Digital Design: Computer Science or Electrical Engineering?

Mark Zwolinski
Electronic Systems Design Group
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
mz@ecs.soton.ac.uk

Abstract

The use of hardware description languages has moved digital systems design closer to computer science. Software engineering techniques are needed to manage complex designs. Deep sub-micron effects have made digital design more asynchronous and more analogue. Therefore there is a tension between high-level abstraction and low-level detail. In this paper, it is argued that we should increasingly teach digital design as if it were a form of software engineering and that the low-level effects must be estimated in the design tools. This is a challenge to universities, both in their teaching and in their research.

1 Introduction

In the past two decades, the world has "gone digital". CDs have replaced vinyl records; digital photography has replaced film and television, radio and telephones have all become digital devices. Increasingly, therefore, *electronic* engineering has meant *digital systems* engineering. With the advance of digital technology has come a massive increase in complexity. Design tools have struggled to keep pace with this new complexity. Similarly, engineering education has found it difficult to stay in touch.

Along with complexity another phenomenon has appeared. As the feature size of integrated circuits shrinks, gates and flip-flops behave less like synchronous digital devices and more like analogue components. It is now more correct to think of integrated circuits as transmission lines connected by switches and not as gates connected by equipotential wires.

Thus, we have two conflicting pulls: on the one hand in order to control the complexity we need to describe digital hardware using software engineering techniques; while on the other we need to be familiar with electrical engineering principles in order to understand and control the sub-micron effects.

In this paper, we will review the digital design curriculum as it has been taught. We will then examine what we can learn from computer science in order to teach the management of complexity. We will argue that to manage the non-synchronous, analogue features of deep sub-micron design we need a new generation of design tools and a new curriculum for a new generation of electronics engineers.

2 Digital System Design 1980-2000

In most electronic engineering degree programmes digital design forms one of the main themes alongside analogue circuit design, programming and physical electronics. Here, we distinguish electronic engineering from electrical power engineering, in which there is an emphasis on machines and high voltages. In some institutions there was a move towards electronics in the decades before 1980. In others, such as the University of Southampton, electronic engineering has always been a distinct discipline.

Through the 1980s, the digital design theme would have included topics such as: Boolean algebra; Boolean minimisation using Karnaugh maps; optimisation of logic in terms of TTL packages; state machines; state minimisation; and mapping to JK flip-flops. Depending on the institution these subjects would have been more or less theoretical, with perhaps an element of design and build in the laboratory. Related topics would have included computer architecture, including bus architectures and assembly language programming and integrated circuit design.

By the mid-1990s, VHDL and Verilog were starting to appear, together with programmable logic. Thus topics such as TTL package minimisation and optimisation in terms of JK flip-flops became obsolete. By removing these topics, there was room in the curriculum to introduce RTL design, including synthesis and simulation. FPGAs made it possible for students to design much more complex systems. It was also realised that design for test is an important aspect of digital design and topics such as the single stuck fault model, scan path design and BIST could be included.

The power of the design tools has revealed a difficulties however. At first glance (and to weaker students) a specification in VHDL is not that different to a C program. The design process has similarities to programming a PIC or other embedded processor: the "program" is compiled and downloaded onto an integrated circuit on a development board. Moreover, the complexity of both FPGAs and embedded processors means that it is impossible to debug either hardware or software designs *in situ*. Finally, we cannot simply give an FPGA or a processor with 100s of pins and operating at 2.2V or less to students and expect them to build circuits in the laboratory.

Thus in practice digital systems design has become a branch of software engineering.

3 Computer Science Lessons

We can regret this transformation of the digital design process or we can welcome the change and seek to exploit it. So, what can we learn from computer science?

The most significant change in computer science teaching in the last 20 years has been the evolution in computing languages. In 1980, computer science students would have been taught FORTRAN IV and possibly Lisp. Today students learn Java and C++. It is said that "real programmers can write FORTRAN in any language"! This is true, but good programmers can use the expressive power available in C++ to write shorter programs that achieve the same task. The key here is abstraction and management of complexity. It is relatively easy to map a FORTRAN program into assembler; it is almost impossible to do the same with a C++ program.

It is not just the mainstream programming languages that have changed. Software management tools have made it easier for teams to develop programs – even simple tools such as makefiles allow programming tasks to be shared. Version control systems (RCS, CVS) allow changes to be tracked. C++ has vast libraries of functions that can easily be used to build sophisticated applications. Scripting languages (e.g. Perl) allow systems to be built from small components.

Perhaps the most significant advance has been in the development of formal methods such as model checking. It is practically impossible to test for every combination of data; formal methods can be used to validate a specification against a final implementation.

By analogy, digital hardware design, in the form of RTL synthesis, is still in the FORTRAN era. Software engineers no longer care exactly which machine instructions are executed, but digital hardware designers are obsessed with knowing exactly what flip-flops are created. IP reuse is talked about but seldom done. Ironically, many engineers are forced to learn Perl in order to make the EDA tools work. And formal methods are not mentioned in polite company!

