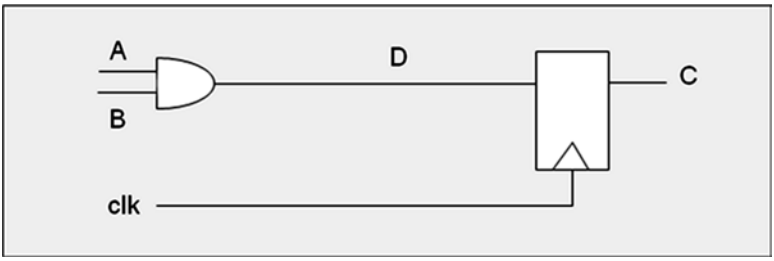
Preface

On a bleak January night in 1992, I sat hunched over a computer screen and a logic analyzer. It was well past midnight, and I was the only person in the lab – probably the only person in the building. We had just gotten an ASIC back from the ASIC house, and, of course, when we fired it up it didn't work. The other guys had narrowed the problem down somewhat; now it was my turn to try to find the cause.

I had narrowed it down to a particular module which had been written by an engineer we'll call Jeff. Working my way through Jeff's code, trying to find the cause of the bug, I realized that he had not indented his *if-then-else* statements. This made it absolutely impossible for me to figure out what his code was doing. So at 1:00 in the morning, I spent an hour or so carefully indenting his code – and thinking very unkind thoughts about Jeff. Once I had his code carefully laid out, it was trivial to find the problem - it was an *else* that should have been associated with a different *if*. Of course, with such poorly structured code, it is unlikely that Jeff knew exactly what his code did. Otherwise he would have spotted the rather obvious problem himself. Instead, the problem made its way into the silicon. Fortunately, we were able to compensate for with a software change.

In early 2010, I happened to interview a significant number of candidates for an entry-level design position. Most of these candidates were right out of school, but a few of them had a couple of years of experience. To each of these candidates I gave a very simple problem, to set them at their ease before I started asking the hard ones. I drew this on the whiteboard:



Every single candidate who was right out of school said (pretty much word for word):

"Let us label the output of the AND gate D."

Then they wrote:

```
assign D = A && B;
always @ (posedge clk) begin
   C <= D;
end
```

None of the experienced candidates wrote it this way. And it never occurred to me that anyone in their right mind would. The experienced folks all wrote:

```
always @ (posedge clk) begin
   C <= A && B;
end
```

This answer is simple and straight to the point. Why would you ever add an extra line (and an extra process) to the code? It makes no real difference in this trivial example. But when one is hunched over a computer screen and a logic analyzer in the small hours of the morning, extra lines of code and extra processes can become very expensive. Especially in designs that are tens or hundreds of thousands of lines of RTL code, all written by someone else.

These two events have bracketed twenty years of trying to deal with, and improve, code based design. As a manager and as a researcher, I have spent far more time reading other peoples' RTL than my own. I have watched how engineers struggled when they had to debug someone else's code – often to the point where it was easier to re-write the whole module than find the bug.

In running an IP development team, I learned how critical quality is, and yet how difficult it is to achieve zero-defect code. For years I thought this was a verification problem. I am now convinced that the problem is how we design hardware and how we write RTL code.

Recently I have spent a lot of time looking at high level design and synthesis tools. I think there is some real value in the approaches they take. But I think there are significant opportunities for improvement in how they approach the design problem. Most importantly, I have come to realize that good design and good code do not miraculously emerge from raising abstraction.

Good design and clean code are a fundamental challenge to the human intellect – to make simple the complex, to make clear the obscure, and to add structure to what can look like complete chaos.

This book is an attempt to frame and answer the question of what makes good design and clear code. It presents my conclusions from the last twenty years of struggling with the problems and challenges of designing complex systems – and in particular, the design of SoCs and the IP that goes in them.

It would be impossible to thank all the people who have helped me as I developed (and borrowed, and occasionally stole) the ideas presented in this book. Many of

my friends and colleagues - both inside and outside Synopsys – have spend numerous hours talking (and arguing) over these issues.

But I would like to thank specifically the IP development team in Synopsys, including Subramaniam Aravindhan, Steve Peltan, James Feagans, Saleem Mohammad, Matt Meyers, and Qiangwen Wang.

I'd like to thank Tri Nguyen, Aaron Yang, and Shaileshkumar Kumbhani who as interns helped with many experiments whose results have shaped my thinking and the conclusions discussed in this book. I am happy to report that they all now have real jobs as IP developers.

Finally, I'd like to thank Jason Buckley, Badri Gopalan, Arturo Salz, Dongxiang Wu, Johannes Stahl, Craig Gleason, Brad Pierce, and David Flynn for their valuable discussions and feedback on the manuscript.

http://www.springer.com/978-1-4419-8585-9

The Simple Art of SoC Design
Closing the Gap between RTL and ESL
Keating, Synopsys Fellow, M.
2011, XVII, 234 p., Hardcover
ISBN: 978-1-4419-8585-9