system.

Debugging programs

All experienced programmers know that complex programs, even subpro-

The simulation process observes the transformations and translations of signals as they propagate through the FPGA from the input pins and provides responses that eventually reach an output pin.

grams, don't run correctly the first time. When we first see how to solve a problem, we tend to be overjoyed, (it's possible!) and then underestimate the time required to implement the solution. This is because our powers of abstraction, based on experience, enable us to see solutions without worrying about every nitty-gritty detail. But the hard truth is that the physical system in which we're embedding our programs requires that every nitty-gritty detail must be handled before it will work. No doubt, you have had the experience of fixing a single bit (or flag) that caused a crashing program to suddenly run correctly. Hallelujah! It works. I can sleep now.

Anyway, things don't work right out of the gate. We generally have to kick them and see what they do and where they expire. In embedded systems program development, we typically use debuggers, simulators, and emulators. These tools enable us to step through the program execution and observe the effects on flags, register contents, memory locations, and so on, and to try to match what we expect at a given place and time with what we see, in the simulator or emulator.

Simulation vs. emulation: Debugging in FPGA design is largely based in simulation. You can see why. Emulation, in the context of embedded microprocessor programs, typically refers to executing programs on special in-circuit emulation (ICE) hardware designed to 1) run exactly like the target machine and 2) provide visibility, access, and control of the target machine in powerful ways, with emulation timing exactly equal to target timing. Thus, the essence of emulation is based on physical hardware that reproduces the microprocessor's gates while adding gates designed for debugging. This methodology works because the microprocessor consists of a fixed array of gates, implementing the CPU, among others.

The FPGA does not have a "fixed" pattern of gates, in the sense of the CPU; in fact, the object of the FPGA programs is to define a pattern of gates, via the process described herein that will implement our design. And our design will differ from every other design. This is why no FPGA emulators exist, in the sense of in-circuit emulators.

Today the fastest processors cannot be emulated in hardware, since they already run as fast as possible, and the ICE circuitry adds additional gate levels, thereby slowing the ICE and preventing it from keeping up with the processor. For this reason, processor designs focus on built-in debugging support. This approach is somewhat analogous to FPGA design, where it's helpful to build in debug aids. Because you're used to building debug aids into programs, from PRINT statements on up, you'll understand the significance of this point, and be able to translate your understanding into the FPGA realm.

Most microprocessor programming, particularly embedded programming, is based on the assumption that sophisticated operating systems and peripheral devices are "connected to" the program, through application interfaces. Thus debugging focuses on the inside of the machine and doesn't attempt to simulate the operating system or the peripherals but instead works with them.

Testbenches in FPGA: Most FPGA systems are standalone systems connected to the real world, and functioning in interaction with the real world. Therefore a large part of debugging and test is concerned with simulating the real world to which the FPGA will attach. In reference to the way logic circuits were debugged historically, this is called the *testbench* and is considered an integral whole—in other words, it can be compiled as a whole. Simulation typically steps through the operation of the testbench, which stimulates the FPGA. The simulation process observes the transformations and translations of signals as they propagate through the FPGA from the input pins and provides responses that eventually reach an output pin.

Because there will likely be tens or hundreds of thousands of gates, you obviously don't watch all of them, but the most meaningful ones in the area of interest. For example, a FIFO buffer may be 48-bits wide and thousands of words long, but the gate output that most interests you at the moment may be the `Buffer_Full` signal, so this would be displayed on the simulation screen, along with other signals of interest. This process somewhat resembles focusing on flags or semaphores in microprocessor programming. You can group multiple signals. For example, eight wires may be given a name, `Data_In_Bus`, and the numeric values appearing on the bus may be displayed, as opposed to showing all eight lines constantly changing. State variables can be given ASCII names and displayed. In the same sense that you may want to focus on program execution within a subroutine, you may want to observe only the signal activity in a subsystem. Once you learn the tools, it becomes quite natural to debug an FPGA subsystem you've designed.

Bottom line: FPGA debugging is done by software simulation and by using actual hardware. Any debug use of the actual hardware must be designed by you.

