Adapting Digital Design Instruction to a Programmable Logic Device Setting

Christopher R. Carroll
University of Minnesota Duluth

Introduction

Programmable Logic Devices have revolutionized the way in which digital circuits are built. Individual Small-Scale- or Medium-Scale-Integration (SSI or MSI) devices are rarely used, and in fact are becoming hard to find. Instead, FPGAs (Field Programmable Gate Arrays) and CPLDs (Complex Programmable Logic Devices) have become the standard for implementing digital systems[1]. FPGAs and CPLDs offer much higher circuit density, higher reliability, and system simplification, all of which make them very attractive to the digital designer. However, with these advantages comes a loss of visibility of the underlying circuit. The designer can "lose touch" with the circuit being designed, and the resulting hardware is really more of a computer-generated black box than it is a carefully crafted, fine-tuned design. Creativity in the design is less visible when using FPGAs or CPLDs, and "elegant" solutions to design problems are not rewarded in the same way as they are with design using SSI and MSI parts.

FPGAs and CPLDs are intended to implement solutions to digital design problems as quickly and efficiently as possible, both qualities that are important in an industrial setting. However, in an educational setting, the solution is not as important as understanding how the solution is reached, and these programmable devices automate and hide that process, making them less attractive as educational tools. Using FPGAs and CPLDs as vehicles for teaching digital circuit design requires that the instructor consciously emphasize what is being done behind the scenes by the synthesis software that configures the programmable device. Otherwise, digital design degenerates into just another programming exercise, albeit using a hardware description language rather than traditional software languages.

Furthermore, using the fixed structures of FPGA or CPLD implementations restricts design strategies and limits possible solutions. Having students design combinational circuits using "all NAND gates" or "all NOR gates" becomes pointless, because the synthesis software that configures the programmable device translates whatever implementation is specified into the standard structure implemented in the device. Combinational circuits that must avoid logic hazards (momentary "glitches" during transitions) cannot be implemented properly in current FPGA structures, and are clumsy to implement in CPLDs. Thus, some digital circuit designs cannot be mapped cleanly to programmable devices. Such designs might continue to need discrete logic gates to demonstrate to students the importance of some techniques.

Programmable Logic Devices, though attractive to the experienced designer, can be awkward and even impossible to use in certain educational settings. Digital design instructors must be aware of these limitations. They must find creative ways around the limitations, and must restrain themselves from being brainwashed by the glitz of FPGAs and CPLDs. This paper identifies techniques for maintaining the excitement and rewards of creative digital design even within the confined restrictions imposed by Programmable Logic Devices.

What is an FPGA?

A Field Programmable Gate Array (FPGA) is a component focused around a large matrix of configurable logic blocks. Each logic block implements a combinational function which is optionally captured in a latch or flip-flop register to allow sequential circuit implementation. Other structures often are present in FPGAs such as RAM memory, special-purpose logic (e.g. multipliers), and input/output circuitry, but the core logic blocks are the topic of interest here. Large digital systems are assembled by connecting logic block inputs and outputs via a programmable interconnection network within the FPGA.

Figure 1 shows a very simplified logic block in a typical FPGA. The logic block receives "n" inputs from the interconnection network in the FPGA and supplies its output to that network for use elsewhere. At the heart of the logic block is a RAM-based look-up table that generates the combinational function to be implemented in that logic block. The look-up table simply stores the truth table for the function to be implemented, and is programmed when the FPGA is configured for the target application.

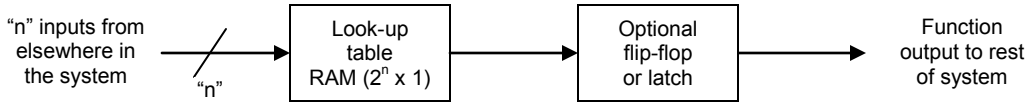

Figure 1. Simplified configurable logic block, the heart of an FPGA

By reading a circuit description defined in a hardware description language such as VHDL[2,3], synthesis software can determine the truth tables for required combinational functions, load the look-up tables with these truth tables, and configure the registered storage in each logic block as needed for the particular application.

