Pros and Cons of replacing discrete logic with programmable logic in introductory digital logic courses.

Kevin Nickels
Trinity University

Abstract

Digital circuit construction with small-scale integrated (SSI) and medium-scale integrated (MSI) logic has long been a cornerstone of introductory digital logic design laboratories. Recently, instructors have begun replacing these projects with designs using complex programmable logic such as programmable array logic (PLA) chips and field programmable gate arrays (FPGAs).

This paper investigates the trend of replacing the "traditional" SSI/MSI breadboarded digital logic design projects with design projects utilizing more complex programmable integrated circuits. Without a doubt, each style has its own strengths and weaknesses.

Utilizing complex programmable integrated circuits (ICs) such as PLAs and FPGAs, more interesting and involved projects can be implemented. Modern programming tools allow the specification of quite complex circuits from a graphical schematic or procedural hardware description. These specifications can be downloaded into the IC, which then functions as specified. Since much of the design is in the programming of the IC, very involved projects can be implemented without significantly increasing the wiring complexity of the project. Students don't spend hours trying to find an upside-down IC or a broken connection, and can concentrate on digital design.

The design complexity and innovation in these projects is unarguably increased over SSI/MSI projects, but this does not come without a pedagogic cost. Some of the arguments for having a laboratory course in the first place are to appeal to the sensor learning style, to expose the students to more "hands-on" and less theoretical projects, and to introduce the practical aspects of designing and implementing digital circuits. These objectives may not be met as well when moving from the SSI/MSI projects to more software-oriented projects.

Both styles of digital design projects have pedagogic strengths and appeal to particular learning styles. It is important to study the course objectives and the student mix when deciding to move projects from the traditional style of physically constructing circuits from SSI and MSI components to a new style of simulating and programming complex chips as a means of verifying digital logic designs. By doing this, we can combine the two methodologies to arrive at a course that appeals to a broad range of students, provides the "hands-on" experience some students need, and utilizes modern technologies to increase the innovation, design complexity, and interest value of implemented projects.

1 Introduction

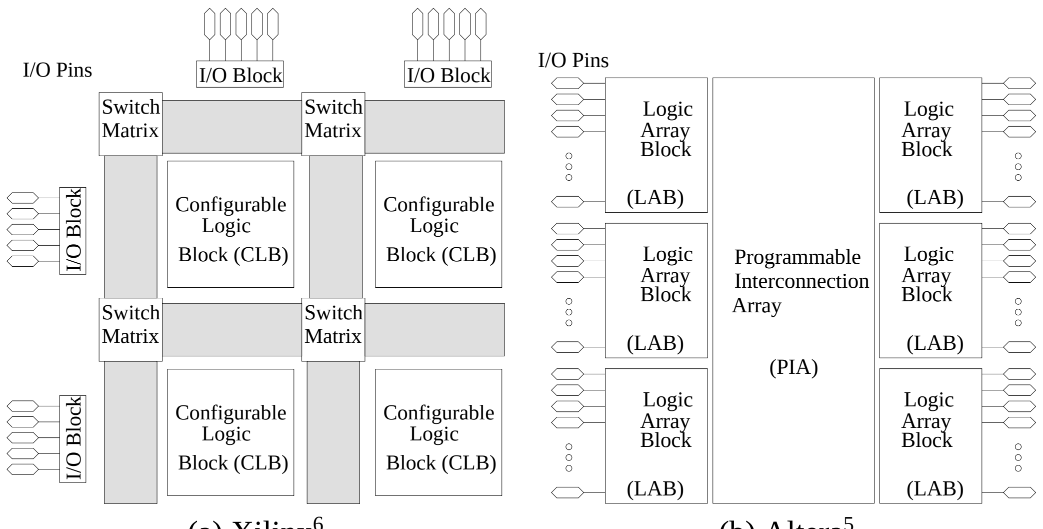
The construction of combinational and sequential digital logic circuits from discrete components, usually utilizing TTL (Transistor Transistor Logic) DIP (Dual In-Line Packaging) chips, will be familiar to anyone who has seen undergraduate electrical and computer engineering labs in the past few decades. In this type of lab, a desired behavior is often given in terms of a problem description with input/output specifications. The theoretical design is completed, and the result mapped into 7400 series chips. The final design is then constructed on a breadboard, and verified by using switches and signal generator as inputs and LEDs (Light Emitting Diodes) as outputs.

TTL SSI (small scale integration) packages contain fewer than 10 gates, and typically implement simple combinational circuits such as AND, OR, and NOT gates, or simple sequential circuits such as D or JK flop-flops. TTL MSI (medium scale integration) packages contain 10 to 100 gates, and implement more complex circuits such as small adders, decoders, and counters.

