MASTERMIND

Project Design and Implementation

NOOK HARQUAIL & ANDREW HOH
ENGINEERING 31
BRIAN POGUE
SPRING 2012

Table of Contents

MASTERMIND

Project Design and Implementation

Abstract

The goal of this project was to construct a two-player Mastermind game using an FPGA board programmed with VHDL. Our program's output was to an LED array, and the inputs were six buttons — two external ones and four on the Nexsys II board. We were successful in fully implementing the game, and added scrolling text to announce each player's turn.

Introduction

The Mastermind game is traditionally a board game, and was invented by Mordecai Meitowitz in 1970. The game consists of a board into which colored pegs are inserted. The first player chooses a code composed of colored pegs, which are hidden from view. The second player attempts to win the game by guessing the first player's combination of colors. After each guess, the first player provides feedback by placing black and white pegs into holes beside the second player's guess; black pegs for each peg which has a color in the solution but is in the wrong place, and white pegs for each peg with the correct place and color. Although fairly simple, the game can be expanded to be extremely challenging by adding more digits to the guess or more color choices. The traditional implementation of the game has six choices of colors for each of four positions in a guess, with ten guesses allowed before losing (Board Game Geek). Algorithms exist that solve the game with fewer than five guesses on average (Rosu).

Design Solution

Specifications

We created a digital implementation of the mastermind game. The game begins when the first player sets an initial code using the four buttons to change colors. The second player tries to guess the solution. After each guess, colors to the right indicate how close the guess is to the solution: green lights for each digit with the right color in the right position and violet lights for each right color in the wrong position. The second player is given eight chances to guess the solution. We also have a reset button, which resets the game at any point.

Operating Instructions

0. the four buttons on the FPGA board control the left four columns of the LED, with the right four columns displaying feedback on the guess (violet for each correct color in the wrong place, green for each digit of the code with the correct place and color).

1. click submit button to start the game

2. the first player selects a four-color code to be the solution (there are six possible colors)

3. the first player presses submit and the solution is hidden

4. the second player tries to guess the code

5. the second player presses submit

6. the second player repeats steps 4 and 5, reacting to feedback

7. the second player wins if he/she reaches the solution in eight or fewer guesses —

otherwise the first player wins

8. the reset button to restarts the game at any point in this process

Theory of Operation

We will necessarily omit some of the details of our project's operation. The full source code — including comments — can be found in Appendix H. The state machine and data path diagrams in Appendices C and D may also prove helpful.

The code is broken into 6 modules: *Mastermind*, *Debouncer*, *CheckAnswers*, *ColorCycler*, *ClockDivider* and *ThreeBitCounter*. We'll discuss *Mastermind* in the last paragraph because it is by far the most complicated and relies on understanding the rest.

First, a note on our data: we use an internal signal — *curr_output* — within *Mastermind* to represent the output colors within a row. It is 24 bits, because there are 8 lights in each row * 3 colors per light. The digits *curr_output_* translate into the output ports *red1* through *green8* — the order is red, blue, green repeating. The input ports of the system are *bt1* through *bt4*, *reset, submit, and clk50*. The last output port is *row_num,* which is the index of the currently enabled row. *Row_num_* is controls which cathode is tied to ground on the LED.

Debouncer is very similar to the debouncer we constructed in the rotary decoder lab. It uses two storage signals to de-bounce the input signal by waiting a number of cycles defined by the constant *MAXCOUNT* before changing the value of the debounced output. It is clocked on the divided clock, *SlowCE*.

ThreeBitCounter is (as its name suggests) a three-bit counter. It is also very simply constructed, and also relies on *SlowCE.* It increments an internal three-bit signal each slowed clock cycle. Its output is used to select which row should be enabled, and then which column in the row should be turned on.

ColorCycler is a module for changing the color of lights in response to button presses. It takes in *initial*, which is a twelve-bit representation of the initial state of the row before button presses (four lights * 3 colors each). It outputs *final*, which is the incremented value of the row. It has an enabler and is clocked with the 50 mHz clock. The heart of the module in *incr,* which represents the state of the buttons — one digit for each column. For every button that is pressed, it increments the three bits which represent that column in a temporary variable. Those bits are concatenated to form the *final* output. Once it reaches the sixth color, it loops around to the first. *ColorCycler* only operates when its enabler (*cc_en_*) is turned on.

ClockDivider is fairly self explanatory. It takes a 50mHz clock and a generic *nclkdiv* value, which determines the speed of the output signal, *slowCE*. *ClockDivider* determines *MAXCLKDIV* by raising 2 to the power of (*nclkdiv* -1). It then counts up to that value, enabling slowCE only when the value of the counter variable is equal to MAXCLKDIV. This code is taken from an early lab. We used a generic *nclkdiv* value, so that we could create multiple clock dividers to run components with different *slowCE* speeds.

