A New 6.111 Laboratory Exercise: Mastermind

by

Nathan A. Mahn

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 2003

Author____

Nathan A. Mahn
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by__________________

Donald E. Troxel
Thesis Supervisor

Accepted by________________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A New 6.111 Laboratory Exercise: Mastermind
by
Nathan A. Mahn

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2003

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Mastermind is a new laboratory assignment which was developed to accelerate students' learning of digital design for 6.111 Introductory Digital System Lab. This project replaces a similar lab in an attempt to introduce students to the additional concept of major and minor finite state machines as well as previously taught concepts. Students learn these fundamental concepts by implementing the game of Mastermind. It requires them to implement a system with programmable chips using VHDL.

This project presents the students with their first design project. They will gain experience and understanding of digital design techniques over the course of several weeks as they progress through the steps required to turn a state machine concept into a realized digital system.

Thesis Supervisor: Donald Troxel
Title: Professor

Contents

List of Figures

Chapter 1: Overview

Mastermind is a laboratory assignment intended to accelerate students' learning of digital design techniques in the curriculum of 6.111 Introductory Digital System Laboratory, the undergraduate-level course in digital design. The intended use of this project is to replace the current introductory lab assignment. This project is more complicated than its current counterpart, and its creation has been focused on keeping it as simple as possible, as well as entertaining and rewarding. As the students' first project dealing with system design, it presents requirements that are simple, independent, and well-defined, which combine to create a complex digital system. These requirements emphasize the most basic, fundamental concepts of digital design.

This laboratory project outlines the board game of Mastermind[1], which the students must build. When the students are done, their lab kit will allow them to play Mastermind with their friends (and the TAs for check-off purposes).

This document describes the game of Mastermind, the design and implementation of a digital system to create the game, and the purpose of this new laboratory project. The next chapter describes just that, the purpose of replacing the current introductory project with this one, and what students should learn from it. Following that is an explanation of how the project should progress for students and staff, along with the handouts for the students and a sample solution.

[1] Mastermind is a registered trademark of Pressman Toy Corporation, by agreement with Invicta Toys and Games, Ltd., UK.

Chapter 2: Background

As with most things, understanding the purpose of this project is divided into two pieces: why and how. First, why is this project necessary? What purpose will it serve in the curriculum of 6.111? Second, what are the goals implied by the first question, and how does it intend to achieve them?

2.1 Motivation and Problem Statement

For MIT students planning a career in the field of digital circuitry, one of the staple courses is 6.111, Introductory Digital Systems Laboratory. This course takes students through the basics of digital design, from the fundamentals of gate logic to the creation of not-so-trivial circuitry.

As a laboratory class, it teaches its concepts through hands-on experience. Students spend more and more hours in lab as the projects grow in size and complexity, learning the concepts by doing, more than anything else. The laboratory projects, then, are designed to mirror the concepts presented in lectures, and are intended to allow students to develop personal experience and teach themselves in a way that provides appreciable feedback.

The first project is an introduction to the tools in the laboratory, and has the students use those tools to explore the basics of digital logic. It also introduces them to the most basic aspects of programmable logic, requiring them to program a small chip with a pre-compiled layout. The second project, then, is the first true design project. Generally speaking, it presents the students with a digital system and has them implement it in their own way. Functional requirements are given, as well as a suggested design structure, but students are left to their own creativity in accomplishing the requirements.

The logical form of this project is some sort of simple control system, and recently has been a traffic light controller. Similarly, the third project continues to push the students, as a larger and more complicated system is laid out for students to create. Audio processing is the topic with about the right level of complexity for this, and a simple pitch-shifting lab involving a microcontroller unit has recently been replaced by a DSP filtering lab which uses new FPGA boards, and accompanying software tools.

With the recent introduction of this new third laboratory project that requires students to learn and employ additional tools and concepts from earlier curriculums, either some course content must be dropped to create room, or that content must be presented in a more concise, efficient way. The underpinning motivation for this Mastermind laboratory project is to introduce the students to more of the concepts taught in lecture than the previous project, thus accelerating their learning and making the third laboratory project that much more tractable when it is presented.

In order to adhere to this motivation in a way that is tractable for beginning digital designers, the lab project must be interesting and easy to conceptualize. If the students already understand what they are creating on a conceptual level, it is that much easier for them to focus on learning the intended lessons of digital design. As such, two types of designs present themselves: a model of a system that students encounter in everyday life, or a simple game. The first could be something like a traffic light controller as is already in place in the class, or perhaps an elevator controller. The second is the more entertaining option, but without any sort of video display (something that would add unnecessary confusion to the topics being focused on) it is difficult to find a game that can be implemented.

A game that is both intricate enough to meet the intended level of complexity and yet has a simple enough interface is that of Mastermind. In this game, one person creates a sequence of colors that his or her opponent must then guess in a limited number of attempts. After each guess, the opponent is given information to narrow subsequent guesses. This information is presented as the number of correct colors in the correct positions, and also the number of correct colors in the incorrect positions.

2.2 Design Goals

As the first project that requires students to design a digital system, care must be taken to balance the desired goals of the project. First and foremost, it must be appropriate for students who have never designed such a system before. If it is too complicated or too large an undertaking, nothing will be learned. Once a system is chosen with the right level of difficulty, it must also demonstrate the appropriate lessons. If a system meets both of these criteria, it should, if at all possible, be fun and engaging for the students, and reward their hard work.

2.2.1 Introductory

While this lab is being created to aid students in learning digital design concepts more quickly, it must remain simple and at an introductory level. Its intent is to prepare the students to learn the concepts presented in the following lab project, not teach them those concepts. The students will have a total of about three weeks to work on the lab between its initial presentation and the kit check-off, which doesn't leave much time for anything that isn't necessary.

As their first design project, the students will need clear instructions and a solid structure for the system specifications. As is the tradition of 6.111 lab assignments, while the problem statement and specifications are well-defined, the solution is not. Students should be left to their own ingenuity to create the system however they see fit, assuming the final product performs as it should. While these two sentiments may seem at odds, this creates the unique experience that is 6.111.

2.2.2 Educational

The concepts that must be taught in this project, in preparation for the more difficult projects later in the term, are well-defined. The previous lab of a traffic light controller taught students how to design a digital system using the finite state machine (FSM) concept, how to interface to a static random access memory (SRAM), as well as other basic concepts of digital design such as how to program in VHDL, one of the most popular digital design languages.

This lab, then must also teach everything that the previous lab taught, with the added requirement of introducing them to the very powerful design concept known as major and minor FSMs. In preparation for the following lab, which now contains multiple complicated subsystems, this lab should introduce students to controlling multiple subsystems (via the major/minor FSM method) while keeping those subsystems simple.

2.2.3 Enjoyable

A project that isn't enjoyable is rarely approached with enthusiasm. In order to motivate students for a project in a rather time-intensive course such as 6.111, assignments should be engaging and as fun as possible. With no real drawbacks to a fun project, there is all the more impetus to choose something fun, such as a game. Surely having the students create a system that they could show their friends and play around with together for a while would provide even more motivation for successfully completing the assignment.

Chapter 3: Design of the Lab Assignment

A simple board game is a prime candidate for an assignment which is intended to teach the basics of system design. For one, there are clear rules for how the game is played, which ties well to system requirements for the project. Students who have played the game before will already have a good understanding of many of the requirements before even reading the specifications detailed in the assignment handout.

This chapter describes the rules for the game of Mastermind, and how the game translates well into a digital system. It then goes on to create a progression of work for the students to follow, setting milestones for them to reach. The end of the chapter briefly discusses the appropriateness and usefulness of the game of Mastermind as a lab assignment for beginning digital designers.

3.1 The Game of Mastermind

The game of Mastermind involves one player creating a "secret code" of a sequence of four colors. Each color in the sequence is one of six possibilities. Traditionally these colors are black, white, cyan, green, red, and yellow. [5] That secret code is hidden from the second player, who attempts to guess the code by asserting 4-color codes and adjusting future guesses based on feedback.

The feedback given for a guess is essentially two numbers: the number of correct portions of the guess, and the number of misplaced portions of the guess. That is, the first number is a count of how many of the guess' colors are the correct color in the correct spot (e.g. guess color number 2 is red, and secret color number 2 is red), and the second number is a count of how many of the guess' colors are the correct color in the incorrect spot (e.g. guess color number 4 is red, and secret color number 2 is red).

The guesser is given ten attempts to match the secret code, building on the feedback from previous guesses. If he or she fails to do so, the other player wins. As one may suspect, algorithms have been created to always win in a certain number of guesses or less, but that is not in the scope or interest of this document. What is in the interest of this document, however, is the conversion of this game into a digital system.

The structure of this game coincides well with the design goals of an introductory lab project outlined above. The order of the game, namely entering a sequence of colors, comparing it to a secret sequence, and keeping track of past guesses, allows for straightforward application of the newly desired concept of major and minor FSMs, while the actual storage of past guesses retains a need for learning how to use SRAM components, just as in this project's predecessor. All of this is within the context of VHDL, as the students must create their FSMs within programmable chips interfaced to their lab kits.

3.2 Organization of the Assignment

When the students are assigned this project, they will receive a document detailing how the game of Mastermind is played, what is expected for a kit to be considered "working," when various checkpoints are due, and some hints as to how to organize their thoughts. The handout also includes a brief explanation of what is expected in the lab report. This document is included in the following chapter.

Along with the lab document, students will be provided with bare-bones VHDL code to build their FSMs from. This code includes an entity declaration with generic inputs and outputs necessary for any FSM, but not any additional signals specific to this assignment.

While students' solutions will not all end up exactly the same, solutions can be expected to be fairly similar. Because one of the requirements is to gain familiarity with major and minor FSM interaction, all students will likely produce two or three (or more) minor FSMs to produce the game functionality, with a major FSM to control them. Designs will diverge for things like what information is communicated between subsystems, and what work will be assigned to which FSM. The lab document encourages students to follow the general structure it describes, to allow them to spend more time designing the subsystems and less time deciding what subsystems to create.

3.2.1 Design Review

When students first receive their lab handout and are introduced to the project, their first task will be to come up with a plan for the various subsystems and how they will work together. They should decide how many FSMs they will use, what each one will do, and what signals each will use.

Before creating their FSMs in VHDL, students will be expected to have a teaching assistant go over their design with them to make sure their thinking is correct and that the system they have in mind will indeed function as it should. While this places a heavy time commitment on the teaching staff for the day or two necessary to review the students' designs, this stage of design is already used in the current lab project, and will be even more important with the switch to a system with multiple FSMs in it.

Additionally, scheduling this milestone well before the final due day ensures that students will start thinking about their design early, and give them enough time to debug any unforeseen problems that crop up when they begin to implement their version of Mastermind.

After the system design is reviewed, students should have a good idea about how each of their subsystems will function, and how they will work together. This allows for additional checkpoints before completing the assignment, in the form of verifying that each individual subsystem operates as it should. While the open-ended nature of the assignment cannot provision for another round of required reviews on the students' designs, students will be encouraged to have the teaching staff verify their subsystem implementations against their original design ideas.

3.2.2 Final Check-off

Once a student has all of their subsystems working properly and working together, a teaching assistant or lab assistant will verify that the overall system plays the game of Mastermind properly. This final version of their project may or may not match their original design exactly, since bugs along the way may require modifications. This is why the internal workings of the design are irrelevant to the final check-off, and only the interface presented on the lab kit is important.

3.2.3 Lab Report

As with any digital system, a written document must accompany the completed project. While the report is generally due about a week after the final check-off, students are encouraged to start before their work is completed. The report should include explanations of the design requirements and methodology for the solution, along with diagrams and schematics of the various components. In this case, a circuit-level diagram will be required but less relevant than the flowcharts for the various FSMs. The final VHDL code students used is also expected.