As complex programmable logic devices have become more available and affordable, many schools are incorporating them into their undergraduate laboratory curriculum.[2,4,7] In some cases, they are completely replacing the discrete TTL DIP implementations described above.

Programmable Logic Arrays (PLAs) are single chip packages (available in DIP format) that implement an array of logic: the inputs can be ANDed together in any combination, and the product terms generated can be ORed together to arrive at any SOP (Sum of Products) Boolean expression with the available number of input terms.[5] The programmer may be required to compute the interconnections, or software distributed by the chip manufacturer may compute the interconnections from Boolean expressions given by the user.

Field Programmable Gate Arrays (FPGAs) are much more complex programmable chips. There are several architectures for FPGAs available: two popular architectures will be described here. The Xilinx chips utilize an “island-type” architecture, where logic functions are broken up into small islands of 4-6 term arbitrary functions, and connections between these islands are computed. Figure 1(a) illustrates the basic structure of the Xilinx FPGA. Altera’s architecture ties the chip inputs and outputs more closely to the logic blocks, as shown in Figure 1(b). This architecture places the logic blocks around one central, highly connected routing array.



(a) Xilinx[6] (b) Altera[5]

Figure 1: Field-Programmable Gate Array Structure

Segmenting a digital design into islands of appropriate size and complexity for the chip and routing the connections between them within the connectivity constraints of the chip is certainly not something that a user can be expected to do. Manufacturers provide standalone tools or interfaces to popular CAD (computer aided drafting) packages to allow users to input the design, graphically or in a hardware description language. These tools then automatically segment the design, route the connections between the segments, and allocate chip resources for the design. The user then downloads the design from the design workstation to the chip.

Manufacturers provide developments boards for students and designers that contain a power supply, some switches for input and LEDs for output, a FPGA, and connections for downloading designs from a workstation all on one board. The switches and LEDs are often hardwired to FPGA input/output pins, and the pinouts are given.

2 Example Laboratory Project

In this popular project, the student is asked to design and implement a simple traffic light controller. It is assumed that there are two sets of lights: one for the east-west road and one for the north-south road. The street lights cycle in a fixed pattern of 45 seconds green, 3 seconds yellow, and 48 seconds red. A design requirement is that a light must always cycle through yellow before changing from green to red. It is assumed that an 0.33 Hz clock and a 45 second timer are available (The timer is referred to as signal $45s$ below). Extensions to this basic scenario include traffic sensors to detect waiting traffic and emergency sensors to detect oncoming emergency vehicles and as quickly as possible (allowing for the required yellow) give the appropriate street a green light.

2.1 Design

The digital logic design portion of this laboratory consists of three steps: the state transition diagram, the present-state/next-state (PSNS) table, and the next-state/output equations. The problem statement is analyzed, and it is found that four states suffice for the basic scenario. A state transition diagram and PSNS table are derived. From the PSNS table, Karnaugh maps for the next-state equations and output equations can be created. The Karnaugh maps yeild minimal Boolean equations for these functions, as follows:

$$\begin{array}{llll} Q_0^+ = 45s \cdot \overline{Q_0} & EWR = \overline{Q_1} & EWY = Q_1 Q_0 & EWG = Q_1 \overline{Q_0} \\ Q_1^+ = \overline{Q_1} Q_0 + Q_1 \overline{Q_0} & NSR = Q_1 & NSY = \overline{Q_1} Q_0 & NSG = \overline{Q_1 Q_0} \end{array}$$

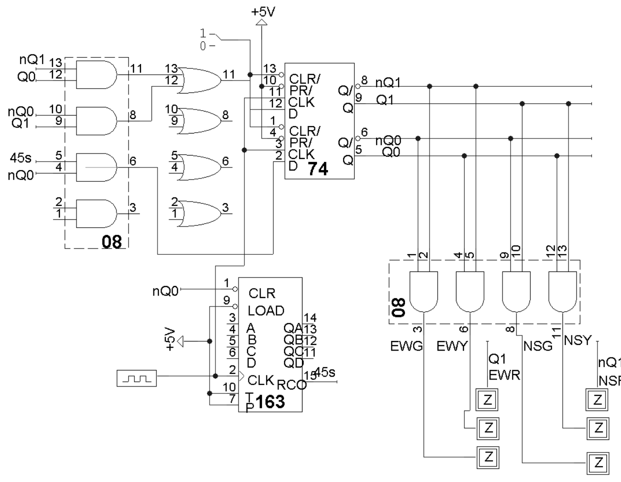
These equations complete the theoretical design of the traffic light controller.