CheckAnswers is the second-most complex module. It takes in solution and guess, which are each twelve bit vectors representing the correct answer and the guess to which it is to be compared. Its output is correctness, which is the twelve bits representing the feedback given to the player on his/her guess. It has an enabler and is clocked with slowCE. It is constructed as a state machine, whose states are an initial, waiting state (waiting), four states for comparing the guess to the solution (fst_cmp through fth_cmp), four states for determining the number of LEDs with the right color, but wrong position (dtm_fst through dtm_fth), and a state for setting the output (aptly named set_output). Correctness is composed of two elements: how many are digits have the right place and color, how many have only the right color in the wrong place, and how many have neither. The first 8 states (after waiting) determine how many of each type of guess the player made. The final lookup table outside the main process expands the two values into a twelve-bit vector which contains the appropriate green and purple lights. A key component to the CheckAnswers algorithm was setting the signals with guess and solution values to impossible vectors. Checking for right colors in the right positions was extremely straightforward. However, for checking the right colors in the wrong positions, the program had to effectively ensure that there were no repetitions. For example, if a player guesses all red and there is one red in the solution, then the program should not assume that one red is in the right spot, while the other three are right, but in the wrong positions.

Mastermind is the final module, which combines everything else to control the final output of the LEDs. It is constructed as a state machine, whose states fall roughly into three categories: initializing data *(waiting, set_init, init_dis_1, init_dis_2, init_clear, clear)*, performing actions for players *(player1, player2)*, and displaying messages (*dis_player1, dis_player2, shift_rows1, shiftrows2, win, lose*).

First, let's look at everything that happens outside the state machine. Clock dividers are set up and debouncers are connected to buttons. Three bit counters are connected to row_counter and sng_counter, which control the enabled row and the enabled column. *Sng_counter* simply selects the current light that should be enabled within *curr_row* by turning off the other lights. Row_counter selects which row is *curr_row* and which cathode is enabled. *Sng_counter* is clocked faster than *row_counter* so that individual columns cycle more rapidly than rows. Finally, *curr_outputL* is broken down into 24 signals, one for each of the LEDs within the current row.

In the state machine, we start out in the initial *¬waiting* state. This is were all variables are reset and the screen is cleared. Any time the reset button is pressed, we will return to *¬waiting*. In this state, the solution is set to an impossible number. The only way to leave this state is by pressing the submit button, which moves the machine to *init_dis_1*.

Here we set up the image for displaying "PLAYER 1" across the display. Of course, that does not fit in an 8x8 square, so we will have to scroll through it. To that end, we set up a counter which will count up to the constant *MAX_COUNT*, which is the time in slowed (*clearCE*) clock cycles for which the message should be displayed. The "PLAYER 1" image is stored within an array of length 8 with long standard logic vectors. Each vector is "scrolled" by taking the three bits for an LED and adding the bits to the end.

From *init_dis_*1*,* we move to *dis_player1*. This simply loads the picture we created in *init_dis_1* into the seven *rows.* If *MAXCOUNTER* has been reached, it moves to *set_initL*, until then it goes to *shift_rows_1*, which will return back here after it has moved the image one light to the left.

Shift_rows moves each of the lights is *dis_pic* one column to the left, and increments a counter variable. It returns to *dis_player1*, which will eventually move to *set_init.*

Here the first row and guess are set to red, and the button storage value incr is set to empty. The variable which turns on color cycling with button presses — cc_en — is turned on. From here, we move to player1.

Player1 checks to see if the solution is still at its default value, and sets the first row to the guess and result (red in the left 4 columns). Color cycling is temporarily disabled. If the player presses submit, we move to the first of the second player's states — *player2*. Otherwise we stay in this state, using the *ColorCycler* module to change the first player's *guess* (which is the *solution_*) based on button presses. *ColorCycler* eliminates rapid cycling by only changing on the rising edge of the button press.

After the first player has set the solution, we display the "PLAYER 2" message. *Dis_player2, init_dis_2,* and *shift_rows_2* work virtually identically to their analogues for the first player. One of the few differences is that turn is set to an arbitrary unreachable turn in *init_dis_2*, so that the game's tiles are not displayed during the message. After these are executed, the state is moved to *clear. (Turn_* will be reset to 0).

Clear hides the first player's solution by resetting the *guess* to four reds, and resetting the color increment values. The next state is, of course, *player2*.

Player2 has similarities to *player1*, but is different in its particulars. Button color cycling is identical, but the variable *turn_* becomes important. It is incremented every invocation of *player2,* and controls which row the second player is working in. During *player1* color cycling (*cc_en_*) is temporarily disabled while button values are calculated, and *rows*(*turn_*) is set to the guess with blank feedback. *CheckAnswers* is enabled by turning on *ca_en_*, the resulting feedback of the *guess* is received from *temp_result_*, which is stored in result. Cc*_en_* is reactivated, and the next state is *set_row*.

Set_row combines the guess with the result, to form the full 24 bits of output data, which it stores it *row(turn-1)* because the result is placed within the previous guess. This state's other purpose is to determine if the game is still in progress or we have reached a terminal state. If the second player has not guessed the solution and there are turns remaining, we return to *player2* to obtain a new guess and display feedback. If more than eight turns have been taken and the solution has not been found, the next state is *lose*. If neither of these are the case, the second player must have won and we move to *win*.