3.3 Discussion

Unfortunately a late acquisition of this project as a thesis topic and the fast pace of the 6.111 curriculum prevented it from being tested as an actual lab during its creation. Extra care has been taken to make sure that the lab document is clear and helpful, and the sample solution has been thoroughly tested to find where students may have difficulties.

3.3.1 Fulfillment of Design Goals

As previously alluded to, Mastermind was specifically designed to fulfill the design goals presented earlier. Those goals include being simple enough for beginning digital designers to accomplish, focusing on concepts used as the foundation of future projects, and engaging students in an entertaining way.

As an introductory lab project, Mastermind is fairly appropriate. The choice of a simple board game as the project topic appears to be very appropriate indeed. The rules of Mastermind are very straightforward, and students should be able to concentrate on designing the system more than understanding how it should work. While the algorithm for checking a guess sequence against a secret sequence may be difficult for students unfamiliar with digital design to reduce to an optimal form, that shouldn't be necessary.

Other than that, it provides an excellent motivation for learning SRAM access techniques and development of a system using major and minor FSMs. Students will get ample experience learning how to use VHDL for most aspects of digital design. Together with the VHDL portions of problem sets, students should be well prepared for the more complex design projects in 6.111. As a replacement, this lab exercise only adds to the things students must learn in a three week period, but that was the desired result. The introduction of the major/minor FSM concept in lab project 2 is important for students so that lab project 3 is not as much of a shock for them, and its nature as a game makes it a more enjoyable vehicle for learning the concept.

Without using a video display, most games are very difficult to represent. Mastermind presented itself as one of the more appealing games with a simple enough interface to allow students to create it using LEDs. The two-player nature of the game allows students to play the game with their friends once they finish their kit, and conveniently it is turn based so that they don't need to huddle over the kit at the same time to play. This may even lead to an improved interface, based on suggestions by playing partners.

3.3.2 Areas of Concern

My main area of concern for this project is that it is definitely at the upper limits for what could be considered an introductory level project. While the various subsystems are straightforward, they are not entirely trivial and will require much of the students' time for the few weeks they are allotted to work on the project. However, I was recently informed that this term students used FPGA chips to complete the current lab 2 project, instead of the CPLD chips used in past terms. If this is the new standard for the course, it will greatly reduce the burden placed on the students for completing Mastermind.

The sample solution presented later in this paper presents an ideal solution, but also describes how the solution could be modified to fit in the smaller CPLD chips. The vast amounts of space available in FPGA chips relative to the size required for this project will allow students to concentrate on building their FSMs and not make them worry about fitting things into small physical areas. While the project can indeed be accomplished in a number of ways using CPLD chips, it does involve some extra tinkering to keep the designs small enough to fit.

Chapter 4: Lab Handout

The following pages contain the Mastermind laboratory assignment. The document was adapted from the current second lab from 6.111, which it is intended to replace, to preserve a similar style of presentation. [4]

The Mastermind handout refers to starter code which students can use as a template for creating their VHDL files. The starter code is included in Appendix B of this document.

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 – Introductory Digital Systems Laboratory

Laboratory 2 – Major and Minor Finite State Machines

Handout Date:
Design Due:
Checkoff Due:
Report Due:
Rev. Report due for Phase II:

INTRODUCTION

This laboratory exercise concerns the design and implementation of the board game Mastermind. Your implementation of this system is to be by a combination of major and minor finite state machines (FSMs).

This lab is intended to provide an introduction to the methodology for designing, building, and debugging a digital system, and creating procedures for testing design completeness.

You are to create the game of Mastermind using your lab kit. In the game of Mastermind, one person creates a secret code and another person tries to guess it based on feedback from his or her guesses. The code is a sequence of four colors, and each color in the sequence has six possible values. Since we only have 3 LED colors readily available, the most obvious way to simulate 6 different colors is by grouping LEDs into sets of three. This description will be based on 3 LEDs per color. This scheme gives us a code length of 12 bits total, 3 for each of the 4 sequence positions. The rest of this document will use the term "color" to refer to the value of such a 3-LED grouping.

The feedback system for Mastermind tells the guesser which colors are correct, and which colors are in the wrong positions. That is, if the first color of a guess matches the first color in the secret, the guesser is told that they have one correct color, but not which one. If the first color of a guess matches the third color in the secret, the guesser is told that one color is the same as one of the colors in the secret but in the wrong position, and again not which one.

The guesser has 10 attempts to guess the correct sequence, narrowing down the possibilities at each stage by carefully choosing color combinations based on the information of past guesses. If the guesser cannot determine the secret in 10 attempts, he or she loses. Figure 1 shows a mock-up of the Mastermind board.

You will somehow need to store past guesses to allow the guesser to review them. Whether you choose to store the date in a RAM chip or in signal vectors is up to you, but the guesser should be able to see which guess is currently displayed on the hex LEDs. We recommend storing the guesses in a RAM, if for not other reason than to gain experience interfacing to one.

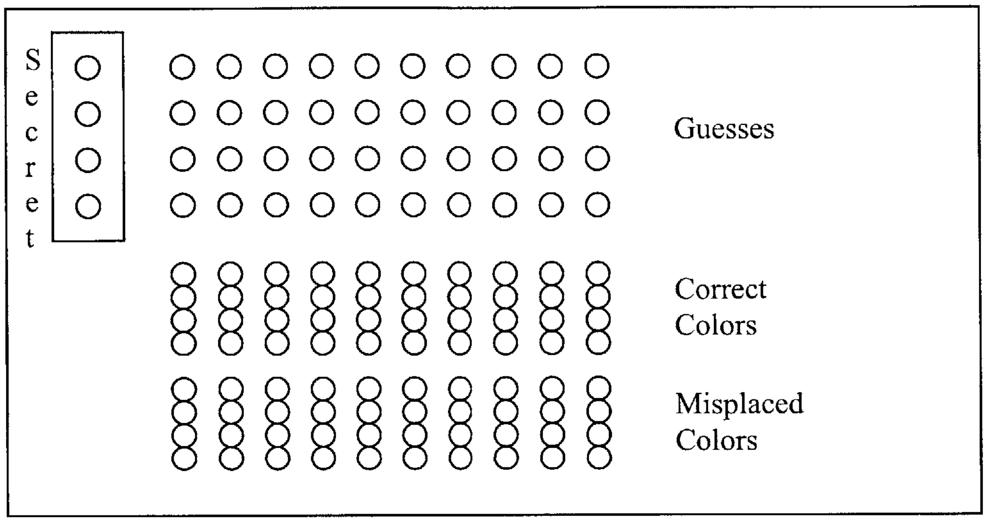

Figure 1: Mastermind Board

OPERATION

The intended purpose of this lab is to familiarize you with the basics of digital programming, and in creating major and minor FSMs. As such, you should design your system with discrete subsystems controlled by an overarching control process.

Every round of guessing involves three steps. The guesser enters a sequence, that sequence is checked against the secret, and that sequence is stored. The last two can happen in either order. The startup phase of entering the secret is just a special case of these steps. This lends itself to three minor FSMs controlled by a major FSM.

The order of operation of the system is as follows: the player choosing the secret enters a color code and stores it while the other player is checking their email or just looking away. Once the secret is stored, the LEDs blank out and the other player starts guessing. During each guess phase, the guesser has the option to review the previous guesses. It's preferable to not lose the current guess while reviewing so the guesser doesn't have to start over, but not necessary. After a guess is entered, it is checked against the secret code and the guesser is given feedback in the form of LEDs indicating how many correct colors and how many misplaced colors are in the current guess. Traditionally, four pegs were used to represent each of these numbers. If you want to use binary representations, you'll only need 6 lights total, but the savings in pins is probably not worthwhile. Finally, the guess is stored for later recall. After 10 iterations of guesses, the game enters the fail state and indicates that the game is over. If, however, the guesser finds the secret code, the game enters a won state and indicates success.

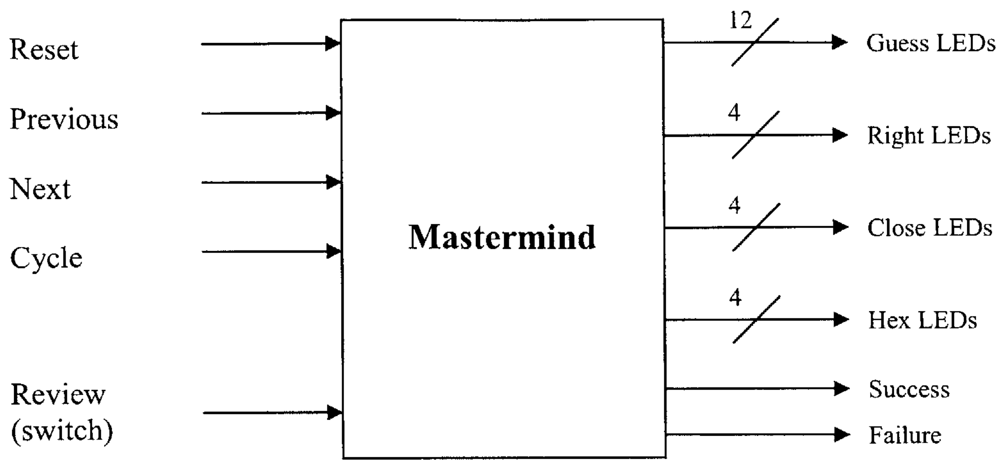

Figure 2: Mastermind Block Diagram

SPECIFICATIONS

A basic block diagram of the system is given in Figure 2. User inputs include the 4 pushbuttons and one switch. You must implement the FSMs in VHDL. If you choose to store guesses in a RAM, you will need to drive its data pins via a tristate buffer. Furthermore, the 6264 SRAM has only 8 data pins and the data vector is 12 bits. This means that the RAM will have to be accessed in 2 cycles unless you use 2 chips. Since you are already designing the storage function as an FSM, spacing the RAM access over 2 states instead of 1 shouldn't present too much of a problem.

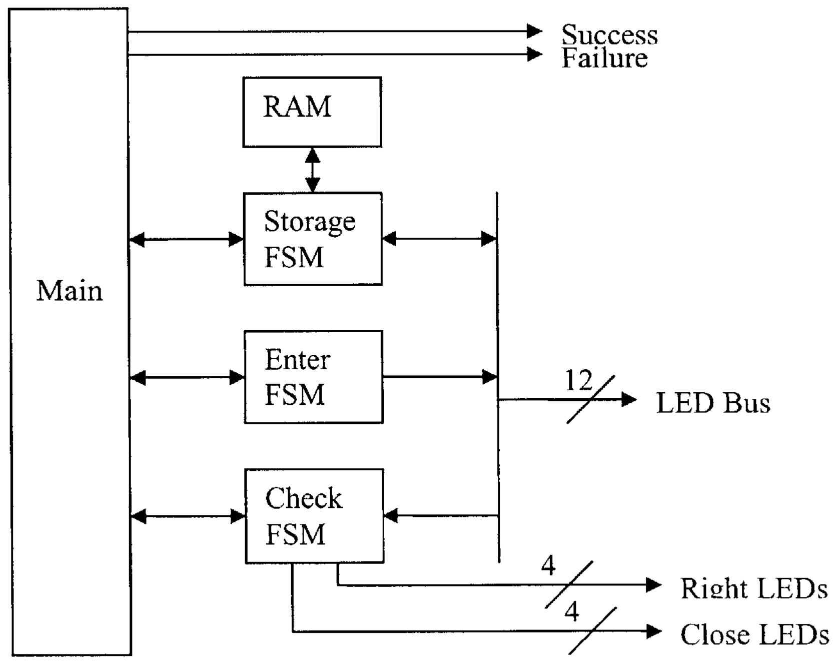

Figure 3: Basic FSM Block Diagram

Figure 3 is a general block diagram of the subsystems and their interconnections. Between the NuBus and the 50-pin connector, you have approximately 55 pins at your disposal. As you can see from Figure 3, you can't spare too many pins for replicating the data vectors. That is, all three subsystems need access to the data vector. Instead of complicating things by duplicating the data, all three can share the same 12 pins connected to the LEDs via tristate buffering. Refer to the handout "Gates, Symbols, and Busses" to understand bussing. It's very important that you implement this tristate buffering properly, or you could damage your kit and probably blow out some LEDs.