What is a CPLD?

A Complex Programmable Logic Device (CPLD) is a component focused around a large number of macrocells. Each macrocell implements a combinational function of "n" variables which is optionally captured in a latch or flip-flop register to allow sequential circuit implementation. The difference between FPGAs and CPLDs is the structure used to implement the combinational function in the macrocell. Whereas the FPGA uses a RAM-based look-up table to store the truth table for the required function, the CPLD implements the function by ORing together implicants for the function generated by a programmable array of AND gates. The AND gates combine selected inputs (or their complements) to produce terms for the function in a typical sum-of-products structure. The result of ORing the implicants is optionally inverted and then optionally captured in a latch or flip-flop to implement a sequential circuit. As in the FPGA case, large digital systems are assembled by interconnecting the macrocells via a programmable interconnection network within the CPLD.

Figure 2 shows a very simplified macrocell in a typical CPLD. The AND gates receive their inputs from the interconnection network in the CPLD, and the macrocell output is supplied to the network for use elsewhere within the CPLD. The combinational function is produced by ORing implicants of the function in traditional sum-of-products form.

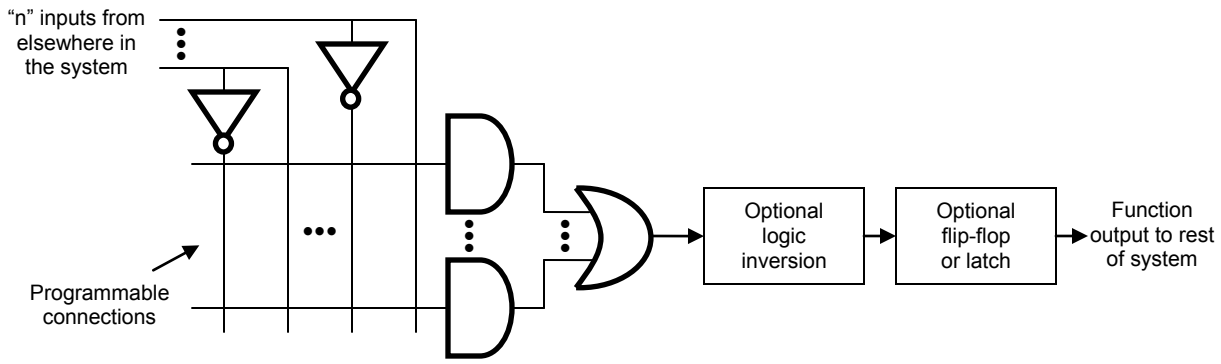

Figure 2. Simplified Macrocell, the heart of a CPLD.

Using the circuit design supplied in a hardware description language such as VHDL, synthesis software analyzes the required combinational logic to determine a minimal sum-of-products form (or product-of-sums form, if the inverting option is used) for each combinational function to be produced, and configures the AND gate connections to produce the required terms for function implementation.

Pedagogy Troubles

Regardless of whether FPGAs or CPLDs are used, the synthesis software that configures the programmable device hides all the details and challenge involved in designing a digital system. Once the system design is defined in a hardware description language, the synthesis software performs all of the optimization and minimization to configure the programmable device to implement the required system. No additional effort is required on the part of the designer. The quality of the design depends upon the quality of the synthesis software, not upon the skill of the designer who wrote the hardware description language design of the system. Clumsy, unrefined system designs are "cleaned up" by the software and produce the same final result as carefully crafted designs. If the only measure of design quality is just how many resources are used in the programmable device, poor designs and high-quality designs will score identically.

This pedagogical problem is evident even in the simplest case of implementing a single combinational function. In the FPGA case, no matter how clumsy or non-minimal the designer's function specification is, the software simply evaluates the truth table for the function, and stores it in the look-up table memory in a logic block. In the CPLD case, specification of non-minimized, redundant functions is immaterial, as the software reduces the function to minimal form. The software even determines whether sum-of-products or product-of-sums form leads to the most economical implementation, so the designer's input to the final implementation is non-existent. In the ultimate insult to a digital designer, it is even possible to simply input the truth table of the desired function(s) and the software will determine the optimal implementation. Where is the design effort to be assessed?