Win and *lose* are both terminal states, which display a static message of a smiling or frowning emoticon, respectively. They loop indefinitely — until reset is pressed.

At various points throughout the state machine, there are if statements to catch *reset* button presses, and return to the *waiting* state immediately.

Others states should not occur, but if they do, the game is restarted by going to *waiting*.

Construction and Debugging

The project was constructed over several weeks, with the majority of time being spent on tweaking the code to eliminate bugs.

The first day after receiving the parts, we worked on the code and tested how the LED matrix functioned. We hooked up resistors to the cathodes and began wiring all the colored lights to their respective ports. We initially found that the red light overpowered all the other colors — whenever any red LED was lit, the other lights on the display were either off or very dim. After consulting the data sheet, we discovered that the red LEDs required a lower voltage that the other colors. We therefore constructed a voltage divider to drop the 3.3V from the FPGA to the desired 2.0V. After this point, we could display any of the six colors plus white in any position on the array. We constructed a simple test program to cycle through the rows and light all the LEDs with white, which we used to diagnose mis-wired pins and later to discover which wire we had accidentally unattached.

We then added submit and reset buttons, and were ready to start testing code on the FPGA.

We avoided having to use a pulse width modulator or constant current driver by only using direct combinations of red, blue, and green. This was perfect because we needed exactly 8 unique colors, 6 colors for the color cycling, off, and white for errors.

We chose to use green and purple lights for correct and nearly correct digits — rather than the red, yellow, and green lights we originally planned — because a red light in the rightmost column of our LED does not work. This was a minor obstacle, easily circumvented.

We found that rows with many lights in them were dimmer than those with fewer, which made the hybrid colors much dimmer than just primary colors. We fixed this by adding a single counter (*sng_counter*) to cycle through each LED individually.

A major problem we had was that compile times grew to be quite long, especially after we added the scrolling messages before each player's turn. By the end, the code took upwards of six minutes to recompile after every change. This hampered testing, as we had to be increasingly careful about wasting compilations, often combining running multiple test changes at once to save time.

We also had a number of problems related to timing. We also were initially reaching an *others* case we had not intended to be reachable during *player2*, which caused the solution to display white error lights rather than the correct output.

We also had problems with *player1* executing when we did not expect it to. This caused the first row to continue to change with the current row throughout the game. Another, similar problem we encountered was that the last played row would display through win and lose messages. We fixed these problems by adding if statements and extra states to prevent them from occurring.

We also had a myriad of other small problems, such as the solution being displayed one row later than the guess with which it was supposed to be associated. This was a simple off by one error, which was easy to solve.

When we implemented the scrolling messages, we at first had difficulty getting them to scroll and then had problems returning to gameplay after the message. To solve these problems, we had to add additional states to properly clear the display after messages and ensure that cycling occurs. We adjusted the time of *MAX_COUNTER*, so that messages would only displayed for as long as they take to move across the display once.

Explanation of Wiring

This explanation will make most sense when used in conjunction with the schematic in Appendix E, the package map in Appendix F, and the overhead picture in Appendix A.

Our circuit involves a large number of wires to control the LED matrix. We used a common-cathode display, so eight of the wires are for the cathodes which control which row is currently enabled. The wires going to the cathodes are yellow. We cycle through each of the rows, so that only one is enabled at a time. The cathode wires are each connected to a 180 Ω resistor to reduce the current to the 20 mA required by the LEDs. Following convention, the power and ground wires are red and black, respectively.

The majority of the remaining wires control individual led colors. Because only one row is enabled by the cathodes at a time the remaining pins are used to control which red, green, and blue lights are on in each row. Each of the positions in the eight rows contains three lights, so there are 24 pins to control color. We color coded the wires: orange for those controlling red LEDs, blue for the blue LEDs, and green for the green LEDs.

Because the red LEDs demand a different voltage than the other lights (Appendix L), we used a voltage divider to reduce the 3.3 V from the board to the 2.0 V required. The eight voltage dividers each consist of a 100 Ω and a 10 Ω resistor. The 10 Ω resistor in each pair is connected to an IO port, the 100 Ω resistor to ground.

The power and ground wires connected to JA1 are to provide power to the reset and submit buttons, which output 1 when they are pressed.

Evaluation of Design

Our original project proposal was as follows (the bolded portions are the ones we ultimately implemented):

"Our base project will be an implementation of the game Mastermind, using four bits of input (leds on or off), and using eight leds (or four multi-colored leds) to give feedback on guesses. The game rules are as follows, slightly modified from http://www.pressmantoy.com/instructions/instruct_mastermind.html. It is described with multiple colors so that this description will be relevant to the expansions described in the second half of the proposal, but you can easily imagine a monochromatic implementation.