2.2 Construction Comparison

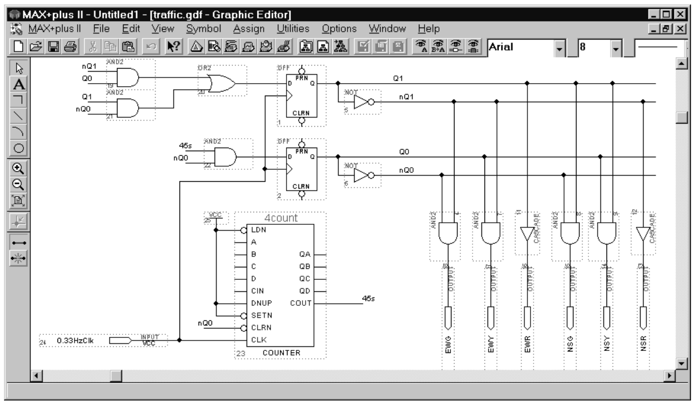
The TTL DIP implementation of this controller involves finding 7400 series chips that implement each component in the schematic, and connecting them as shown in Figure 2(a). Additionally, power and ground connections for each TTL chip are required.

This implementation requires wiring 5 separate DIPs on a breadboard, connecting 6 LEDs to monitor the outputs of the design, a clock to drive the sequential circuit, and a reset switch. This involves approximately 35 separate wires, not including power and ground connections for each chip.

The FPGA implementation involves describing the circuit in a format that the manufacturer can accept. The graphical input to Altera's MAXPLUS-II software is shown in Figure 2(b). Input and output pins that correspond to pins that are accessible on the FPGA development kit are chosen, and the design is downloaded to the FPGA.



(a) TTL Implementation



(b) FPGA Implementation

Figure 2: Alternate Implementations of Traffic Light Controller

This implementation may require wiring 6 LEDs to specified output pins of the FPGA, if a development kit is involved that does not provide hardwired LEDs. Similarly, a reset switch and clock may need to be attached to the FPGA. No other wiring is required.

3 Debugging Comparison

Common design errors include problems with reading the PSNS table, with the Karnaugh maps, and with translating the Boolean equations into a gate-level schematic. Errors in this stage will cause problems with all implementations of the design.

Common implementation errors in the TTL implementation involve placing chips in the breadboard backwards, not connecting power and ground wires, wiring to the correct pin of an incorrect chip, and missed connections. Occasionally, wires that are broken inside the insulation and faulty chips complicate this (although not as often as the students will claim).

Debugging these errors usually involves putting the machine into a known state, looking at the next-state equations to determine the desired next state, connecting a logic probe (or LED) to the input of the state flip-flops, and moving the probe backward in the next-state logic to determine the location of the incorrect logic signal. This is repeated until the machine makes the desired set of state transitions. Output debugging is similar, with the probe tracing the output logic.

Once the design is completely probed in a given state, the error can be fixed by replacing the faulty chip or wire. A fully labelled schematic with pin assignments is invaluable in debugging these circuits.

Common implementation errors in the FPGA implementation involve assigning outputs to a pin other than the one wired (by the user or the development system manufacturer) to the LED, incorrect graphical entry of the schematic, and many other software tool problems such as compilation problems, incorrect library references, or operating system problems.

Debugging these errors usually consists of a series of compile and test steps, resembling the compilation of a programming project. The only signal values available for debugging are any that have been pre-assigned (the designated outputs, plus any predefined probes). The probing process described above can be repeated, but only with a recompile for each movement of the probe. Simulation alleviates many of these problems by allowing the user to debug the logic before downloading the circuit to the FPGA. A fully debugged circuit usually only has input/output assignment problems.

4 Pedagogic Comparison

Now that the implementation and common errors of a project using these two technologies have been discussed, we move on to a discussion of the impact of these issues on the project implemented, and what the students learn by completing these projects.

The TTL implementation of a design is the closest in feel to a "direct" implementation of a gate-level design. The correspondence between the logic design, the gate-level schematic, and the TTL schematic is easily seen. The effect on the digital signals of each type of gate is also easy to see, as the signals propagate across the breadboard. For students of the "sensor" learning style,[3] this type of implementation offers an opportunity to touch chips that they can associate with the abstract concept of a gate or flip-flop.

However, the TTL implementation is also the most prone to wiring error. As all the connections between gates are made on the breadboard, there are many opportunities for a miswire or omission. Once breadboards have been used for a few years, they develop some loose connections, so that a circuit may work at times, and not work at other times. This implementation is by far the more time consuming of the two implementations discussed here *to construct*. Without close attention to wire routing, even a moderately complex TTL-based design begins to resemble a birds nest. For this reason, circuits with a chip count of approximately 7 are the most complex the author would recommend for a single afternoon introductory lab.