In current CPLD implementations, the flip-flop in each macrocell can be implemented as either a D- or a T-type flip-flop. The configuration software decides which flip-flop type yields the best result, not the circuit designer. If the designer specifies the less-optimal type, the software will

change the design, eliminating the designer's input to the final implementation. Again, the design effort expended by the circuit designer is not assessable in the final implementation.

From an instructor's point of view, trying to assess the quality of digital designs cannot rely on the result of configuring an FPGA or CPLD for the particular application. Every designer, regardless of ability, will get the same result! Instead, assessment of design quality must be based on the system description provided by the hardware description language input to the configuration software by the designer, before the synthesis software has a chance to modify it.

A possible approach that may work in assessing the quality of digital designs relies on counting the number and types of component library elements used in a design. All configuration software allows description of digital circuits using libraries of combinational and sequential components. One technique for distinguishing the quality of one design from another is simply to count the number of library elements used. Simple gates and flip-flops count as one point each. More complex, traditionally MSI, components such as counters, shift registers, four-bit adders, etc. would count as perhaps ten points each. A working design with fewer total "points" than another is clearly the better design.

Implementation Troubles

Some characteristics of FPGA and CPLD design structures complicate certain design situations. One such situation is the occasional need to implement combinational logic in non-minimal form. The synthesis software for configuring programmable devices always minimizes functions to simplest form. This sometimes prevents desired implementations from being possible.

Avoiding logic hazards is a particular case where minimizing logic is not the right thing to do. Figure 3 shows the Karnaugh map and minimal sum-of-products implementation for a function of three variables, f(a,b,c) = ab + a'c. As implemented in Figure 3, this function has a static-1 logic hazard, meaning that in at least one case, changing one input variable between values that start and end with the function value being logic 1, might generate a brief logic 0 glitch on the output during the transition. The hazard can be seen in this case by starting with a=b=c=1 and then changing a to 0. The possibility exists that the top AND gate output might become 0 before the bottom AND gate output becomes 1, which could lead to a brief 0 on the OR gate output during that transition, which is a logic hazard. In clocked systems where it is important only that function values achieve their new post-transition value before the next clock transition, hazards do not pose a problem. However, in asynchronous circuits where there is no clock to identify when to look at signals, functions must be valid at all times, including during transitions, so hazards must be avoided in such systems.

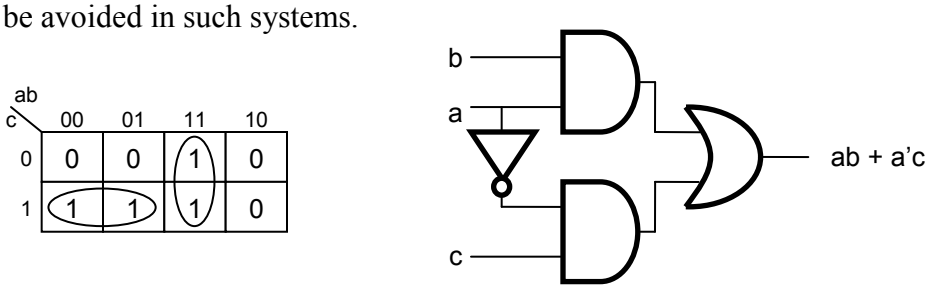

Figure 3. Simple combinational circuit that displays a static-1 logic hazard

Logic hazard problems can always be solved, but sometimes the solution requires adding redundant logic which makes the resulting circuit non-minimal. With sum-of-product implementations, the technique is to be sure that every pair of adjacent 1's in the Karnaugh map is included together in some implicant of the function. In this case, that means that the implicant "bc" must be added to the function implementation, resulting in f(a,b,c) = ab + a'c + bc, as shown in Figure 4. This is a non-minimal implementation of the function f(a,b,c), but it is hazard-free, and in situations that must avoid hazards, it solves the hazard problem.