1. The code maker selects a code.
2. The code breaker attempts to duplicate the exact colors and positions of the secret

code each turn.
3. The system generates feedback on each guess as follows:
4. A green color light for each code item that's the right color and in the right position.
5. A yellow color light for each code item that's the right color, but not in the right position. Red lights indicate a color is not used in the code.
6. The system does not put the lights in any particular order. It's part of the challenge for the code breaker to figure out which lights correspond to which digit. The codebreaker should remember that one light corresponds to one digit and a green light takes precedence over a yellow one.

This is fairly simple, but there are many opportunities for expansion, arranged roughly in the order in which we would implement them:

•We could offer the ability to have the game randomly choose the code.

•Rather than a binary code, we could have leds with multiple colors for each position. The standard arrangement is six colors (ROYGBV) in four rows. In this scenario, we would have a button for each of the columns to cycle through the colors, and a button to accept the input.

•We could keep track of the player's last row of input and display it for the next guess by default, because they will often want to keep many entries the same.

•Our original design includes only one row of leds for input, we could used an **led matrix** or vga display to show multiple rows of guesses (and the results of each).

•We could add more than two players — one player sets the initial code, others take turns guessing.

•We could keep track of the score over multiple games (a sane scoring method might be to award the person who chose the code points equal to the number of guesses taken to solve it).

•We could create an ai opponent, using a guessing algorithm with various levels of difficulty — culminating in an opponent who can guess the code quite quickly (I suspect it is possible to always solve the game within 8 guesses).

•We could read and write high scores to persistent memory.

•If we used an led matrix or vga display, we could display a short introductory animation with instructions when the game started, and **game over**/ whose turn is it/high score **messages**.

•Now that we have many choices for the user (how many players, ai or not, how difficult an ai setting, perhaps how many colors to use), we could display a textual user interface with options to go back and change options, and a button for returning to the main menu during a game (with confirmation)."

Although we did not implement some of the flashier features suggested in the project proposal, we constructed a base game which is very playable. The final system consists of six buttons for setting colors, resetting the game, and submitting guesses — and an LED

Matrix that displays guesses and feedback on which guesses were in the right place and/or had the right color (see Appendix A). The major addition we made to the original project was the scrolling message display before each player's turn. This was an afterthought to the original project, but it proved to be as complex as a small additional project, involving several additional states and many additional hours of coding and debugging. With the addition of six extra states, a couple of new bugs appeared within the code. Unfortunately, we were not able to add more features because of the 6 to 7 minute compiling time and the abnormally difficult test-benching.

Conclusions and Recommendations

Controlling the RGB matrix was slightly more difficult than we had expected, but we would recommend that future students opt for a parallel rather than serially controlled led matrix if feasible. We found that the wiring portion of the lab was quite enjoyable: even though it was time consuming, debugging was extremely easy. (We could directly test that we were wiring the display correctly by using the power and ground columns of the breadboard, and were able to quickly diagnose wiring problems using the multimeter.) After our struggles with Lab 5, we were happy to avoid the headaches associated with serial interfaces.

We separated out a few discrete components — such as the debouncer and answer checking algorithm — into separate files, but the bulk of the code is within the *Mastermind* state machine. We could easily have split portions of the state machine into other files (most states could conceivably have been separate modules). However, splitting it into more modules would have made the structure of the code less obvious, and would have necessitated the creation of many modules which do not have discrete, easily reusable functions.

We therefore kept the majority of functions for controlling what was displayed on the LED within the *Mastermind* state machine, for both our and the reader's convenience.

Acknowledgements

Thanks to the professors and TAs. Special thanks to Dave Picard for helping us figure out which parts we needed, and particularly for having an expansion for the breadboard on hand. Thanks to our comrades in the lab for keeping us sane.

Division of Labor

Andrew and Nook each actively partook in every step of the final project. Andrew found the 8X8 LED display on sparkfun.com, and Nook did the initial research and ordering of the LED matrix. For the VHDL aspect of the project, Andrew wrote the *Mastermind* file, while Nook created the *ClockDivider*, *Debouncer*, *ThreeBitCounter*, as well as the *CheckAnswers*. We worked together to wire the LED and pushbuttons on the breadboard. We both thoroughly tested the project using testbenches, oscilloscopes, multimeters, and the FPGA board. Nook wrote this report and Andrew structured the appendices.

References

"Boardgame Geek — Mastermind (1971) (1971)." *Mastermind (1971)*. Web. 23 May 2012. <http://boardgamegeek.com/boardgame/2392/mastermind>.