These errors contribute to problems in learning when the students spend an inordinate amount of time working with, and solving, problems that have little to do with the course objectives. Worse yet these problems may only have to do with laboratory hardware, and may not generalize to the practical problems encountered prototyping circuits in practice. Many students spend hours checking and rechecking their PSNS equations only to find that they have bent a pin under the package when inserting it into the breadboard. Certainly, this is not the type of lesson the students need to learn as a result of laboratory work.

The graphical implementation of a design begins to resemble a software programming or simulation project. If a good simulator is available, the bulk of the implementation time takes place in front of a computer in a test and compile cycle. Once the design is simulated to work correctly, it is downloaded to the FPGA and (barring any input/output assignment problems) is shown to work. The illustration of a working circuit is an anti-climax to many students working on FPGA-based designs: few new insights are gained by the final step. If the available simulator is not adequate, students often route logic probes from different places in the circuit to a few output pins dedicated for the debugging process. Each time these probes need to be moved, a compile and download cycle is required. The cycle time for an incremental change is much longer for a FPGA-based implementation than for the equivalent TTL-based implementation.

The errors introduced in this implementation tend to be different from those encountered in the TTL-based implementation. Miswires are more easily spotted, and wire tangles are avoided by shortcuts in the software such as named signals (signals that are tagged with the same name are considered to be connected to one another, regardless of their routing in the schematic.) Finally, a hierarchical structure is available, where a small circuit is tested and debugged, then used in a larger circuit. This modular design allows the construction of much more complex circuits than are possible with SSI and MSI based logic. Horning[4] gives example projects containing up to 72 states that have been implemented using FPGAs with Altera's design tools.

The problems encountered in a typical laboratory also do not necessarily contribute to desired learning goals. Proficiency with a computer simulator and schematic editor requires a considerable amount of time and effort. Although both are getting better as time progresses, Operating system and software tool problems may dominate the experience for a student, contributing little to their understanding of the underlying digital design tasks.

In an introductory course, these problems may preclude the exclusive use of programmable logic devices in the laboratory. Many of these arguments hold even when these devices are used in advanced courses such as those described by Chren and Zomberg,[1] but are overcome by the nature of the course and the prerequisite requirement for an introductory logic design course.

5 Discussion

While there is much to be gained by using complex programmable logic in introductory digital design laboratories, it should be implemented with caution. The use of this technology will allow an increase in the design complexity while allowing a decrease in the amount of time spent debugging wiring and other hardware implementation problems.

However, there are some serious drawbacks to using these technologies, particularly in an introductory laboratory. Students studying sequential and combinational logic for the first time often have problems “seeing” how the design will translate into input/output behavior. Students operating in the “sensor” learning style will have more difficulty grasping the effect of various changes in logic designs. The total amount of time spent debugging does not seem to change, but the types of activities the students engage in changes from finding miswires and problems with chip placement to a more computer-based approach of simulation and re-compilation.

Bibliography

1. W. A. Chren and B. G. Zomberg. Programmable logic course development in an engineering curriculum. In *Proceedings American Society for Engineering Education Annual Conference*, pages 1154–1158, 1993.

2. R. Coowar. Designing with field programmable gate arrays. In *Proceedings American Society for Engineering Education Annual Conference*, pages 853–859, 1995.

3. R. M. Felder and L. K. Silverman. Learning and teaching styles in engineering. *Journal of Engineering Education*, 77(2), February 1988.

4. D. W. Horning. Integration of digital design tools into a digital design sequence. In *Proceedings American Society for Engineering Education Annual Conference*, pages 1104–1108, 1993.

5. R. Katz. *Contemporary Logic Design*. Benjamin/Cummings, California, 1994.

6. M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals*. Prentice-Hall, New Jersey, 2nd edition, 1997.

7. A. K. Ojha. Implementation of an undergraduate laboratory on programmable logic devices. In *Proceedings American Society for Engineering Education Annual Conference*, pages 846–852, 1995.

KEVIN M. NICKELS
Kevin Nickels is an Assistant Professor of Engineering Science at Trinity University. He received the B.S. degree in Computer and Electrical Engineering from Purdue University in 1993, and received the M.S. degree in 1996 and the doctor of philosophy degree in 1998 in Electrical Engineering from The University of Illinois at Urbana-Champaign. He is currently working in the areas of computer vision, pattern recognition,and robotics. His e-mail address is `knickels@engr.trinity.edu`