In order to ensure proper behavior, be sure to synchronize all user inputs. In chips, this involves D type flip-flops, and in VHDL it requires a simple clocked process that assigns incoming signals to internal signals every clock cycle. The only signal that should not be synchronized is reset.

Since different modules can share pins for inputs (and outputs as long as tristate buffers are used), there is some room for creativity in where user inputs go. Our suggestion is to take user inputs directly into the Enter FSM for creating a color sequence. Reviewing is slightly trickier to implement with entering review mode and staying there, and since you want to show the guesser which guess is being displayed anyway, we suggest sending user inputs into the Main FSM as well and keeping track of the number of guesses and the current guesses there, and sending an index to the Storage FSM to use as an address.

Here's a more in depth explanation of how each of the FSMs could possibly work:

Enter: All colors start out blank. When the cycle button is pressed, change the currently active color. If the sixth color is being displayed and the cycle button is pressed again, wrap to the first color, not an empty color. In all positions but the first, pressing the "prev" button changes the active position to the previous one. In the first position, it does nothing. In all positions but the last, the "next" button changes the active position to the next one, but only after the current position has a valid color. In the last position, pressing "next" sets the sequence and indicates "done" to the Main FSM, provided that the fourth color is not empty. This module can drive the LED Bus. As such, it needs a tristate buffer between the user-modifiable values and the actual output to the Bus. If at any time the "review" switch is activated, this function will be exited by the Main FSM. As such, it should only continue operation while the "go" signal is high, and likewise the Main FSM should hold this "go" signal high until it receives a "done" back or until the user chooses to review previous guesses. You may want to design all minor FSMs this way for consistency. If you want to be clever, design this module to retain the partial guess while the guesser is reviewing.

Check: You will want to store the secret code in a signal vector in this FSM, not the Storage FSM. This is because if the Storage FSM needs to send the secret across the LED Bus, the guesser could see it if they're quick. Also, since both sequences can't be on the LED bus at the same time, it will have to keep track of it somehow. This is the only minor FSM that does not drive the LED Bus. On receiving a "go" signal from the Main FSM, this module simply compares the secret sequence to the current value on the LED Bus, in groupings of 3 bits at a time, and updates the "Right" and "Close" LED sets before indicating "done". It also sends a "correct" signal back to the Main FSM as well, to end the game should the guesser choose wisely.

Storage: Perhaps the most complicated of the modules VHDL-wise, the Storage FSM has two different data busses, and therefore needs two different tristate buffers. As far as signals go, besides "go" and "done" it needs a mode indication to determine if it's writing the value on the LED Bus to the RAM, or putting the value from the RAM on the LED Bus, and it needs a guess index to address the RAM. As mentioned earlier, it takes 2 cycles to write a full data vector to a single RAM (or read one back). All this requires is two states for reading or writing, and one additional bit beyond the index from the Main FSM to determine the "high half" from the "low half" of data.

Main: Basically this subsystem just runs the other FSMs in the proper order and with the proper settings. We recommend keeping the review counters and information here. Basically, that means a close facsimile to the code in the Enter FSM, but instead of moving to previous and next color positions, it moves to the previous or next past guess for review. Review mode basically involves reading from the Storage FSM and then invoking the Check FSM so that the guesser can see the feedback for that particular guess.

Figures 4 through 7 describe a possible I/O scheme for the various subsystems. Keep in mind that basic things like synchronization are handled independently within each one. The only two that actually require synchronization are the Main FSM and the Enter FSM, since they are the only modules that users directly input to.

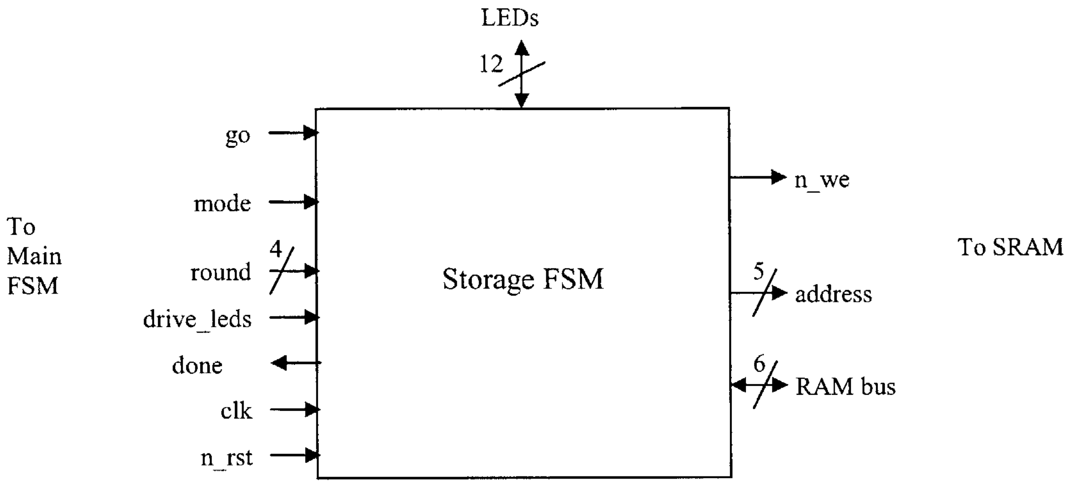

Figure 4: Storage FSM I/O Diagram

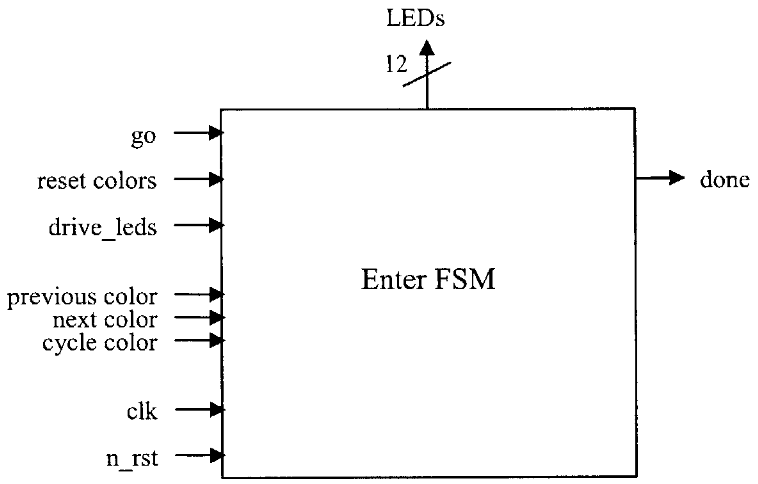

Figure 5: Enter FSM I/O Diagram

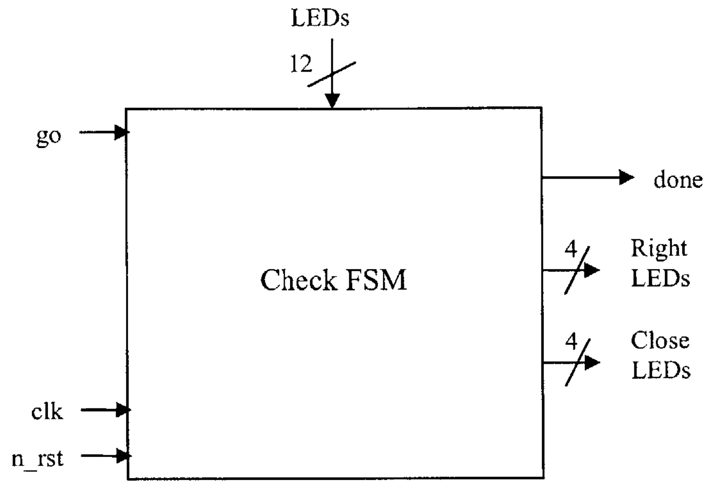

Figure 6: Check FSM I/O Diagram

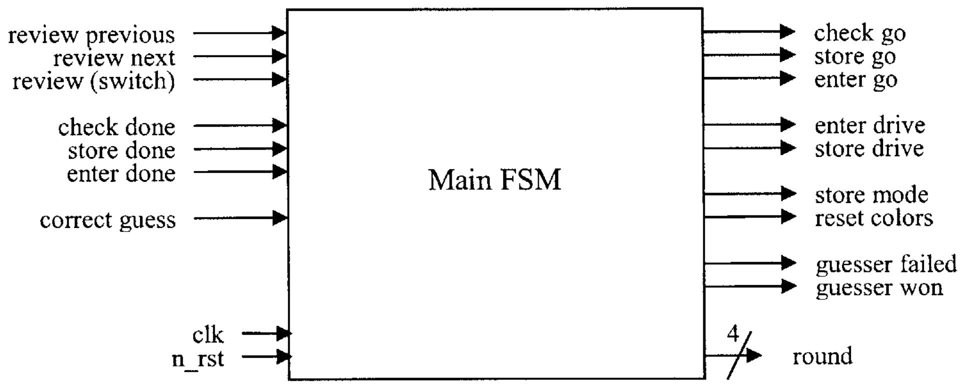

Figure 7: Main FSM I/O Diagram

WHERE TO START

A starter FSM is located in the 6.111 locker. Copy it to your locker by executing:

```
cp /mit/6.111/vhdl/lab2.X/fsm.vhd
chmod 600 fsm.vhd
```

The beauty of major and minor FSMs is the degree to which different systems are independent of one another. Keep this in mind when designing them. Apart from the common LED bus, the minor FSMs should only interface with the Main FSM if possible. Furthermore, the interaction between the Main FSM and a minor FSM should be limited to a "go" signal from the Main FSM, a "done" signal back to it, and certain operation flags and values. For example, when storing a guess to memory, the Main FSM should assert a "go" signal along with an indication that the Storage FSM should write the value on the LED Bus and an index to tell it which address to write it to.

First things first, design the FSM for each module and meet with your TA to make sure it follows the guidelines. Only then should you start writing VHDL. The first module you may want to write is the

Storage FSM, since it may be the most subtle of the three and you should make sure you have plenty of time and motivation for it.

DEBUGGING

Another benefit of completely separating the minor FSMs is the ease of testing each one. When learning to design digital systems, it's vitally important to develop good testing habits right away. The minor FSMs of this design are ideal for testing the system bit by bit, because a separate testbench can be written for each one without worrying about overly-complicated or overly-grungy patchwork to emulate the rest of the system. Even better, with well-defined interactions between the Main FSM and the minor FSMs, it's easy to see where synergistic problems, if any, might occur. That is, it's hard for errors to creep in as you build up your system when interactions are limited to "go" and "done" type signals.

Besides easy testing in simulation, it should be easy to create secondary control files to test each FSM separately on your lab kit. You can program only, say, the Enter FSM with all of the inputs and outputs sent to pins to make sure you can cycle through colors and change between the positions properly. Repeat this for each module before putting them all in together.