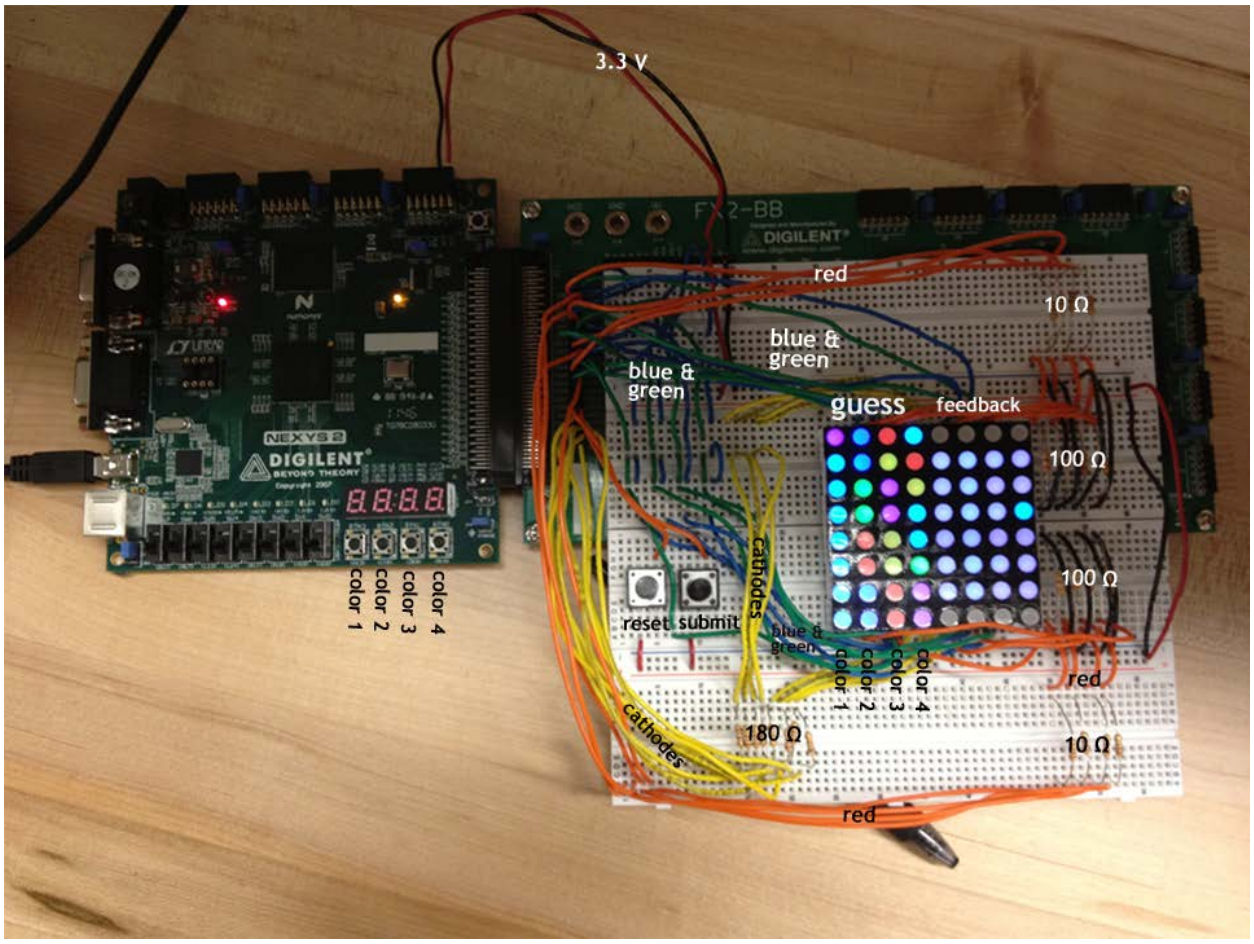
Mealy, Brian. "Low Carb VHDL Tutorial." Web. 19 May 2012. <http://www.eecs.ucf.edu/~mingjie/EEL4783_2012/>.

"Pressman Toy Instructions for Mastermind." *Pressman Toy Instructions for Mastermind*. Web. 20 May 2012. <http://www.pressmantoy.com/instructions/instruct_mastermind.html>.

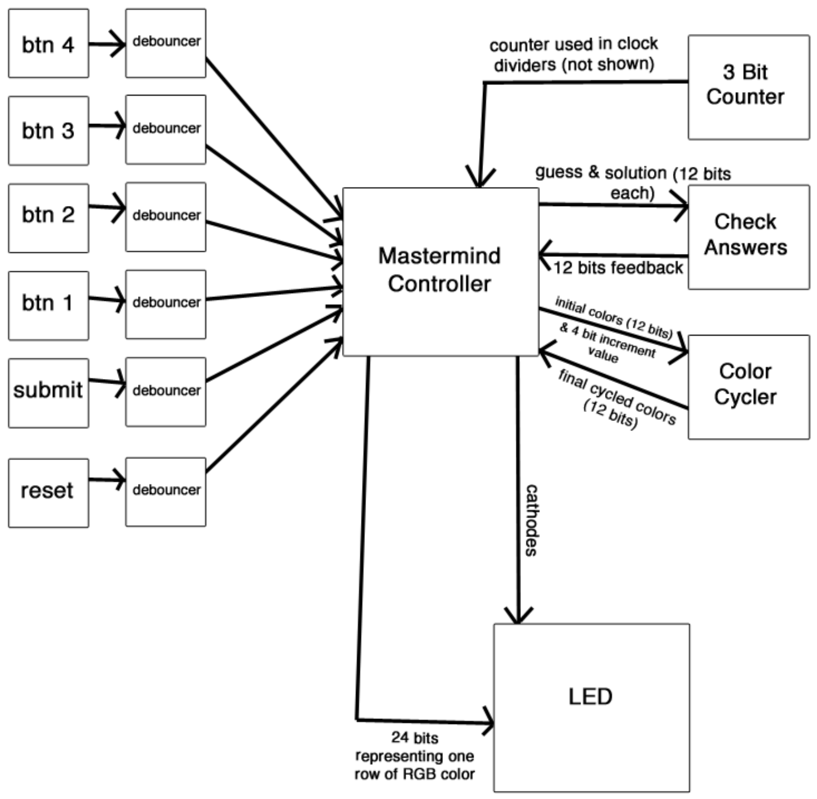
Rosu, Radu. "The Mastermind Solver Page." *The Mastermind Solver Page*. Web. 24 May 2012. <http://www.unc.edu/~radu/mm/MMS.html>.