This situation cannot continue. There is a "design gap". A consumer integrated circuit might remain in production for a few weeks, but might have taken several hundred man years to design.

The current response of integrated circuit manufacturers is to export design to low wage economies, such as India and China. Thus the cost is kept low by reducing the cost per engineer. An alternative model is to increase the productivity of each engineer. The lessons of computer science suggest a way to achieve that objective.

4 Electrical Engineering Tamed

The argument for abstraction might be countered by observing that as devices get smaller the engineering problems get more difficult. As educators we can promote the use of software engineering tools. As researchers, however, we have new opportunities in trying to reconcile electrical engineering with computer science.

At this point an example might be useful. A common problem in RTL design is that of timing closure. A designer wants to achieve a certain speed with minimal hardware costs. Speed versus area is the classic trade-off. In order to minimise area, resources are shared. In order to share resources, multiplexers must be included. Therefore, when synthesised the design does not meet the speed requirements. Hence, the shared resources are no longer shared, but this makes the design larger and the delays greater, due to longer wiring. Thus the speed gets worse, not better and thus timing closure is never achieved.

The problem arises because the synthesis tool estimates performance only in terms of logic delays, not wiring delays. The solution, as implemented in so-called physical synthesis tools, is to estimate the wiring delay and to include this as a cost in the optimisation function. It is important to appreciate that, at this level, the wiring cost is estimated – it would be prohibitive to perform a full layout at each optimisation iteration.

In a computer science view of synthesis, the physical effects are ignored. In an electrical engineering view, these effects dominate the design process, impeding abstraction. We need to tame the electrical engineering problem by producing relatively simple high-level models of the low-level effects. In the timing closure problem, this is done by generating a floorplan of the design and estimating typical and worst-case delays. We do not attempt to calculate exact delays. In fact this estimation has a second benefit: the floorplan is based on high-level information about the design, which is lost during synthesis. The floorplan can be passed onto low-level layout tools, allowing them to produce a better solution more efficiently.

To date, relatively little work has been done in high-level estimation. In commercial tools, physical synthesis has been applied at RTL, but there has been research into using similar techniques at a behavioural level. Similarly, the cost of a design in terms of overall area, power and testability can be estimated at an early stage. Problems such as crosstalk and asynchronous communication could be designed out of a system at an early stage. We should, perhaps, think of this as architectural exploration rather than behavioural synthesis. Models and tools for performing these estimates and, importantly, for arbitrating between conflicting objectives are needed and could provide a fertile area for university research. Such tools would therefore allow us to control complexity.

5 A Curriculum for Digital Systems Design

It is not really possible to design a curriculum for digital systems design in isolation. In practice an electronic engineering degree should also include: Mathematics; Circuit Theory; Analogue Design; Computer Architecture; Programming; Solid-state Physics; Electro-magnetism; Communications; Signal Processing; and so on. Different countries have degrees of different lengths. Different institutions have different specialisations and structure their programmes in different ways. We can outline key topics that should be included in a digital systems design theme.

In the first year, it is clearly necessary to introduce the basics: Boolean algebra; Karnaugh maps; gates; flip-flops; state machines and programmable logic. Laboratory exercises will include the use of simple programmable devices and hence require the use of languages such as Abel. There is a clear need for students to understand, from a practical point of view, the need for decoupling capacitors and the effects of sending pulses along transmission lines.

In the second year, hardware description languages (VHDL, Verilog or SystemC) can be introduced. Coupled with this is the need to introduce students to simulation and RTL synthesis tools. Even though an FPGA can be easily reprogrammed, it should still be right first time, because it is almost impossible to access internal states for debugging. The principle of dividing a design into controller and datapath can be taught with all these subjects being assimilated through design exercises. Finally the idea of design for test should be introduced, together with the principles of test pattern generation.

In the third and subsequent years, the level of abstraction can be moved upwards. At this point software tools (e.g. CVS) need to be introduced. Verification principles, particularly formal methods, and tools need to be explained. With the increasing complexity of systems on chip, the idea of a single, global clock is no longer tenable and design for asynchronous communications needs to be introduced. As we now no longer distinguish so rigidly between hardware and software, we can introduce the idea of hardware/software co-design.

As high-level synthesis tools become more readily available, they need to be introduced into the curriculum. At first these will be university tools, suitable only for advanced courses. It is inevitable, however, that such tools will become industry standards and their use must be promoted in the teaching of digital design.

6 Conclusions

With increasing use of hardware description languages, digital design is starting to resemble software engineering. At present, tools and techniques do not fully support modelling and estimation of low-level electrical effects. As a result of industrial pressure and university research tools will emerge. It is only through the adoption of software engineering methods that the complexity of future generations of digital devices can be managed.

Is digital design computer science or electrical engineering? Undoubtedly, the answer is that from the designers' point of view, digital design will increasingly resemble computer science. We will still need, however, tool builders who understand electrical engineering and there will be occasions when the assumptions implicit in those tools break down. We still need therefore generations of fully rounded electronics engineers.