Unleashing Student Creativity with Digital Design Patterns

Dr. Miguel Bazdresch, Rochester Institute of Technology

Miguel Bazdresch (mxbiee@rit.edu) obtained his PhD in Electronic Communications from the Ecole Nationale Superieure des Telecommunications, in France, in 2004. He worked for several years designing digital integrated circuits for the telecommunications industry. After teaching at ITESO University, in Mexico, from 2005 to 2012, he came to the Electronics, Computers and Telecommunications Engineering Technology Department at the Rochester Institute of Technology, where he is an Assistant Professor.

Unleashing Student Creativity with Digital Design Patterns

Introduction

The design of digital logic circuits is, in part, a creative process. A solution to a logic design problem must be imagined before it can be implemented. Creativity takes time and effort to develop. With sufficient experience, students who design logic circuits can become quite proficient in it (this often occurs only after graduation). In a classroom environment, however, teaching creativity is challenging. We believe that, with specific techniques, it is possible to encourage its development in such a way that students are able to design, implement and test solutions to more complex problems than before.

Our approach is as follows. The most general circuit design technique is arguably top-down design, using the divide-and-conquer strategy. When faced with a complex problem, however, students often have difficulty in determining how a design should be partitioned. This process is made easier if students formulate the end result, that is, the large set of simple problems that the original problem will be broken into. We mix top-down and bottom-up strategies so that the creative process has definite end-points, with the complex problem at one end and simple problems at the other end. Then, creativity is required only in the intermediate steps.

What should the simple problems at the bottom look like? We posit that they are not the traditional textbook logic building blocks, such as decoders, encoders, and multiplexers. We take inspiration from software design patterns. A software pattern is a code template for a solution to a commonly found problem in programming and computer science. Moreover, software patterns describe accepted best practices that have been found to be correct over many years of use. We have started development of a library of digital design patterns, inspired by their software counterparts. These patterns help guide students during the top-down design process, since they suggest a specific strategy: decompose the large design problem into a large set of known patterns.

In this paper, we present incipient results and assessment on the benefits of using design patterns as a learning aid in advanced digital design courses. We propose several instances of design patterns and our perspective on future development of this idea, including a library of anti-patterns and architectural patterns.

Software design patterns

It is well known that software design is a very complex problem. In fact, software engineering is the discipline that aims to create tools to manage the complexity involved in any software project of more than trivial size. Recently, so-called software patterns have emerged as a very valuable tool in this endeavor.[1] In brief, a software pattern (or software design pattern) is a template that

can be used to solve a particular, common problem. By the time a pattern has been adopted, it has been thoroughly tested and it is recognized as a best practice. The concept has been widely adopted by industry, and many different categories of patterns have emerged. For instance, there are patterns for user interface, for visualization, and for object-oriented programing, among many others. There are also patterns specific to a given language, like Java, or category of languages, such as dynamic languages.

Being widely accepted in industry, patterns have naturally been introduced in programming and computer science curricula.[2] Most interesting for our purpose, design patterns introduce a fundamental change in the way software programming and architecture is taught. Since patterns are "common solutions to common problems", students learn to, whenever possible, break down the problem they are trying to solve into pieces that correspond to a pattern. In other words, they try to formulate their solution in terms of small pieces, each of which corresponds to a pattern and, in consequence, for which a best-practice template for a solution already exists.

In a similar vein, software engineering practitioners have also identified anti-patterns: solutions that are common, and at first sight seem to make sense, but have actually been proved to be counter-productive.

Patterns in digital logic design

We propose that software design patterns can influence digital circuit design education. For many years, "top-down" and "bottom-up" have been the core design methodologies taught to students. [3,4,5,6] Top-down design is the most powerful flow, since it allows breaking up a complex problem into simple, manageable chunks. However, students who are just getting started in digital circuit design face a difficult challenge: what should the small, manageable chunks look like? Without the advantage of many years of experience to build their intuition, a top-down design may be difficult for the students to put into practice. In addition, limited lab and instructor time make any mistake in design partitioning to be very costly.

More importantly, there are few formal ways to specify the top-down design flow. Teaching is done mostly by example, where the instructor follows the design flow for a sample problem. This does not translate into efficient student learning, since the process is intuitive and not structured. When faced with a different problem that they need to solve by themselves, beginner students often do not even know where to start.

In summary, digital design methodologies lack a formal, structured aspect, which would make them more amenable to teaching and learning. We propose to adopt software engineering's idea of design patterns and adapt it to digital design. As far as we are aware, this is the first time this idea is proposed in the digital logic design education literature.

In our proposal, students would learn "common solutions to common problems", or templates that are often applicable, just as in software engineering. Then, they would try to apply the top-

down design flow, breaking up a complex problem into pieces that correspond to the patterns. Our hypothesis is that, by reducing intuition's role in the design process, this "directed top-down" methodology is easier to teach and to learn.

We should contrast this with the traditional text book approach. Most textbooks indeed present fundamental building blocks, such as multiplexers, decoders, encoders, arithmetic units, etc. Most commonly, these are used in a "bottom-up" design flow. While some of these are very common (multiplexers and counters, for example), most of them are not suitable as design patterns for beginners.