Appendices

Appendix A: Overhead Picture

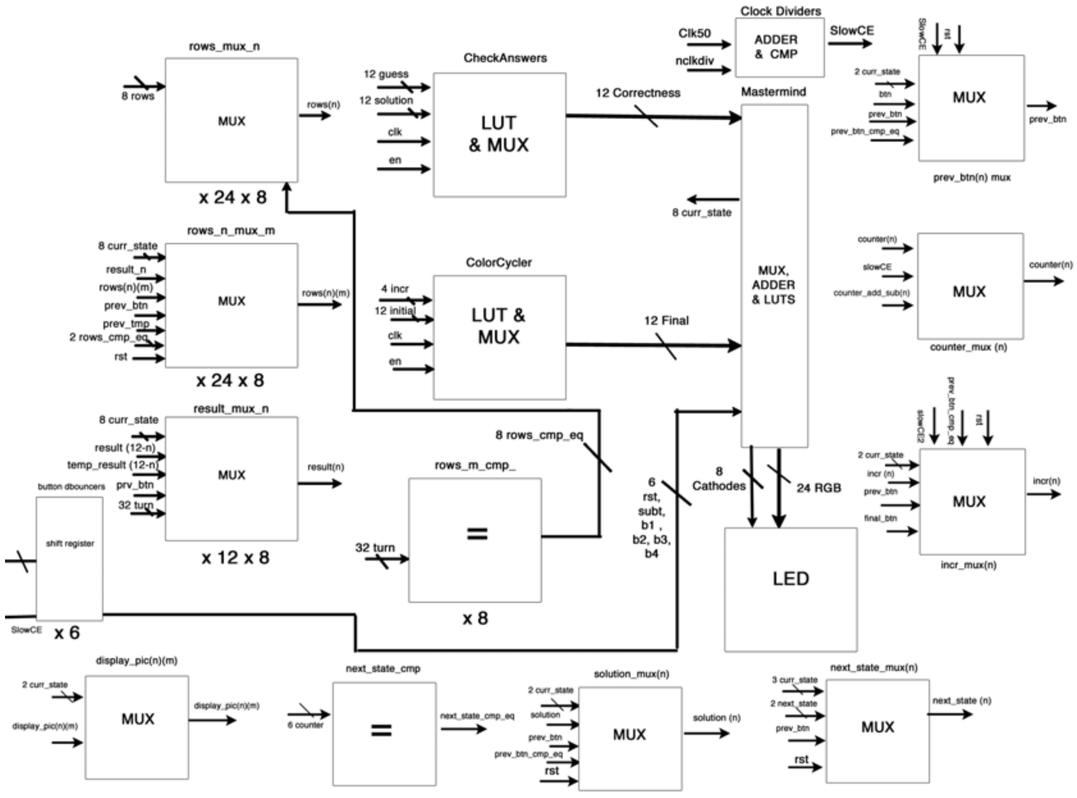


APPENDIX B: BLOCK DIAGRAM


btn 4
btn 3
btn 2
btn 1
submit
reset
debouncer
counter used in clock dividers (not shown)
3 Bit Counter
guess & solution (12 bits each)
Check Answers
12 bits feedback
Mastermind Controller
initial colors (12 bits) & 4 bit increment value
Color Cycler
final cycled colors (12 bits)
cathodes
LED
24 bits representing one row of RGB color