Netlist and Top ~ Main: Although it's not really a part of the process, it's worthwhile to understand that the output of the FPGA design process is a *netlist* or list of nets or wires that connect gate outputs to other gate inputs. Further, there is a top level from which everything descends. Think of the top as the Main point in a microprocessor program where the program starts. Although there may be 50 or more modules that are created independently in an FPGA design, when the

Probably the biggest surprise for experienced embedded programmers is that programs that are functionally good, that run, and can be simulated, and produce correct results, may not be synthesizable.

process is finished, all will be linked in the netlist. Any module not in the list will have no effect. This is analogous to a subroutine that is never called. If there is no connection to a module, the module can't do anything.

Documenting programs

I can't think of any significant differences between good documentation practice in microprocessor-based embedded systems and FPGA-based embedded systems programming. The contents will obviously differ somewhat, but the practice and procedures are the same.

Delivering product

FPGAs are surprisingly similar to microprocessors in the actual delivery of working hardware and software.

The microprocessor hardware, boards, power supplies, connections, must be correctly designed, and the software must be burned in or downloaded as described above.

The FPGA hardware, boards, power supplies, connections, and so forth, must be correctly designed, and the software must be burned in or downloaded as described above.

If the hardware is correct, the software can evolve. This allows bug fixes and feature addition. This is true for microprocessors and FPGAs. The Verilog or VHDL hardware description language (HDL)—each a high-level design language—provides fast time-to-market using FPGAs. The HDL system allows design, debug, and verify—all within the same environment. As with microprocessor design, FPGA design can be command-line driven or IDE-based.

Differences

Most of what we've just discussed has focused on the similarities between embedded microprocessor programming and FPGA programming. We now look at some significant differences.

Unsynthesizable

Probably the biggest surprise for experienced embedded programmers is that programs that are functionally good, that run, and can be simulated, and produce correct results, may not be synthesizable. What does this mean? It means that you can write "good" functional programs that are impossible to convert into a netlist that can be mapped into an FPGA. Why is this? Primarily because Verilog is a "superset" of synthesizable syntax. Historically, Verilog was designed as a simulation language for simulating logic systems. It was only later that synthesis technology was able to actually convert the RTL output of the simulation compiler into netlists based on gate-level structures actually found in FPGAs. It is therefore understandable that the full simulation language, designed before synthesis tools, is not fully synthesizable. What does this mean to you? It means the problem is more complex than it initially seemed. How to create synthesizable programs is beyond the scope of this article, but it's a subject for future articles.

Why would you even want to use a language containing nonsynthesizable constructs? There are several reasons. We'll look at two.

First, you may want to represent a system that will later be partitioned into software and hardware subsystems. It's easier to design such a system using the full language and later restricting the hardware portion of the design to synthesizable constructs.

A more general reason is the following. FPGA design should always include a testbench, which is the environment that provides inputs, including clock(s) and data, and accepts outputs from the FPGA. It's the software that describes the world as "seen by" FPGA pins. This world is compiled to be simulated but not synthesized. Think about this. The code in the FPGA must be mapped into real logical gates in the FPGA, therefore, by definition, it must be synthesizable, since synthesis is the process of converting RTL language into gate-level language, and hence, into a field programmable gate array. But the code "outside of" the FPGA is not going to be put inside of an FPGA. It's going to be used by the designer to simulate the environment of the FPGA, while debugging. For this reason, it's useful to allow high-level programming constructs that simplify the construction of the testbench. If the testbench construction were limited to synthesizable constructs it would force the designer to use lower-level abstractions than is necessary. Figure 2 shows the synthesizable and nonsynthesizable portions of a design.

This is a very important point, so I'll repeat it. The Verilog language, designed initially for simulating logic, offers powerful high-level constructs that are useful for simulating the "real world" to which the FPGA will be connected. The constructs will not be mapped directly into FPGA logic structures or be converted into a gate-level netlist. Since only a subset of the Verilog language is synthesizable, that testbench design is easier, but FPGA programming, per se, is more difficult. Newer versions of Verilog (coming on the market "real soon now") will change the "mix" of synthesizable to nonsynthesizable language abstractions, but this problem will probably always be with us to some degree.