Some example patterns

We have developed a small library of patterns. We expect it to grow as we gain more experience with this methodology. A few examples of digital logic design patterns, suitable for students getting started in design, are:

The edge detector. Any design that interfaces with asynchronous, external input devices will require an edge detector. Both falling-edge and rising-edge detectors can be implemented. These can be combined to build a toggle detector.

The single-bit memory. This is a flip-flop with its output fed back into its input through a multiplexer that selects either the feedback or an external signal, depending on the value of a "strobe" or "clock enable" signal.

The multi-bit memory. This pattern builds on the single-bit memory to build flip-flop based memories of any size. It can be combined with counters, timers and other registers that require being loaded with a certain value at a certain time.

The timer. This circuit is loaded with a number and counts back from it, generating a pulse when the count reaches zero.

The LED indicator. This is common in user interfaces implemented in boards with LEDs. Two variants are the persistent indicator (stays on until turned off) and the temporal indicator (stays on while a condition is true). These can be extended to 7-segment displays.

The parallel-in, serial-out register. Together with the serial-in, parallel-out register, useful in implementing serial communications.

The pattern detector. This circuit detects when a given pattern of ones and zeroes is present in its input.

The PWM generator. This circuit can control external devices, such as motors, or the brightness of a light source.

As can be seen, the patterns are not necessarily very complicated or different from design blocks that may be introduced in traditional courses. They can be used with arithmetic circuits, state machines, shift-registers, counters, and other digital circuits. What is different in this approach is that these patterns are known to solve common problems, their best implementation is given (or better, discussed) in class, and they are used as signposts in the top-down design flow.

It is important to expose students to a large quantity of patterns (and ant patterns), in an organized way that allows them to develop their own design strategies and intuition. This is an ongoing task.

Implementation

Fully embracing design patterns in a digital design course will have an impact on the course organization. A sizable portion of the course will be devoted to describing patterns, having students get familiar with them, and providing examples of their use in larger, more complex problems.

One possible way to use design patterns in a digital design course is to have students develop their own package of patterns. As patterns are studied and discovered, students add them to a package, which in turn is included in their own designs. To make the package as flexible as possible, generics are used to make the patterns configurable. Students are asked to instantiate components from their own package and configure them via generics. As a possibility, students could be graded on the number of patterns they are able to use in their designs, and for identifying new patterns to add to their packages.

Architectural patterns

Architectural patterns are more advanced than the kind of patterns described above. These patterns don't map directly to code, but provide guidance when designing the architecture of more complex solutions. An example of this kind of pattern is when a complex logic design is divided into a "data path" section and a "control" section.

Another architectural pattern, whose complete definition we are working on, is called the "chain of events" pattern. We have identified that students have the tendency to partition a logic design into blocks that run independently of each other, and then run into problems when they need the blocks to coordinate and produce a definite result. An example would be a problem where the user pushes a button and a result is displayed. The design consists of two state machines plus some logic. The state machines run independently, and in consequence produce an incorrect result (besides being very hard to simulate and debug). The "chain of events" pattern would guide students towards a design whose blocks activate in sequence as result of user action.

These patterns would be appropriate for a senior design course.

Assessment

We have only started testing the idea of digital design patterns in our courses, and any results are preliminary. We have two assessment measures at this point. One is indirect, and is the difficulty of the capstone project in a course on hardware description languages. Most students were able to design and test a craps game simulator on an FPGA-based educational development board. The design included the user interface, the rules implementation and the random number generators. Students implemented a small package of their own with some design patterns, and were asked to use them, but were not graded on their use.

The second measure is the students' self-evaluation, specifically their own perception as digital designers. 100% of the respondents agreed or strongly agreed that the course had helped them become better digital designers.

Conclusions

We have presented a proposal to adopt and adapt some ideas from the field of software design patterns to digital logic design teaching. In particular, we believe that the top-down design methodology, as traditionally taught, is hard for students to master because it is too unstructured. The top-down flow seems to students to be open-ended, because often they do not know how the design problem should be partitioned. When provided with a set of common digital blocks, along with their best-practice implementation, students can work to express a large design problem in terms of known patterns, resulting in more efficient learning.

Bibliography

1. Gamma E. et al, "Design Patterns. Elements of Reusable Object-oriented Software", Addison-Wesley, 1994, ISBN 978-0201633610.
2. Chatzigeorgiou A. et al, "An Empirical Study on Students' Ability to Comprehend Design Patterns", Computers & Education, Vol. 51, No. 3, pp. 1007—1016, Nov. 2008.
3. Chang M., "Teaching top-down design using VHDL and CPLD", in Proc. of the 1996 Frontiers in Education Conference, pp. 514—517.
4. Comer D. J., "Application of Top-Down Principles to Digital System Design", IEEE Transactions on Education, Vol. 26, No. 4, Nov. 1983, pp. 170—172.
5. Sandige R. S., "Top-Down Design Process for gate-Level Combinational Logic Design", IEEE Transactions on Education, Vol. 35, No. 3, Aug. 1992, pp. 247—252.
6. Hadjilogiou, J., "An Innovative Top-Down Approach to Teaching Engineering Courses", in Proc. of the 2001 Frontiers in Education Conference, pp. 19—24.