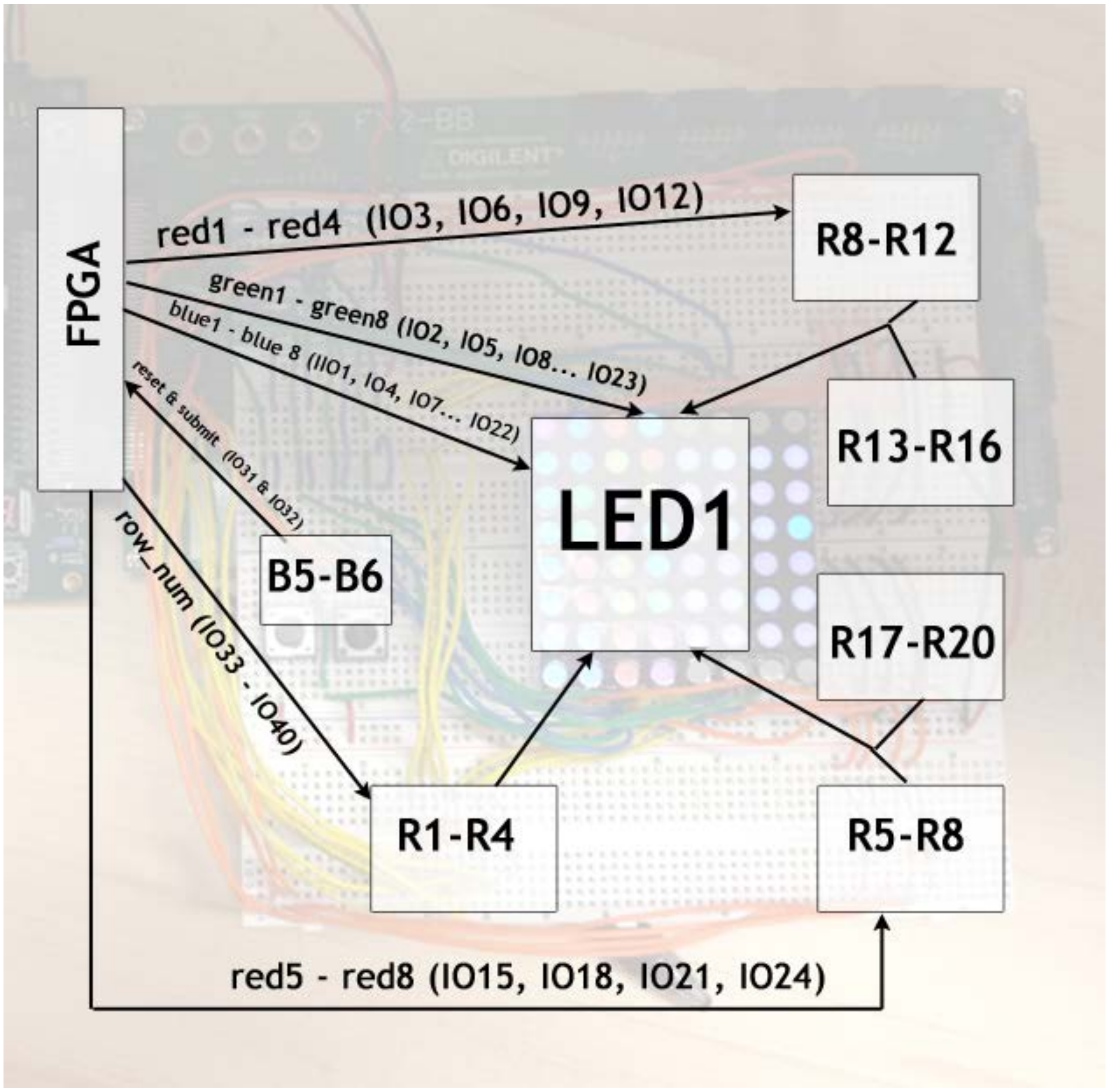
Appendix C: State Machines

Appendix D: Data Path



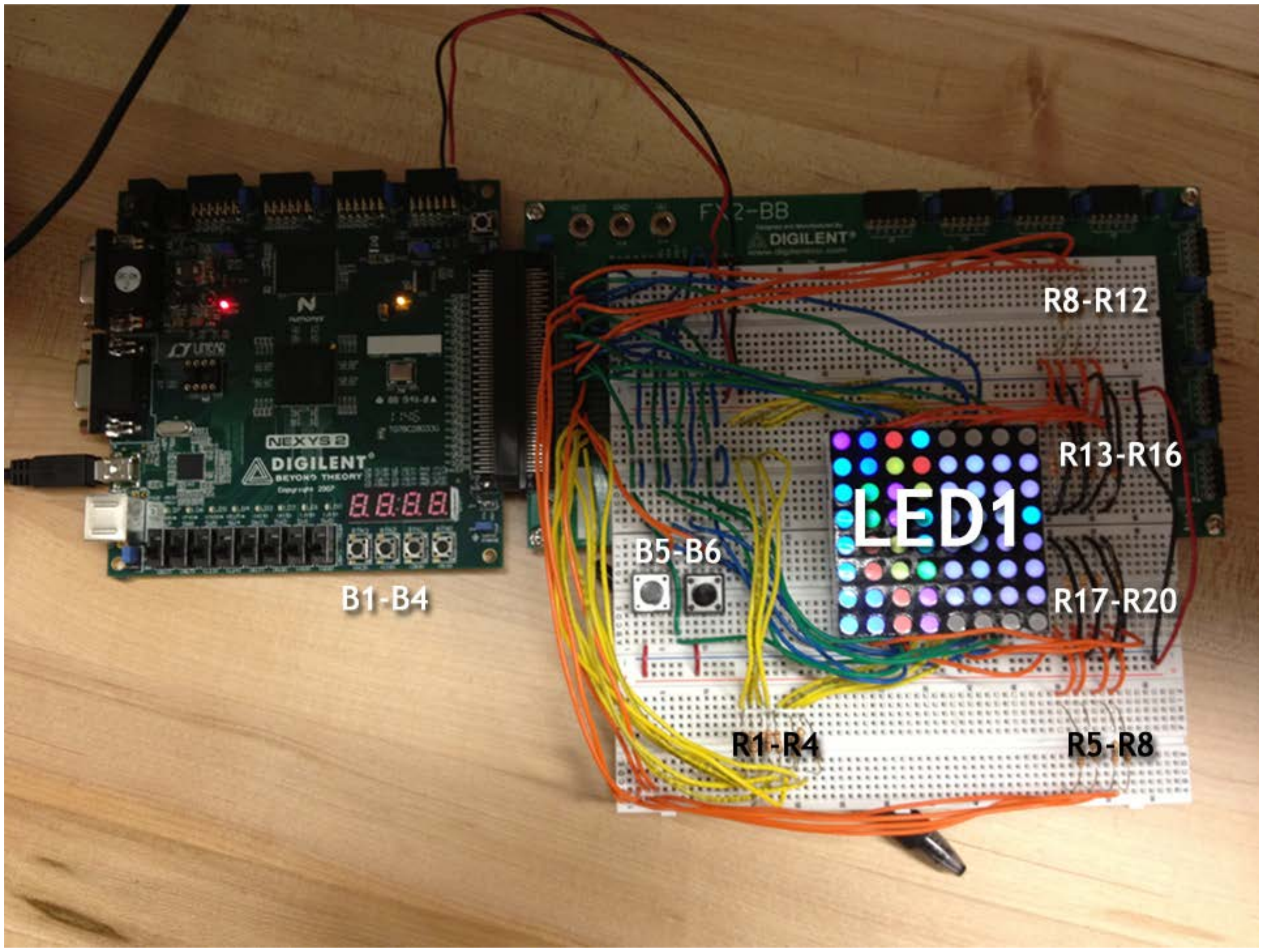
(RTL Schematics follow)

APPENDIX E: SCHEMATIC



FPGA
red1 - red4 (IO3, IO6, IO9, IO12)
R8-R12
green1 - green8 (IO2, IO5, IO8... IO23)
blue1 - blue 8 (IIO1, IO4, IO7... IO22)
reset & submit (IO31 & IO32)
R13-R16
LED1
B5-B6
row_num (IO33 - IO40)
R17-R20
R1-R4
R5-R8
red5 - red8 (IO15, IO18, IO21, IO24)

APPENDIX F: PACKAGE MAP



Appendix G: Parts List

Nexsys 2 FPGA Board
(http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS2)

8x8 RGB LED Matrix (http://www.sparkfun.com/products/683/)

2 x Momentary Pushbutton (http://www.sparkfun.com/products/9190)

8 x 10 Ω Resistor

8 x 390 Ω Resistor

8 x 180 Ω Resistor

Connecting Wires

Appendix H: VHDL Source

Appendix I: Resource Utilization

Appendix J: Timing Diagrams