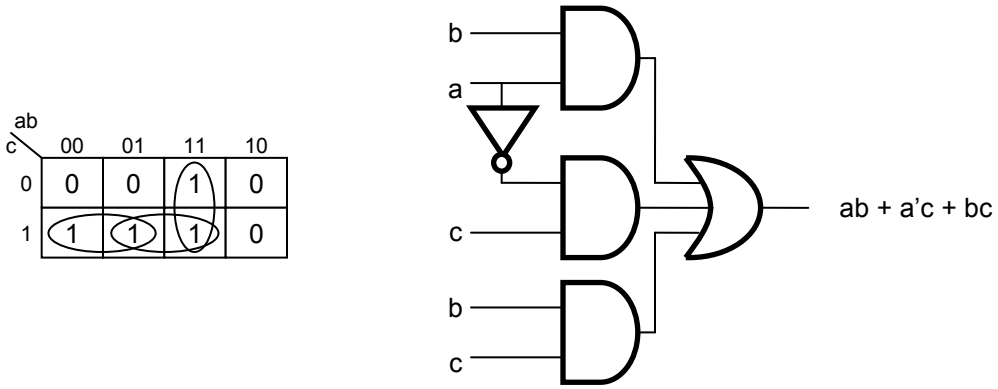

Figure 4. Non-minimal, but hazard-free implementation of Figure 3's function.

Unfortunately, programmable device synthesis software will not allow non-minimal expressions to survive. In the CPLD case, the redundant term in the expression of Figure 4 will be removed, and the CPLD implementation of the function will return to Figure 3's circuit, restoring the hazard. In the FPGA case, the truth table for the function f(a,b,c) will be determined and stored in the look-up table. This implementation essentially makes each minterm of the function a separate implicant, thus forming the function f(a,b,c) = a'b'c + a'bc + abc' + abc, which is loaded with logic hazards. Neither of these implementations is acceptable if hazards must be avoided.

So what can be done to avoid logic hazards? In the FPGA case, nothing can be done. The technique of storing the function truth table in a look-up table forces each minterm to be treated as a separate implicant of the function, which eliminates the chance to solve logic hazard problems. If logic hazards must be avoided, FPGAs cannot be used.

In the CPLD case, the synthesis software will remove redundant terms in functions. However, it is possible to separately generate each needed implicant, including redundant ones, and then separately combine them. This is spectacularly wasteful of CPLD resources, as each implicant must be generated in its own macrocell, and then an addition macrocell is required to combine the implicants to produce the hazard-free function. However, a solution is possible with the CPLD to avoid logic hazards, though very clumsy. Conceivably one could edit the netlist files produced during the configuration software execution to force redundant implicants to survive and thus eliminate logic hazards in CPLD implementations more efficiently. However, editing the netlist files (EDIC format, e.g.) is not a common skill among student digital designers, and should not be a part of the required curriculum in a digital design class. After all, the class is about designing digital circuits, not about how to coerce software tools into performing as desired.

Summary

Programmable devices such as FPGAs and CPLDs are here to stay, and have revolutionized digital design. However, using such devices as the basis for digital circuit education has some serious shortcomings. Assessing the quality of designs implemented on programmable devices requires some creativity on the part of the instructor, since the final implementation reflects more the quality of the configuration software used than it does the quality of the original design. Even worse, some designs cannot be implemented properly at all on FPGAs, and can be implemented only clumsily on CPLDs. When those designs are required in a system, programmable devices may not be the best implementation choice.

References

1. Xilinx, CoolRunner-II CPLD Family, Product Specification, Xilinx Comporation (2008).
2. Bhasker, J., *VHDL Primer*, Prentice Hall, New Jersey (1999).
3. Yalamanchili, S., *VHDL Starter's Guide*, Prentice Hall, New Jersey (1998).

Biography

CHRISTOPHER R. CARROLL received a Bachelor degree from Georgia Tech, and M.S. and Ph.D. degrees from Caltech. After teaching at Duke University, he is now Associate Professor of Electrical and Computer Engineering at UMD, with interests in special-purpose digital system design, VLSI, and microprocessor applications.