Foreword

How cool is this? A "programming" book written for both software developers and hardware design engineers.

Where I came from, the software folk and the hardware folk got along pretty much okay, but they sometimes seemed to be from different planets. The hardware engineers would roll up their sleeves and work for months without a break, while the software programmers would sit back and relax, or play ping-pong, or whatever software programmers do, until the hardware was stable and software development could begin in earnest. At this point the hardware team would be off planning their vacations, repairing their marriages or trying to remember where they lived. All to no avail, because the software team would soon be coming after them for some perceived deficiency or new feature request, and the hardware design would quickly become a hardware redesign.

It's amazing that the software and hardware teams I worked with didn't break into full-on hand-to-hand combat. I suppose that in hardware/software codesign, as in life, "good fences make good neighbors."

Why is this book of interest to the hardware folks?

As "I R A Hardware Engineer" (maybe you figured that out already), let's start by considering things from this side of the fence. Electronic design automation (EDA) refers to the software tools used to design electronic components and systems. In the early days of digital logic design, we captured our circuits as gate-level schematics.

This approach began to run out of steam in the mid- to late-1980s, but we were saved by the introduction of sophisticated hardware description languages (HDLs). These allowed us to describe the functionality of a design at a reasonably high level of abstraction called the register transfer level (RTL). In turn, these descriptions allowed alternative design scenarios to be more quickly and easily represented and simulated. And, of course, the real boost to productivity came in the form of logic synthesis technology, which could automatically compile these high-level descriptions into equivalent gate-level netlists.

Another consideration is the way in which a design is physically realized. Initially, the only options were to use lots of off-the-shelf "jelly-bean" devices and simple programmable logic devices, or to create your own application-specific device. Then, in 1984, a new component appeared on the scene: the field-programmable gate array (FPGA). The first incarnations of these devices were relatively simple and contained only a few thousand equivalent gates, but my, how things have changed. Today's offerings can boast tens of millions of equivalent gates coupled with large numbers of embedded functions like RAM blocks and multipliers.

And now we are running into a design productivity crunch again. Although languages like Verilog and VHDL are great for precisely describing a design's functional intent, they are somewhat cumbersome when it comes to representing today's multimillion-gate systems. Actually capturing and verifying (via simulation and/or formal verification) such a design in HDL is very time-consuming.

A bigger problem, however, is that describing a design using a traditional HDL requires us to make a lot of implementation-level (microarchitecture) decisions. In turn, this makes make exploring alternative design scenarios and architectures time-consuming and complex. For example, if you already have a design captured in HDL and someone says, "Let's try doubling the number of pipeline stages to see what happens," you know that you can cancel your plans for the coming weekend.

The solution is, once again, to move to a higher level of abstraction. The most natural level for hardware design engineers is the standard ANSI C programming language (don't talk to me about C++, because trying to wrap my brain around that makes my eyes water). This book shows you how to express designs using standard ANSI C extended with a small number of C-compatible library functions. Once the design has been captured and verified (and alternative solutions and architectures explored), the book also discusses the tools used to automatically convert the untimed C description into its synthesizable HDL equivalent.

The end result is that this methodology allows hardware design engineers to achieve a working prototype much faster than using an HDL-only approach. "But what of the software folks?" you cry…

And what about the software guys and gals?

Actually, the software fraternity is the primary target of this book, because the techniques described allow software-centric engineers to take advantage of the massive amount of parallelism inherent in a dedicated hardware implementation while still using a software development methodology.

Perhaps not surprisingly, the majority of software development engineers writing C code for digital signal processing (DSP) and embedded applications tend to regard "raw hardware" in the form of an FPGA with anything ranging from trepidation (on a good day) to outright loathing. But fear not, my braves, because "there's nothing to fear but fear itself," as my dear old dad used to say.

In the case of someone tasked with creating a DSP application, for example, the majority of DSP code first sees the light of day in C. As horrendous as it seems, once the algorithms have been verified at this level of abstraction, a large proportion of the code gets rewritten and "tweaked" in assembly code in a desperate attempt to achieve the required performance. This manual translation is, of course, both painful and time-consuming.

The bigger problem is that the DSP code will eventually be run on a general-purpose microprocessor or a dedicated DSP device. Both of these realizations are inherently slow, because they are based on a classical von Neuman architecture, which requires them to

- Fetch an instruction
- Fetch a piece of data
- Fetch another piece of data
- Perform an operation
- Store the result

  :
- Do the same thing all over again

Now, consider a typical DSP-like function along these lines:

```
y = (a * b) + (c * d) + (e * f) + (g * h);
```

If you run this through a DSP chip, it will take a substantial number of operations and clock cycles to execute. Now consider an equivalent dedicated hardware implementation in an FPGA, in which all of the multiplications are performed in parallel without the need to fetch and decode the instructions. This results in orders-of-magnitude speed improvement.

Thus, one of the purposes of this book is to show you how to take your C code representation and use it to "program" an FPGA to implement your algorithms directly in hardware.

These techniques are not limited to DSP applications; they are also of interest to embedded applications developers in general. The concepts described in this book allow you to view all or part of an FPGA as being "just another programmable core." The difference is that, instead of running your code on an embedded (soft or hard) microprocessor core, you can now decide to implement appropriate portions of the code directly in hardware to achieve phenomenal increases in performance.

So what's the catch?

Actually, there is no catch. As the authors say, if you can write C for applications targeting standard processors, you can write C for programmable hardware.

There are certainly other C-based methodologies and tools available for FPGA designs, but these tend to be focused on satisfying the requirements of the hardware design engineers. By comparison, *Practical FPGA Programming in C* is unique in that it addresses the needs of both hardware and software developers. While it doesn't completely tear down the fence between hardware and software engineering teams, it at least provides another gateway.

Clive "Max" Maxfield — Christmas Eve, 2004