PROCEDURES AND REQUIREMENTS

Your kit should stand alone as a playable Mastermind game. Your hardware check-off will consist of playing a game with a TA or other staff member. Here are requirements for project check-off and also for your lab report.

1. Before proceeding with creating your FSMs, be sure to understand the concepts of synchronization and tri-state buffering, and also how to implement them correctly in VHDL.

2. Provide complete state transition diagrams for each of your FSMs, including control signals for both input and output.

3. Provide a connectivity explanation, whether in one large diagram or several well-labeled component diagrams.

4. Use VHDL to implement all FSM functionality within programmable devices. You must have your design examined and approved by a member of the teaching staff before programming any chips.

5. Demonstrate your entire system to a member of the teaching staff, showing all of its functionality. Have your diagrams and VHDL on hand to answer questions.

Laboratory Report

You are to provide a laboratory report which meets the requirements specified in the "Report Guide" handout. Your report should include the following: all FSMs, connectivity diagram(s), and VHDL source files. You should also include some text describing your design and methods of implementing it. The report should flow, be well organized, and, most importantly, be complete. Verbosity is not a requirement.

Design Notes

Data sheets for the 6264 SRAM are attached. PLEASE read the data sheet carefully as this chip is easily damaged by incorrect use (wiring). ASK QUESTIONS IF YOU ARE NOT SURE!

The 6264 has a tristate Input/Output (I/O) bus. Reread the handout "Gates, Symbols, and Busses" which pertains to bussing. The I/O bus of the 6264 MUST be driven by a tristate buffer; contact a staff member if you need help creating one in VHDL.

Tristate bus contention occurs when two (or more) drivers are active at the same time. The 6264 tristate output is enabled when the /OE output is asserted low, the /CS is asserted low, and the /WE line is high. While it is true that many logic designers allow tristate bus contention to occur for short times (due to chip delays), it is not a good idea. For this laboratory exercise, you are to ensure that NO tristate bus contention can occur. The actual write pulse is the AND of both the /CS and the /WE asserted low. It is essential that the address lines to the SRAM not change when the write pulse is active. Otherwise you may write to multiple locations!

While the 6264 is advertised as a static RAM, a memory cycles is actually initiated whenever ANY address line changes. Thus, the address lines may NOT be tristated whenever the /CE is asserted, as the internal timing circuitry is actuated by noise on the HI-Z address lines.

One way to ensure both that tristate bus contention does not occur and that the address lines do not change when the write pulse is active is to connect the system clock, /CLK, to the chip select pin; see Figure 8. The Address Lines do not change until after the rising edge of /CLK. The /WE line can then be provided by your Store FSM. As long as the /WE line is low prior to (or concurrent with) the chip select being asserted, then the SRAM will not drive the I/O pins. The control line to the tristate gate connected to the switches can also be an output of your FSM, but it should also be gated with the system clock.

During T1, data from the SRAM will appear on the I/O pins, and during T2 the data from the Store FSM will appear at the I/O pins for storage. The tristate code should coincide with the /WE pulse to the SRAM.

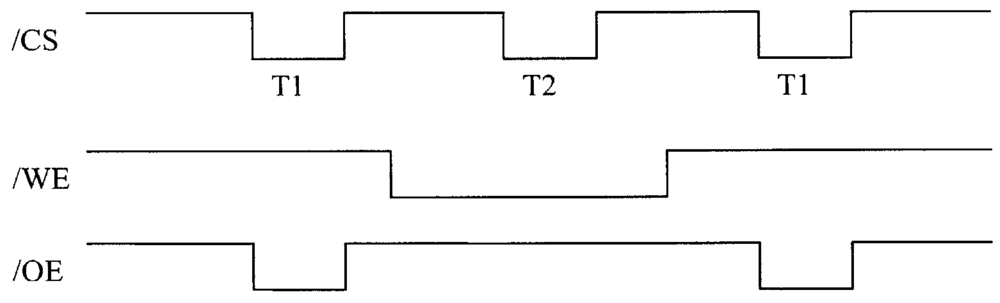

Figure 8: Example Timing Diagram for SRAM I/O

Chapter 5: Sample Solution

This chapter presents a solution to the lab project described in the previous chapter. It is, of course, only one possible solution. It follows the suggestions of that description, and also includes comments how the various FSMs could be modified to account for fitting them into CPLD chips, which requires some changes to how the FSMs are distributed. Figure 5-1 is a general block diagram for the system.

Because the traditional Mastermind colors are impossible to display with LEDs, each color is represented by a set of three LEDs: one red, one yellow, and one green. This means that the LED bus is a 12 bit vector, with each color represented by three bits. There are a total of eight possible values for three bits, though, so only values one through six are used. When the colors are reset, they are set to zero until a player chooses one. When the active color has the value of six (110) and the cycle button is pressed, the code entry FSM automatically wraps to the value of one (001) to prevent an invalid color.

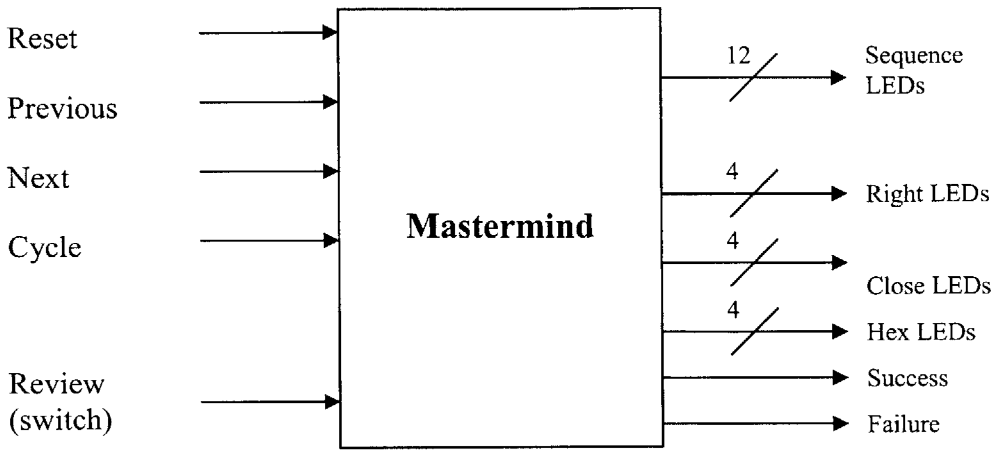

Figure 5-1: Mastermind Block Diagram

5.1 Storage FSM

Figures 5-2 and 5-3 refer to the block diagram and flowchart for the storage FSM, respectively. This component can read and drive the LEDs used to display the color sequences, and also interfaces to a single HM6264 SRAM storage chip. Both reading from and writing to the SRAM involves two clock cycles, one for the upper half of the LED vector values, and the other for the lower half. This is due to the SRAM chip only having eight input/output (I/O) pins, while the complete LED vector is 12 bits long.

When the "go" signal from the main control FSM is asserted to this subsystem, it checks whether it is supposed to read from the SRAM and place that value on the LEDs or read the value of the LEDs and store that value in a location on the SRAM. The address for reading and writing is generated in the Main FSM, though the Storage FSM could also handle that duty.

This subsystem actually requires two tri-state buffers: one for the LED bus which it shares with the code-entry FSM, and one for the SRAM bus which it must read from and write to. The first buffer is controlled by the Main FSM, while the second is controlled internally based on which control loop it goes through.

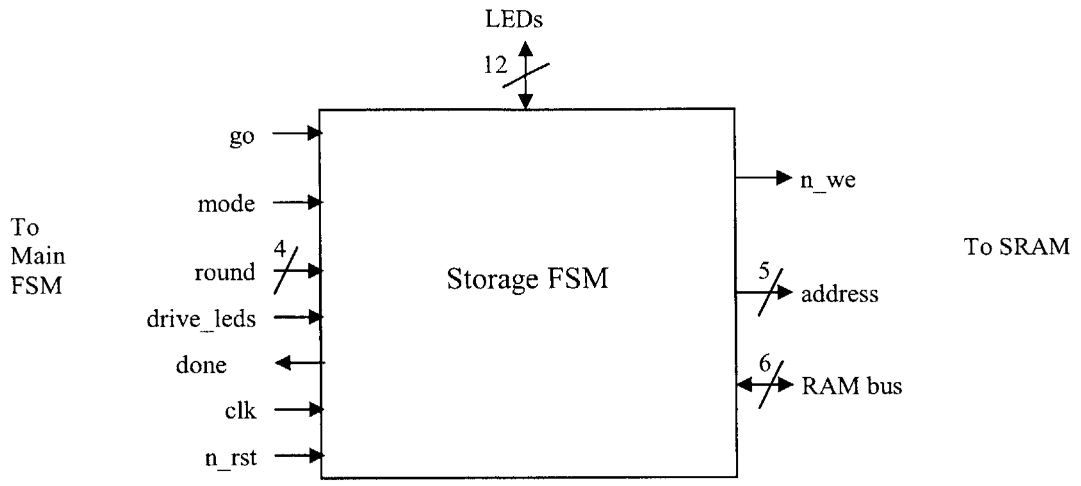

Figure 5-2: Storage FSM Block Diagram

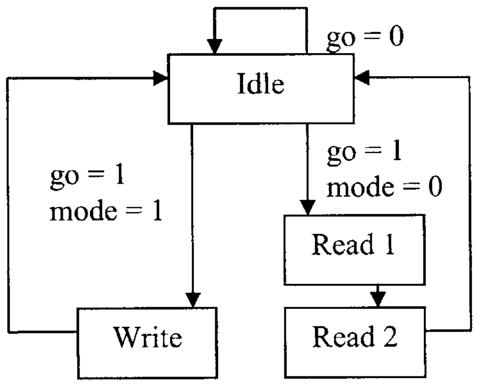

Figure 5-3: Storage FSM State Transition Diagram

Note that some functions are absorbed into the idle state. Namely, when the system receives a "go" signal and checks the read/write mode flag, it begins SRAM access by driving the new address received from the control FSM and asserts the write signal if appropriate.

Also, if this project is created within a set of four CPLD chips, as is available to the students, this FSM must be absorbed into another of the FSMs. As will be discussed later, the FSM that checks a guess sequence against the secret is by far the largest, and will probably need to span two chips. The Storage FSM is the best candidate for removal, since it is only a small number of states. The steps the author took to absorb the storage FSM placed the function in the control FSM, and required two SRAM chips instead of one. By controlling the output enable pins of the SRAM chips along with the write enable pins, control of the LEDs can still be designated to either the guess entry FSM or to reading back a past guess. Another possibility is placing this FSM in a CPLD chip with the smaller portion of the Check FSM, which is the part that counts the number of correct colors in the correct positions of the guess. This approach would probably be preferable, as it would retain the separation of subsystem functionality.

5.2 Enter FSM

The Enter FSM is the other subsystem that can control the LED bus. It allows players to enter both the secret sequence and the guess sequences into the LEDs. When this subsystem receives a "go" signal from the major FSM, it allows the user to adjust the colors of the four positions. The inputs chosen for this solution involve a "next" signal, a "previous" signal, and a "cycle" signal. When the "cycle" button is pressed, the active color position changes to the next possible color. When the "previous" button is pressed, the active color position is moved to the previous spot, and stays the same if already on the first color. Similarly, when the "next" button is pressed, the active color position is moved to the next spot, but only if the current position holds a valid color. Figures 5-4 and 5-5 display the Enter FSM's block diagram and state transition diagram, respectively.