C and Verilog

For the last few years, there has been an effort to extend Verilog by adding C language constructs. The effort is now culminating in IEEE standards processes, but the situation is unresolved at this time, with the (hopefully remote) possibility that the language will split into two versions. The problems are legal, commercial, and conceptual. It's also fair to say that most Verilog programmers are unenthusiastic about the process,

For the last few years, there has been an effort to extend Verilog by adding C language constructs. This effort is now culminating in IEEE standards processes, but the situation is unresolved at this time.

but the companies involved think that they could sell more software if it becomes easier for the army of C programmers to participate in FPGA design; therefore, there's real pressure to proceed with the standards process. Because the uncertainties will be resolved over the next year or so, we won't focus on this situation here.

The situation is analogous to the assembly language vs. C programming history. When resources are scarce, it pays to design efficiently, therefore assembly language is best. When resources become free, the efficiency of the design process dominates, and C language programming is preferred. In the FPGA world, resources are not yet free, therefore Verilog is the language of choice. This will probably change over time.

Note that a C interface to Verilog already exists, as shown in Figure 3. There are also variants of FPGAs that contain a microprocessor core on the silicon along with the FPGA circuitry. In such cases the microprocessors can be programmed using C while the FPGA gates would be programmed using Verilog or VHDL. The use of C in the functional testbench code makes sense, because it need not be synthesizable.

For those embedded systems programming teams that would like to extend their capabilities into FPGA design, I recommend that those with the least understanding of hardware should focus on testbench design, while those more capable of hardware design focus on learning how to write synthesizable code.

Optimization

Now that we have a more complete picture of FPGA design, let's compare

FIGURE 2 Synthesizable and nonsynthesizable portions of design

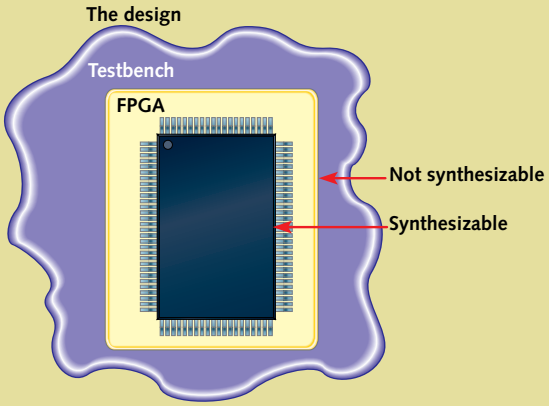
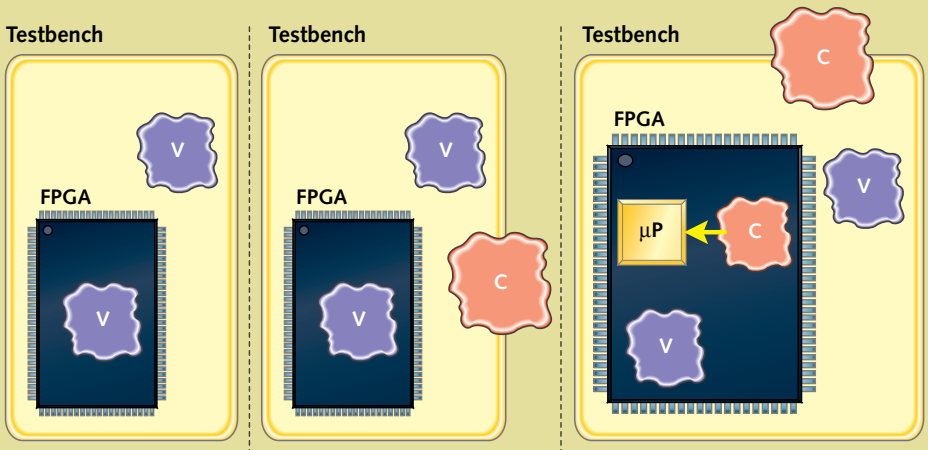
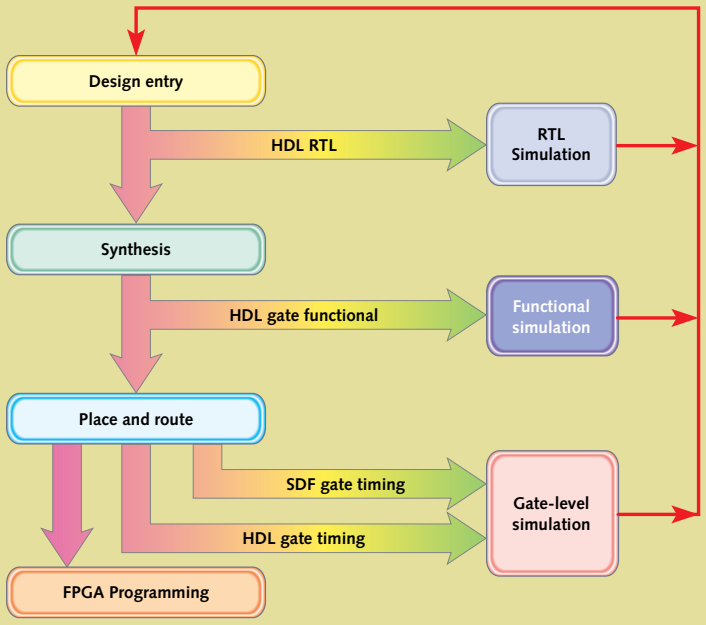