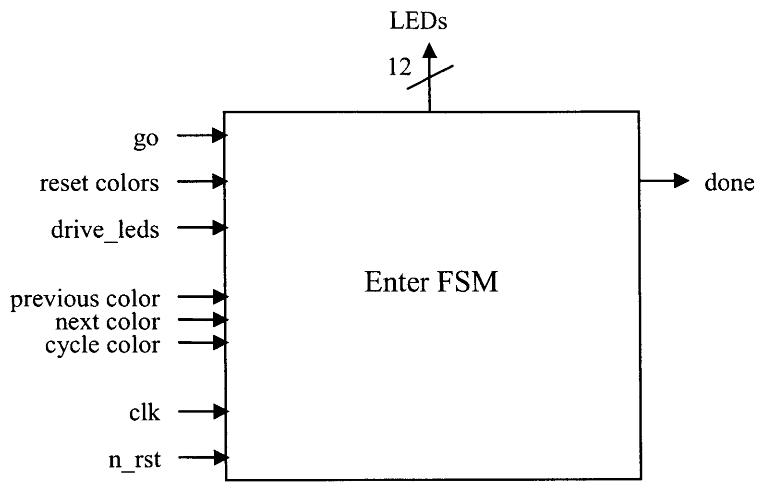

Figure 5-4: Enter FSM Block Diagram

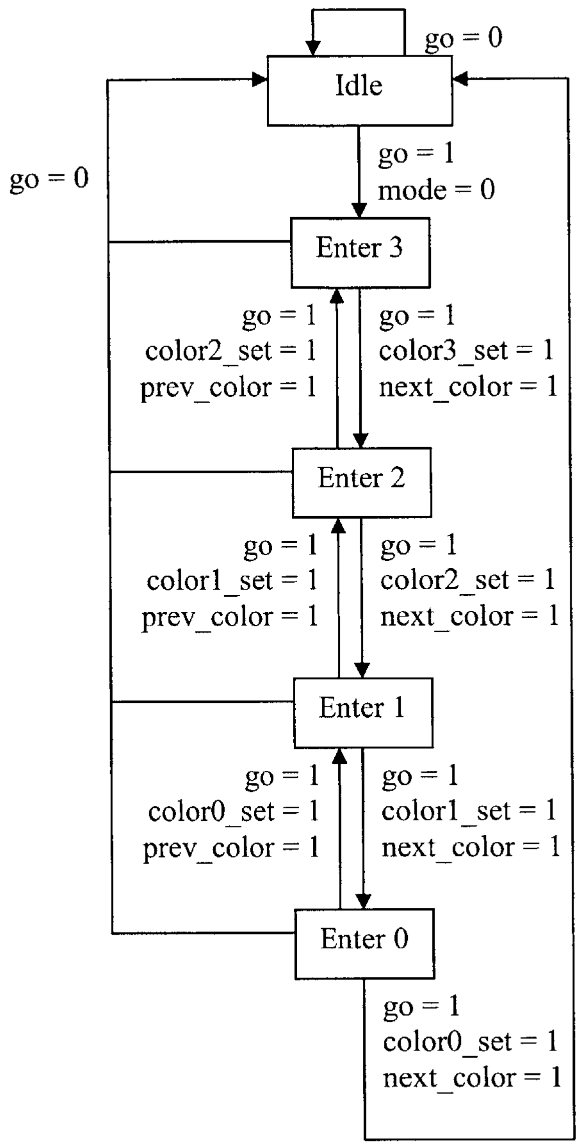

Figure 5-5: Enter FSM State Transition Diagram

As a convenience, the Enter FSM only resets its color sequence when the Main FSM asserts a "reset colors" signal. This allows the guesser to review past guesses without requiring them to re-enter their current guess.

Also, the Enter FSM returns to an idle state whenever its "go" signal is not asserted. This is necessary to allow players to review past guesses while entering the next guess. In practice, all three minor FSMs function this way, but only this one requires it.

5.3 Check FSM

Perhaps the most complicated of the minor FSMs is the one that checks a guess against the secret code. It must check each color of one of the sequences against every color of the other sequence, resulting in a total of 16 comparisons of three-bit numbers. This solution stores each of these 16 comparisons in a flag signal, and combines this with two "masks" to determine which color comparisons to check. These "masks" keep track of which colors have already been matched up as either correct or close.

First, it checks for any exact matches. If any are found, it updates each mask to prevent future use of that color. Next, when it checks for colors in incorrect positions, it progresses through each secret color and looks for the first guess color that both matches and is unused by checking the masks and the comparison flags.

Figures 5-6 and 5-7 describe the Check FSM's I/O scheme and state transitions, respectively. This subsystem has very few input signals from the major FSM. Early designs included a "store" signal because this subsystem is the best candidate for storing the secret sequence, but this signal was removed and instead the FSM stores the sequence on the LED bus in an internal vector the first time it is given a "go" signal.

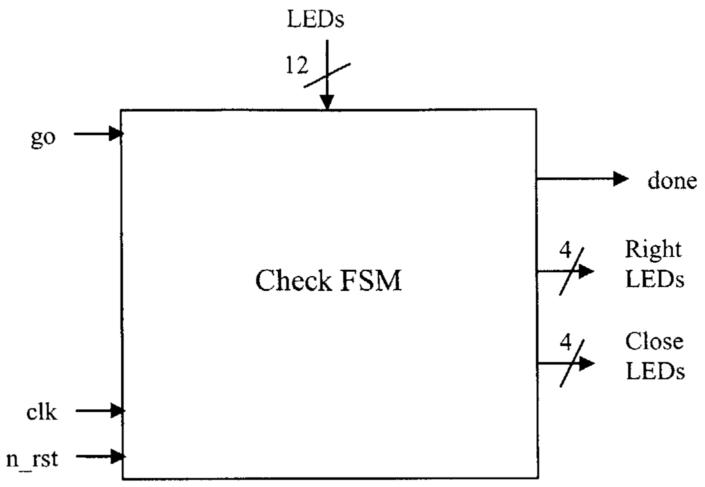

Figure 5-6: Check FSM Block Diagram

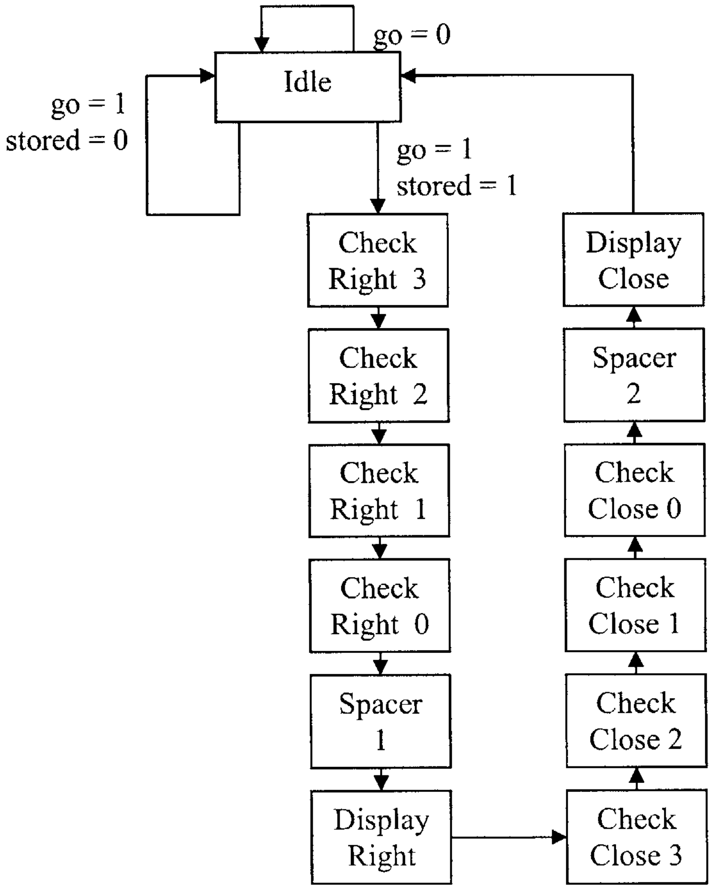

Figure 5-7: Check FSM State Transition Diagram

The spacer states are necessary because space was saved by only creating one adder for each of the feedback numbers. As a result, each state sets an "increment" flag to add to one of the numbers. This creates a single clock delay in achieving the final value, and since the states don't need to wait for anything, each takes only one clock cycle.

If this subsystem needs to be split to fit into CPLD chips, the "down" path in the above diagram can be separated from the "up" path. An additional four bits must be used as I/O to communicate the state of the masks. In this case, the "Close FSM" uses the mask sent from the "Right FSM" as its starting point for both the "secret mask" and the "guess mask" in its checks. As can be seen in the accompanying VHDL code in Appendix A, the Close FSM is substantially more complex than the Right FSM. The smaller amount of space taken up by the simpler Right FSM then leaves space for the storage room to share a CPLD chip with.

5.4 Main FSM

The Main FSM provides the "go" signals to the three minor FSMs in the proper order to play the game of Mastermind. When the system is reset, it starts the Enter FSM to allow the first player to enter the secret code. This code is then stored inside the Check FSM and the LEDs are cleared for the first guess. After the first guess is entered, the Main FSM first runs the Check FSM to provide feedback to the guesser, and then runs the Storage FSM to hold the guess for later review. These two steps can be taken in either order. After both finish, the colors are reset again and system returns to the Enter FSM again.

During the guess entrance stage, if the review switch is set the "go" signal to the Enter FSM is immediately dropped and control of the LEDs is given to the Storage FSM, which retrieves the specified guess. The Check FSM is then run to re-check this previous guess against the secret so the guesser can see the feedback for that particular guess. Note that if the review switch it set during the first guess, nothing happens and the system remains in the Enter FSM since there are no past guesses to review.

The guesser wins when the Check FSM asserts a "correct" signal along with its "done" flag. When this occurs, the system enters an infinite loop in a win state and asserts the "won" output to an LED. Similarly, if the Main FSM's guess counter hits ten without receiving a "correct" signal, it enters a different infinite loop in a lose state and asserts the "failed" output to a different LED.

Figures 5-8 and 5-9 respectively show the I/O diagram and system flow of the Main FSM. As is evident in Figure 5-9, the flow of the "new guess" path and the "review guess" path are very similar. The only real difference is where the value on the LED bus is driven from.

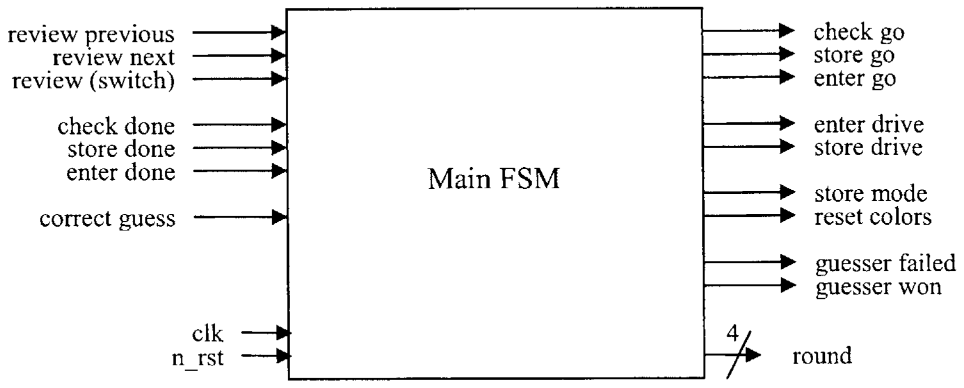

Figure 5-8: Main FSM Block Diagram

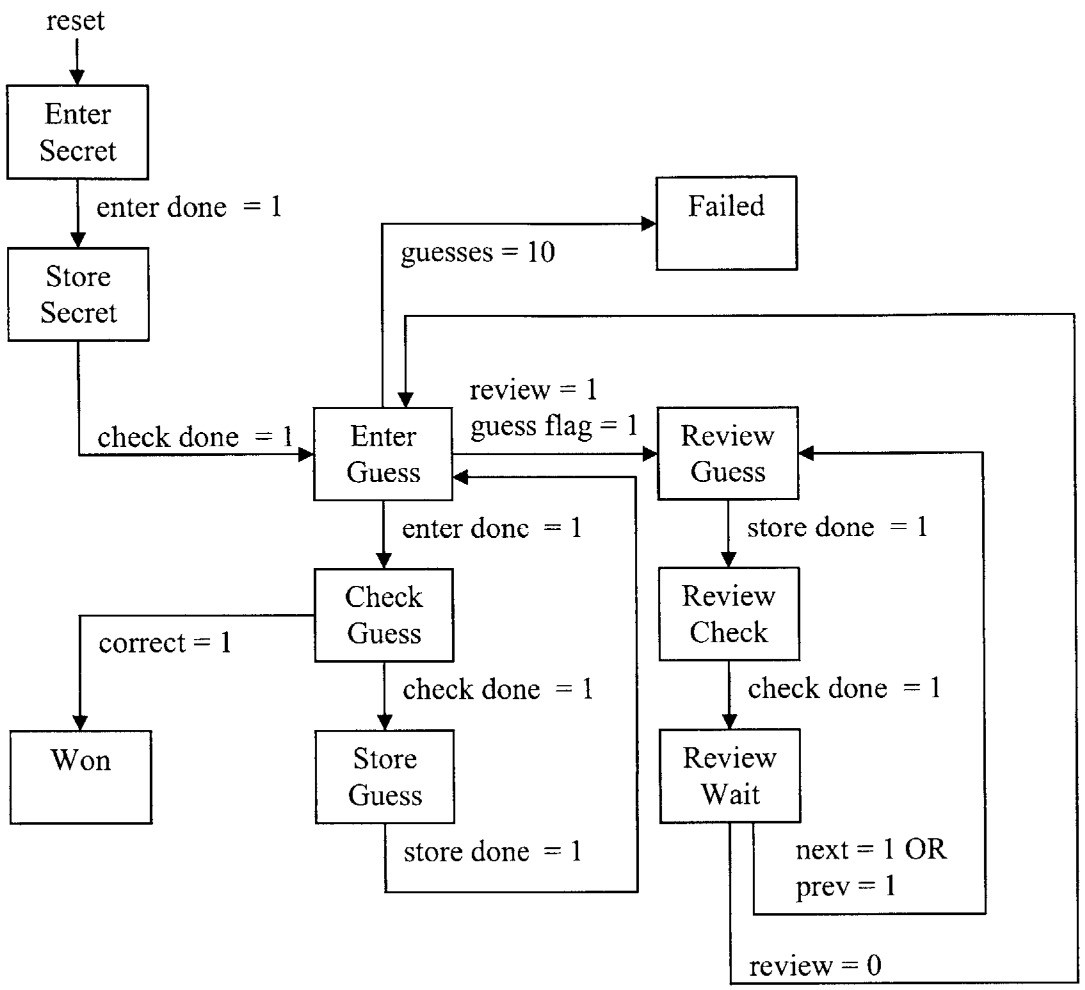

Figure 5-9: Main FSM State Transition Diagram

As discussed in the Storage FSM section, the Main FSM can be modified to control the SRAM directly. In this case, the Main FSM's output "round" is used to directly address the two SRAM chips needed. It also controls the write enable for the chips, and drives their output enable pins to replace the "store drive" signal it would otherwise send to the Storage FSM.

Chapter 6: Conclusion

Mastermind provides the desired lessons in VHDL and state machine design. With a recent update to the third lab project for 6.111, this moderate increase in the complexity to the second lab project should rebalance the learning curve for the course.

As a simple board game, Mastermind is easily understandable in concept, and thus leaves students to spend their time understanding design concepts, learning VHDL, and debugging their designs. Whether students have played the game before or not, all should be able to dive right into the course concepts. Those concepts, along with the other goals set out for this project, are satisfied in the system described in this document. Hopefully students will find the assignment enjoyable on many levels, not only the satisfaction of learning new things, but also creating a game they can play with their friends.

Though this project is significantly more in-depth than its predecessor, such a change is important to keep up with the changing goals of the class, and its move towards more VHDL-oriented design. If students stay in contact with the staff and work diligently in the time they are allotted, they should not have any problem finishing.

Appendix A: Code Used in Sample Solution

A.1 Storage FSM

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity storage_fsm is
  port ( clk              : in std_logic;
         n_rst            : in std_logic;
         -- initiate process
         go               : in std_logic;
         -- mode
         --     0 = read
         --     1 = write
         mode             : in std_logic;
         -- which round are we reading/writing (0 through 8)
         round            : in std_logic_vector(3 downto 0);
         -- full data vector to/from the led data bus
         leds             : inout std_logic_vector(11 downto 0);
         drive_leds       : in std_logic;

         -- indication that process is done or response is valid
         done             : out std_logic;
         -- write enable to RAM
         n_we             : out std_logic;
         -- address sent to RAM (0 through 8 plus high/low half)
         addr             : out std_logic_vector(4 downto 0);
         -- 6 bit I/O bus between CPLD and RAM (high or low half of data)
         rambus           : inout std_logic_vector(5 downto 0)
    );

end storage_fsm;

architecture behavioral of storage_fsm is
  type StateType is (idle, write, read1, read2);

  signal state           : StateType;
  signal n_we_sig        : std_logic;
  signal to_ram          : std_logic_vector(5 downto 0);
  signal to_leds         : std_logic_vector(11 downto 0);

  constant read_mode     : std_logic := '0';
  constant write_mode    : std_logic := '1';


begin  -- behavioral

  n_we <= n_we_sig;

  -- tristate logic to send/receive data from the RAM
  rambus <= to_ram when n_we_sig = '0' else (others => 'Z');
  -- tristate logic to share the leds with the enter fsm
  leds <= to_leds when drive_leds = '1' else (others => 'Z');

  state_clocked: process(clk, n_rst)
  begin
    if (n_rst = '0') then
      -- reset everything here
      n_we_sig <= '1';
```

```
        state <= idle;
        done <= '0';
        addr <= (others => '0');
        to_ram <= (others => '0');
        to_leds <= (others => '0');
      elsif rising_edge(clk) then
        -- default signals to be innocuous
        n_we_sig <= '1';
        addr <= (others => '0');
        to_ram <= (others => '0');
        done <= '0';
        case state is
          when idle =>
            -- default to remaining in idle state
            state <= idle;
            -- action requested
            if go = '1' then
              -- for either mode, we can start the RAM access now
              if mode = write_mode then
                state <= write;
                addr <= round & '0';
                n_we_sig <= '0';
                to_ram <= leds(5 downto 0);
              else
                state <= read1;
                addr <= round & '0';
              end if;
            end if;
          when write =>
            n_we_sig <= '0';
            addr <= round & '1';
            to_ram <= leds(11 downto 6);
            done <= '1';
            state <= idle;
          when read1 =>
            addr <= round & '1';
            to_leds(5 downto 0) <= rambus;
            state <= read2;
          when read2 =>
            to_leds(11 downto 6) <= rambus;
            done <= '1';
            state <= idle;
          when others =>
            state <= idle;
        end case;
        if go = '0' then
          n_we_sig <= '1';
          addr <= (others => '0');
          to_ram <= (others => '0');
          done <= '0';
          state <= idle;
        end if;
      end if;
    end process state_clocked;

  end architecture behavioral;
```

A.2 Enter FSM

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity enter_fsm is
  port ( clk                : in std_logic;
         n_rst              : in std_logic;
         -- initiate process
         go                 : in std_logic;
         reset_colors       : in std_logic;
         drive_leds         : in std_logic;
         -- user controls
         next_color         : in std_logic;
         prev_color         : in std_logic;
         cycle_color        : in std_logic;

         done               : out std_logic;
         -- full data vector to the led data bus
         leds               : out std_logic_vector(11 downto 0)
    );

end enter_fsm;

architecture behavioral of enter_fsm is
  type StateType is (idle, enter3, enter2, enter1, enter0);

  signal state              : StateType;

  signal color3             : std_logic_vector(2 downto 0);
  signal color2             : std_logic_vector(2 downto 0);
  signal color1             : std_logic_vector(2 downto 0);
  signal color0             : std_logic_vector(2 downto 0);
  signal s_next_color       : std_logic;
  signal s_prev_color       : std_logic;
  signal s_cycle_color      : std_logic;
  signal s_next_color_d     : std_logic;
  signal s_prev_color_d     : std_logic;
  signal s_cycle_color_d    : std_logic;
  signal color3_set         : std_logic;
  signal color2_set         : std_logic;
  signal color1_set         : std_logic;
  signal color0_set         : std_logic;


begin  -- behavioral

  -- sync input signals to the clock
  sync_signals: process(clk)
  begin
    if rising_edge(clk) then
      -- sync from user
      s_next_color  <= next_color;
      s_prev_color  <= prev_color;
      s_cycle_color <= cycle_color;
      -- make delay pipe for edge trigger
      s_next_color_d  <= s_next_color;
      s_prev_color_d  <= s_prev_color;
      s_cycle_color_d <= s_cycle_color;
    end if;
  end process sync_signals;

  -- tristate logic to share leds with the storage fsm
  leds <= (color3 & color2 & color1 & color0) when drive_leds = '1' else
(others => 'Z');

  color3_set <= (color3(2) or color3(1) or color3(0));
  color2_set <= (color2(2) or color2(1) or color2(0));
```

```
color1_set <= (color1(2) or color1(1) or color1(0));
color0_set <= (color0(2) or color0(1) or color0(0));

state_clocked: process(clk, n_rst)
begin
  if (n_rst = '0') then
    -- reset everything here
    state <= idle;
    done <= '0';
    color3 <= (others => '0');
    color2 <= (others => '0');
    color1 <= (others => '0');
    color0 <= (others => '0');
  elsif rising_edge(clk) then
    -- default signals to be innocuous
    done <= '0';
    case state is
      when idle =>
        -- default to remaining in idle state
        state <= idle;
        -- action requested
        if go = '1' then
          state <= enter3;
        end if;
      when enter3 =>
        state <= enter3;
        if (s_next_color_d = '0' and s_next_color = '1'
            and color3_set = '1') then
          state <= enter2;
        elsif (s_prev_color_d = '0' and s_prev_color = '1'
               and color3_set = '1') then
          -- stay in 3 because there's no previous color
        elsif (s_cycle_color_d = '0' and s_cycle_color = '1') then
          if color3 = "110" then
            -- if color3 is all on, wrap to one on
            color3 <= "001";
          else
            -- otherwise increment to next color
            color3 <= color3 + "001";
          end if;
        end if;
      when enter2 =>
        state <= enter2;
        if (s_next_color_d = '0' and s_next_color = '1'
            and color2_set = '1') then
          state <= enter1;
        elsif (s_prev_color_d = '0' and s_prev_color = '1'
               and color2_set = '1') then
          state <= enter3;
        elsif (s_cycle_color_d = '0' and s_cycle_color = '1') then
          if color2 = "110" then
            -- if color2 is all on, wrap to one on
            color2 <= "001";
          else
            -- otherwise increment to next color
            color2 <= color2 + "001";
          end if;
        end if;
      when enter1 =>
        state <= enter1;
        if (s_next_color_d = '0' and s_next_color = '1'
            and color1_set = '1') then
          state <= enter0;
        elsif (s_prev_color_d = '0' and s_prev_color = '1'
               and color1_set = '1') then
          state <= enter2;
        elsif (s_cycle_color_d = '0' and s_cycle_color = '1') then
          if color1 = "110" then
            -- if color1 is all on, wrap to one on
            color1 <= "001";
          else
            -- otherwise increment to next color
```

```
              color1 <= color1 + "001";
            end if;
          end if;
        when enter0 =>
          state <= enter0;
          if (s_next_color_d = '0' and s_next_color = '1'
              and color0_set = '1') then
            -- this signal is "set this code now"
            state <= idle;
            done <= '1';
          elsif (s_prev_color_d = '0' and s_prev_color = '1'
                 and color0_set = '1') then
            state <= enter1;
          elsif (s_cycle_color_d = '0' and s_cycle_color = '1') then
            if color0 = "110" then
              -- if color3 is all on, wrap to one on
              color0 <= "001";
            else
              -- otherwise increment to next color
              color0 <= color0 + "001";
            end if;
          end if;
        when others =>
          state <= idle;
      end case;
      if go = '0' then
        state <= idle;
        done <= '0';
      end if;
      if reset_colors = '1' then
        -- set things up to enter a color code
        color3 <= (others => '0');
        color2 <= (others => '0');
        color1 <= (others => '0');
        color0 <= (others => '0');
      end if;
    end if;
  end process state_clocked;

end architecture behavioral;
```