FIGURE 3 Testbenchs and C code in debugging



Most synthesizers can produce a Verilog language description of this gate-level code. The beauty is that this gate-level Verilog can be compiled and simulated. Thus, we can debug at the actual gate level.

FIGURE 4 The FPGA programming process



the difference in outputs before and after placing and routing.

When the system designed in Verilog is compiled, the output is an RTL netlist. When input to a synthesizer, the Verilog is converted into a gate-level netlist, capable of being mapped into FPGA hardware (assuming successful synthesis.) Most synthesizers can produce a Verilog language description of this gate-level code. The beauty is that this gate-level Verilog can be compiled and simulated. Thus, we can debug at the actual gate level. The simulation of the RTL Verilog is called *functional simulation*, while the simulation of the synthesizer Verilog output is called *gate-level simulation*, as shown in Figure 4.

What's the difference between functional and gate-level simulation? One difference is that, just as C compilers can optimize C code, synthesizers can optimize FPGA netlists. In fact, if you specify the goal, synthesizers can optimize to meet your goal. The goals are typically *area vs. delay*. Area optimization will attempt to use the fewest number of gates (silicon area) on an FPGA, at the expense of execution speed. Delay optimization attempts to maximize the execution speed, even if more FPGA area is required. The net result is that the functional code you wrote in Verilog at the RTL level may have different implementations, and signals that you used to debug the functional code may have been optimized out of existence. That is, they may disappear in the final gate-level implementation. Thus, even though you've thoroughly tested and simulated the RTL code, you'll want to do the same at the gate level.

Synthesizers typically allow constraints to be specified as part of the optimization process. One such constraint is `dont_touch` to prevent the synthesizer from doing whatever it wants to specific elements of the design. Another constraint is `preserve_hierarchy` as an alternative to flatten the design. Because hierarchical boundaries can prevent or limit optimization, synthesizers, which flatten the design will typically provide more optimal results.

SDF and back-annotation

OK, now we're ready for the last major difference between microprocessor and FPGA programming processes. Just as there are cases in microprocessor design where speed of execution is critical, the same is true for FPGA design. When microprocessor code must be timed, the clock speed and number of cycles per instruction can be used to compute execution speed, which will be strongly processor dependent.

When a synthesizer produces gate-level Verilog for an FPGA, it's strongly FPGA dependent; that is, the delays associated with vendor-specific FPGA structures are known and can be used to compute operation speeds. Thus simulation of gate-level code for a specific FPGA is realistic in this sense.

But remember, we still have to place and route this gate-level netlist. This operation will add delay for longer routes, thus slowing the final execution speed. If your design must meet some real-world spec, such as a 12MHz USB (48MHz clock) or 480MHz USB2.0 then you must run at this speed, or you haven't solved the problem. How can you tell whether the routed code will run fast enough?