A.3 Check FSM

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity check_fsm is
  port ( clk              : in std_logic;
         n_rst            : in std_logic;
         -- initiate process
         go               : in std_logic;
         -- full data vector to/from the led data bus
         leds             : in std_logic_vector(11 downto 0);

         -- indication that process is done or response is valid
         done             : out std_logic;
         -- feedback LEDs to tell guesser how they did
         right_leds       : out std_logic_vector(3 downto 0);
         close_leds       : out std_logic_vector(3 downto 0)
    );

end check_fsm;

architecture behavioral of check_fsm is
  type StateType is (idle, check_right3, check_right2, check_right1,
               check_right0, disp_right, check_close3, check_close2,
               check_close1, check_close0, disp_close, spacer1, spacer2);

  signal state            : StateType;
  signal secret           : std_logic_vector(11 downto 0);
  signal close_count      : std_logic_vector(2 downto 0);
  signal right_count      : std_logic_vector(2 downto 0);
  signal clear_counts     : std_logic;
  signal inc_right        : std_logic;
  signal inc_close        : std_logic;
  signal secret_mask      : std_logic_vector(3 downto 0);
  signal led_mask         : std_logic_vector(3 downto 0);
  signal stored           : std_logic;
  signal match_3_3        : std_logic;
  signal match_3_2        : std_logic;
  signal match_3_1        : std_logic;
  signal match_3_0        : std_logic;
  signal match_2_3        : std_logic;
  signal match_2_2        : std_logic;
  signal match_2_1        : std_logic;
  signal match_2_0        : std_logic;
  signal match_1_3        : std_logic;
  signal match_1_2        : std_logic;
  signal match_1_1        : std_logic;
  signal match_1_0        : std_logic;
  signal match_0_3        : std_logic;
  signal match_0_2        : std_logic;
  signal match_0_1        : std_logic;
  signal match_0_0        : std_logic;

  constant four           : std_logic_vector(3 downto 0) := "1111";
  constant three          : std_logic_vector(3 downto 0) := "0111";
  constant two            : std_logic_vector(3 downto 0) := "0011";
  constant one            : std_logic_vector(3 downto 0) := "0001";
  constant zero           : std_logic_vector(3 downto 0) := "0000";


begin  -- behavioral

  -- always have current flags of matches between secret and guess
  match_3_3 <= ((secret(11) xnor leds(11)) and (secret(10) xnor leds(10))
               and (secret(9) xnor leds(9)));
  match_3_2 <= ((secret(11) xnor leds(8)) and (secret(10) xnor leds(7))
               and (secret(9) xnor leds(6)));
```

```
match_3_1 <= ((secret(11) xnor leds(5)) and (secret(10) xnor leds(4))
              and (secret(9) xnor leds(3)));
match_3_0 <= ((secret(11) xnor leds(2)) and (secret(10) xnor leds(1))
              and (secret(9) xnor leds(0)));
match_2_3 <= ((secret(8) xnor leds(11)) and (secret(7) xnor leds(10))
              and (secret(6) xnor leds(9)));
match_2_2 <= ((secret(8) xnor leds(8)) and (secret(7) xnor leds(7))
              and (secret(6) xnor leds(6)));
match_2_1 <= ((secret(8) xnor leds(5)) and (secret(7) xnor leds(4))
              and (secret(6) xnor leds(3)));
match_2_0 <= ((secret(8) xnor leds(2)) and (secret(7) xnor leds(1))
              and (secret(6) xnor leds(0)));
match_1_3 <= ((secret(5) xnor leds(11)) and (secret(4) xnor leds(10))
              and (secret(3) xnor leds(9)));
match_1_2 <= ((secret(5) xnor leds(8)) and (secret(4) xnor leds(7))
              and (secret(3) xnor leds(6)));
match_1_1 <= ((secret(5) xnor leds(5)) and (secret(4) xnor leds(4))
              and (secret(3) xnor leds(3)));
match_1_0 <= ((secret(5) xnor leds(2)) and (secret(4) xnor leds(1))
              and (secret(3) xnor leds(0)));
match_0_3 <= ((secret(2) xnor leds(11)) and (secret(1) xnor leds(10))
              and (secret(0) xnor leds(9)));
match_0_2 <= ((secret(2) xnor leds(8)) and (secret(1) xnor leds(7))
              and (secret(0) xnor leds(6)));
match_0_1 <= ((secret(2) xnor leds(5)) and (secret(1) xnor leds(4))
              and (secret(0) xnor leds(3)));
match_0_0 <= ((secret(2) xnor leds(2)) and (secret(1) xnor leds(1))
              and (secret(0) xnor leds(0)));

increment: process(n_rst, clk)
begin
  if n_rst = '0' then
    close_count <= (others => '0');
    right_count <= (others => '0');
  elsif rising_edge(clk) then
    if inc_close = '1' then
      close_count <= close_count + 1;
    end if;
    if inc_right = '1' then
      right_count <= right_count + 1;
    end if;
    if clear_counts = '1' then
      close_count <= (others => '0');
      right_count <= (others => '0');
    end if;
  end if;
end process increment;

state_clocked: process(clk, n_rst)
begin
  if (n_rst = '0') then
    -- reset everything here
    state <= idle;
    done <= '0';
    secret <= (others => '0');
    stored <= '0';
    inc_right <= '0';
    inc_close <= '0';
    close_leds <= (others => '0');
    right_leds <= (others => '0');
  elsif rising_edge(clk) then
    -- default signals to be innocuous
    done <= '0';
    inc_right <= '0';
    inc_close <= '0';
    clear_counts <= '0';
    case state is
      when idle =>
        -- default to remaining in idle state
        state <= idle;
        -- action requested
        if go = '1' then
      if stored = '1' then
      -- reset masks for new comparisons
        secret_mask <= (others => '1');
        led_mask <= (others => '1');
        clear_counts <= '1';
        state <= check_right3;
      else
        secret <= leds;
        state <= idle;
        stored <= '1';
        done <= '1';

      end if;
    end if;
  when check_right3 =>
    if match_3_3 = '1' then
      inc_right <= '1';
      secret_mask(3) <= '0';
      led_mask(3) <= '0';
    end if;
    state <= check_right2;
  when check_right2 =>
    if match_2_2 = '1' then
      inc_right <= '1';
      secret_mask(2) <= '0';
      led_mask(2) <= '0';
    end if;
    state <= check_right1;
  when check_right1 =>
    if match_1_1 = '1' then
      inc_right <= '1';
      secret_mask(1) <= '0';
      led_mask(1) <= '0';
    end if;
    state <= check_right0;
  when check_right0 =>
    if match_0_0 = '1' then
      inc_right <= '1';
      secret_mask(0) <= '0';
      led_mask(0) <= '0';
    end if;
    state <= spacer1;
  when spacer1 =>
    state <= disp_right;
  when disp_right =>
    case right_count is
      when "100" =>
         right_leds <= four;
      when "011" =>
         right_leds <= three;
      when "010" =>
         right_leds <= two;
      when "001" =>
         right_leds <= one;
      when others =>
         right_leds <= zero;
    end case;
    state <= check_close3;
  when check_close3 =>
    -- check secret color 3 against...
    if secret_mask(3) = '1' then
      -- this secret color has not been used
      if (led_mask(2) = '1' and match_3_2 = '1') then
        -- guess color 2 matches secret color 3
        inc_close <= '1';
        secret_mask(3) <= '0';
        led_mask(2) <= '0';
      elsif (led_mask(1) = '1' and match_3_1 = '1') then
        -- guess color 1
        inc_close <= '1';
        secret_mask(3) <= '0';
        led_mask(1) <= '0';
      elsif (led_mask(0) = '1' and match_3_0 = '1') then
        -- guess color 0
        inc_close <= '1';
        secret_mask(3) <= '0';
        led_mask(0) <= '0';
      end if;
    else
      -- do nothing since no off-matches were found
    end if;
    state <= check_close2;
  when check_close2 =>
    -- check secret color 2 against...
    if secret_mask(2) = '1' then
      -- this secret color has not been used
      if (led_mask(3) = '1' and match_2_3 = '1') then
        -- guess color 3
        inc_close <= '1';
        secret_mask(2) <= '0';
        led_mask(3) <= '0';
      elsif (led_mask(1) = '1' and match_2_1 = '1') then
        -- guess color 1
        inc_close <= '1';
        secret_mask(2) <= '0';
        led_mask(1) <= '0';
      elsif (led_mask(0) = '1' and match_2_0 = '1') then
        -- guess color 0
        inc_close <= '1';
        secret_mask(2) <= '0';
        led_mask(0) <= '0';
      end if;
    else
      -- do nothing
    end if;
    state <= check_close1;
  when check_close1 =>
    -- check secret color 1 against...
    if secret_mask(1) = '1' then
      -- this secret color has not been used
      if (led_mask(3) = '1' and match_1_3 = '1') then
        -- guess color 3
        inc_close <= '1';
        secret_mask(1) <= '0';
        led_mask(3) <= '0';
      elsif (led_mask(2) = '1' and match_1_2 = '1') then
        -- guess color 2
        inc_close <= '1';
        secret_mask(1) <= '0';
        led_mask(2) <= '0';
      elsif (led_mask(0) = '1' and match_1_0 = '1') then
        -- guess color 0
        inc_close <= '1';
        secret_mask(1) <= '0';
        led_mask(0) <= '0';
      end if;
    else
      -- do nothing
    end if;
    state <= check_close0;
  when check_close0 =>
    -- check secret color 0 against...
    if secret_mask(0) = '1' then
      -- this secret color has not been used
      if (led_mask(3) = '1' and match_0_3 = '1') then
        -- guess color 3
        inc_close <= '1';
        secret_mask(0) <= '0';
        led_mask(3) <= '0';
      elsif (led_mask(2) = '1' and match_0_2 = '1') then
        -- guess color 2
        inc_close <= '1';
        secret_mask(0) <= '0';
        led_mask(2) <= '0';
```

```
            elsif (led_mask(1) = '1' and match_0_1 = '1') then
              -- guess color 1
              inc_close <= '1';
              secret_mask(0) <= '0';
              led_mask(1) <= '0';
            end if;
          else
            -- do nothing
          end if;
          state <= spacer2;
        when spacer2 =>
          state <= disp_close;
        when disp_close =>
          case close_count is
            when "100" =>
               close_leds <= four;
            when "011" =>
               close_leds <= three;
            when "010" =>
               close_leds <= two;
            when "001" =>
               close_leds <= one;
            when others =>
               close_leds <= zero;
          end case;
          -- signal that the check is finished
          done <= '1';
          state <= idle;
        when others =>
          state <= idle;
      end case;
      if go <= '0' then
        done <= '0';
        state <= idle;
        inc_close <= '0';
        inc_right <= '0';
        clear_counts <= '1';
        end if;
    end if;
  end process state_clocked;

end architecture behavioral;
```

A.4 Main FSM

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity main_fsm is
  port ( clk              : in std_logic;
         n_rst              : in std_logic;

         review_prev        : in std_logic;
         review_next        : in std_logic;
         review             : in std_logic;

         check_done         : in std_logic;
         store_done         : in std_logic;
         enter_done         : in std_logic;

         correct_guess      : in std_logic;

         check_go           : out std_logic;
         store_go           : out std_logic;
         enter_go           : out std_logic;

         enter_drive        : out std_logic;
         store_drive        : out std_logic;

         store_mode         : out std_logic;

         reset_colors       : out std_logic;

         guesser_failed     : out std_logic;
         guesser_won        : out std_logic;

         round             : out std_logic_vector(3 downto 0)
    );

end main_fsm;

architecture behavioral of main_fsm is
  type StateType is (enter_secret, store_secret,
             enter_guess, check_guess, store_guess,
             review_guess, review_check, review_wait,
             won, failed);

  signal state             : StateType;
  signal max_guess         : std_logic_vector(3 downto 0);
  signal next_guess        : std_logic_vector(3 downto 0);
  signal current_guess     : std_logic_vector(3 downto 0);
  signal first_guess_flag  : std_logic;
  signal s_review_prev     : std_logic;
  signal s_review_next     : std_logic;


begin  -- behavioral

  -- sync input signals to the clock
  sync_signals: process(clk)
  begin
    if rising_edge(clk) then
      s_review_prev <= review_prev;
      s_review_next <= review_next;
    end if;
  end process sync_signals;

  max_guess <= next_guess + "1111";
```

```
state_clocked: process(clk, n_rst)
begin
  if (n_rst = '0') then
    -- reset everything here
    state <= enter_secret;
    enter_go <= '0';
    store_go <= '0';
    check_go <= '0';
    enter_drive <='0';
    store_drive <= '0';
    guesser_failed <= '0';
    guesser_won <= '0';
    store_mode <= '0';
    reset_colors <= '1';
    first_guess_flag <= '0';
    round <= (others => '0');
    max_guess <= (others => '0');
    current_guess <= (others => '0');
  elsif rising_edge(clk) then
    -- default signals to be innocuous
    enter_go <= '0';
    store_go <= '0';
    check_go <= '0';
    enter_drive <= '0';
    store_drive <= '0';
    guesser_failed <= '0';
    guesser_won <= '0';
    store_mode <= '0';
    reset_colors <= '0';
    round <= current_guess;
    case state is
      when enter_secret =>
        state <= enter_secret;
        enter_drive <= '1';
        enter_go <= '1';
        if (enter_done = '1') then
          state <= store_secret;
          enter_go <= '0';
        end if;
      when store_secret =>
        check_go <= '1';
        enter_drive <= '1';
        state <= store_secret;
        if (check_done = '1') then
          check_go <= '0';
          state <= enter_guess;
          reset_colors <= '1';
        end if;
      when enter_guess =>
        enter_go <= '1';
        enter_drive <= '1';
        state <= enter_guess;
        if next_guess = "1010" then
          state <= failed;
        elsif (review = '1' and first_guess_flag = '1') then
          state <= review_guess;
          enter_go <= '0';
        elsif (enter_done = '1') then
          enter_go <= '0';
          state <= check_guess;
        end if;
      when check_guess =>
        check_go <= '1';
        enter_drive <= '1';
        state <= check_guess;
        if (check_done = '1') then
          check_go <= '0';
          if correct_guess = '1' then
            state <= won;
          else
            state <= store_guess;
          end if;
          end if;
        when store_guess =>
          store_go <= '1';
          enter_drive <= '1';
          store_mode <= '1';
          round <= next_guess;
          state <= store_guess;
          if (store_done = '1') then
            next_guess <= next_guess + "0001";
            first_guess_flag <= '1';
            current_guess <= next_guess;
            store_go <= '0';
            state <= enter_guess;
            reset_colors <= '1';
          end if;
        when review_guess =>
          round <= current_guess;
          store_go <= '1';
          store_drive <= '1';
          store_mode <= '0';
          state <= review_guess;
          if (store_done = '1') then
            state <= review_check;
            store_go <= '0';
          end if;
        when review_check =>
          check_go <= '1';
          store_drive <= '1';
          state <= review_check;
          if (check_done = '1') then
            state <= review_wait;
            check_go <= '0';
          end if;
        when review_wait =>
          state <= review_wait;
          store_drive <= '1';
          if review = '0' then
            state <= enter_guess;
          elsif (s_review_next = '0' and review_next = '1') then
            if current_guess = max_guess then
              current_guess <= current_guess;
            else
              current_guess <= current_guess + "0001";
            end if;
            state <= review_guess;
          elsif (s_review_prev = '0' and review_prev = '1') then
            if current_guess = "0000" then
              current_guess <= current_guess;
            else
              current_guess <= current_guess + "1111";
            end if;
            state <= review_guess;
          end if;
        when won =>
          state <= won;
          enter_drive <= '1';
          guesser_won <= '1';
        when failed =>
          state <= failed;
          enter_drive <= '1';
          guesser_failed <= '1';
        when others =>
          state <= failed;
      end case;
    end if;
  end process state_clocked;

end architecture behavioral;
```

Appendix B: Starter Code Provided to Students

B.1 FSM File Template

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity fsm is
  port ( clk                    : in std_logic;
         rst_n                  : in std_logic;
         -- initiate process
         go                     : in std_logic;

         -- indication that process is done or response is valid
         done                   : out std_logic;
      );

end fsm;

architecture behavioral of fsm is
   type StateType is (idle, state1);

   signal state                 : StateType;

begin  -- behavioral

   state_clocked: process(clk, n_rst)
   begin
      if (rst_n = '0') then
         -- reset everything here
      elsif rising_edge(clk) then
         -- default signals to be innocuous
         case state is
            when idle =>
               -- default to remaining in idle state
               state <= idle;
               -- action requested
               if go = '1' then
                  -- start process
               end if;
            when state1 =>
               -- first state in process
            when others =>
               state <= idle;
         end case;
      end if;
   end process state_clocked;

end architecture behavioral;
```

B.2 Control File Template (CPLD)

```
-- NuBus
Attribute PIN_NUMBERS of  is "24"; -- A0
Attribute PIN_NUMBERS of  is "25"; -- A1
Attribute PIN_NUMBERS of  is "26"; -- A2
Attribute PIN_NUMBERS of  is "27"; -- A3
Attribute PIN_NUMBERS of  is "28"; -- A4
Attribute PIN_NUMBERS of  is "29"; -- A5
Attribute PIN_NUMBERS of  is "30"; -- A6
Attribute PIN_NUMBERS of  is "31"; -- A7
Attribute PIN_NUMBERS of  is "33"; -- A8
Attribute PIN_NUMBERS of  is "34"; -- A9
Attribute PIN_NUMBERS of  is "36"; -- A10
Attribute PIN_NUMBERS of  is "37"; -- A11
Attribute PIN_NUMBERS of  is "38"; -- A12
Attribute PIN_NUMBERS of  is "39"; -- A13
Attribute PIN_NUMBERS of  is "40"; -- A14
Attribute PIN_NUMBERS of  is "45"; -- A15
Attribute PIN_NUMBERS of  is "46"; -- A16
Attribute PIN_NUMBERS of  is "47"; -- A17
Attribute PIN_NUMBERS of  is "48"; -- A18
Attribute PIN_NUMBERS of  is "49"; -- A19
Attribute PIN_NUMBERS of  is "50"; -- A20
Attribute PIN_NUMBERS of  is "52"; -- A21
Attribute PIN_NUMBERS of  is "54"; -- A22
Attribute PIN_NUMBERS of  is "55"; -- A23
Attribute PIN_NUMBERS of  is "56"; -- A24
Attribute PIN_NUMBERS of  is "57"; -- A25
Attribute PIN_NUMBERS of  is "58"; -- A26
Attribute PIN_NUMBERS of  is "59"; -- A27
Attribute PIN_NUMBERS of  is "60"; -- A28
Attribute PIN_NUMBERS of  is "61"; -- A29
Attribute PIN_NUMBERS of  is "66"; -- A30

-- Connector
Attribute PIN_NUMBERS of  is  "3"; -- L1-00 L2-08
Attribute PIN_NUMBERS of  is  "4"; -- L1-01 L2-09
Attribute PIN_NUMBERS of  is  "5"; -- L1-02 L2-10
Attribute PIN_NUMBERS of  is  "6"; -- L1-03 L2-11
Attribute PIN_NUMBERS of  is  "7"; -- L1-04 L2-12
Attribute PIN_NUMBERS of  is  "8"; -- L1-05 L2-13
Attribute PIN_NUMBERS of  is  "9"; -- L1-06 L2-14
Attribute PIN_NUMBERS of  is "10"; -- L1-07 L2-15
Attribute PIN_NUMBERS of  is "15"; -- L1-08   gnd
Attribute PIN_NUMBERS of  is "16"; -- L1-09 L3-00
Attribute PIN_NUMBERS of  is "17"; -- L1-10 L3-01
Attribute PIN_NUMBERS of  is "18"; -- L1-11 L3-02
Attribute PIN_NUMBERS of  is "67"; -- L1-12 L3-03
Attribute PIN_NUMBERS of  is "68"; -- L1-13 L3-04
Attribute PIN_NUMBERS of  is "69"; -- L1-14 L3-05
Attribute PIN_NUMBERS of  is "70"; -- L1-15 L3-06
Attribute PIN_NUMBERS of  is "71"; --   gnd L3-07
Attribute PIN_NUMBERS of  is "75"; -- L2-00 L3-08
Attribute PIN_NUMBERS of  is "76"; -- L2-01 L3-09
Attribute PIN_NUMBERS of  is "77"; -- L2-02 L3-10
Attribute PIN_NUMBERS of  is "78"; -- L2-03 L3-11
Attribute PIN_NUMBERS of  is "79"; -- L2-04 L3-12
Attribute PIN_NUMBERS of  is "80"; -- L2-05 L3-13
Attribute PIN_NUMBERS of  is "81"; -- L2-06 L3-14
Attribute PIN_NUMBERS of  is "82"; -- L2-07 L3-15
```

Bibliography

[1] Marc D. Tanner. Helium Breath: An Updated 6.111 Curriculum. Master's thesis, Massachusetts Institute of Technology, May 1998.

[2] Kevin Skahill. *VHDL for Programmable Logic.* Addison-Wesley, Menlo Park, 1996

[3] Donald Troxel and James Kirtley. 6.111 course notes. Massachusetts Institute of Technology

[4] Donald E. Troxel. *6.111 Laboratory 2: Finite State Machines.* Massachusetts Institute of Technology

[5] Toby Nelson. Investigations into the Master Mind Board Game. February 5, 1999
URL: http://www.tnelson.demon.co.uk/mastermind/index.html