To solve this problem, place and route programs (supplied by the FPGA vendors) will also produce Verilog output and will produce SDF files, which are files in standard delay format, that capture the delays associated with the placed and routed netlist. Simulators can use this SDF information to back annotate the gate-level code, thus allowing simulation of the final FPGA design at its most accurate. Because the FPGA elements are well characterized, with typical setup

and hold times, the simulator can detect failure to meet these specs. On graphical waveforms, the failures typically show up in red, while good timing is "in the green" when the desired clock frequency is used. When the FPGA runs in the green with the desired clock frequency used, and behaves in the testbench as is desired, you have an FPGA design that's ready to be downloaded to hardware for real-world testing.

Perhaps we should point out here that the three versions of FPGA code simulation, RTL, functional, and gate-level will typically all use the same testbench code; that is, there are not three versions of the testbench.

In review

We've seen that, conceptually, microprocessor programming and FPGA programming follow almost identical paths. They're based on architectural design and are described in terms of C-like high-level languages, which are compiled or synthesized, possibly in conjunction with third-party library objects. The output of this process linked and loaded in one memory address dimension or placed and routed in two dimensional gate arrays, such that these fixed bit patterns can be downloaded to correctly designed hardware and, eventually, be made to function in the manner for which the architecture was designed.

When gates were precious entities and tools 100% proprietary, it made ultimate sense to arrange these limited gates into universally used objects, such as CPU registers, ALUs, instruction decoders, and address decoders. You would then provide a set of instructions that linked and relinked these elements, so that, for example, two CPU register outputs could be connected (via buses) to an ALU input, then the ALU output connected to a destination register, and then the ALU input connected (via buses) to a specific memory address, and the ALU output connected to a different register, and so on and so on. It made perfect sense.

One difference in microprocessor and FPGA design is subjective. There is an astonishing elegance and "cleanness" of FPGA design vs. microprocessor program design.

When the scale and, therefore, the economics changes, everything changes. When gates are no longer precious but are commodities, the fixed elements approach no longer makes as much sense. The monstrous development in languages, tools, I/O devices, standards, and so on will keep CPU development and implementation alive for decades, if not centuries, but the economics are now and trending more so in the FPGA direction. This has recently been given another economic boost relative to ASICs. The cost of repairs in hardware—that is, ASICs—is increasing drastically with decreasing line width and increasing gate density, making FPGA technology even more relevant for embedded systems designers. Today over a million gates are available, tomorrow 10 million, accompanied on a single chip by millions of memory bits. You can make anything you want by describing it in a programming language, such as Verilog, and going through the process described above.

FPGA elegance

One difference in microprocessor and FPGA design is subjective. There is an astonishing elegance and "cleanness" of FPGA design vs. microprocessor program design. In design after design, I've realized how much time is spent in embedded system programming "getting ready" to do something. This involves setting up data in registers or memory locations and setting up pointers in other registers, to get some source, do something, and put it somewhere.

Programmers don't really see this, just as fish probably don't see water, because that's the nature of the process. It's less visible with high-level language and more visible with assembly language, but it's always there.

In FPGA this "getting ready" doesn't really occur. Everything is where it belongs and happens all at once, in one clock cycle.

This is not to say that you can't design registers, buses, and ALUs in FPGAs, but you'll find that you really don't spend much time "getting ready to do something." I won't push this point, because you have to design FPGAs before it hits you over the head. But remember: you read it here first.

Another significant difference with FPGA design lies in the parallel nature of FPGA processes. Instead of a single program counter based "locus of control," an FPGA typically clocks all gates at once. Thus you can have many processes occur in parallel, instead of sequentially. This also takes some getting used to because it's so different from the way programmers think.

Why would you want to program an FPGA in the first place? Well, if you're designing accounting programs, you don't. But many embedded systems are tightly coupled to the real world, and there are many problems that simply happen too quickly to be handled in software. In this case you can let your competitor have these problems (which tend to be expensive!) and you can stick with the slower, easier (low-profit) problems. Or you can program FPGA solutions. It's not mandatory. It's an opportunity. **esp**

Ed Klingman worked as a research physicist at NASA for seven years, then founded Cybernetic Micro Systems, Inc, now celebrating its 25th anniversary. He is author of the Prentice-Hall textbooks Microprocessor Systems Design, Vols. I *and* II *and numerous technical papers, and has been awarded 20 U.S. patents. You can reach him at klingman@geneman.com.*