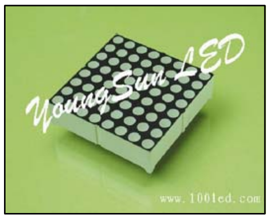
Appendix K: Analysis of Warnings

We have no warnings!

Appendix L: LED Matrix Data Sheet

深 圳 市 昱 申 科 技 有 限 公 司
CHINA YOUNG SUN LED TECHNOLOGY CO., LTD.
TEL: (86) 755-28079401 28079402 28079403 28079404 28079405
FAX: (86) 755-28079407 E-mail: info@100LED.com Web: www.100LED.com

Model No.: YSM-2388CRGBC
8mm Pitch RED/GREEN/BLUE Tripple Color Dot Matrix



Applications:

- Moving Message Display
- Full Color Display
- Banking Board
- Score Boards
- Digital Display

LED Chip Absolute Maximum Ratings: (Ta=25℃)

Pararmeter	Symbol	Red	Green	Blue	Unit
Forward curret	I_F	20	20	20	mA
Peak forward current(Duty Cycle=$\frac{1}{10}$, 10KHz)	I_{PF}	30	30	30	mA
Reverse voltage (V_R=5V)	I_R	10	10	10	μ A
Operating temp	T_{OPR}	-25 - 85	-25 - 85	-25 - 85	℃
Storage temp	T_{STG}	-30-85	-30-85	-30-85	℃
Spectral Line half-width	λ PH	20	30	30	nm

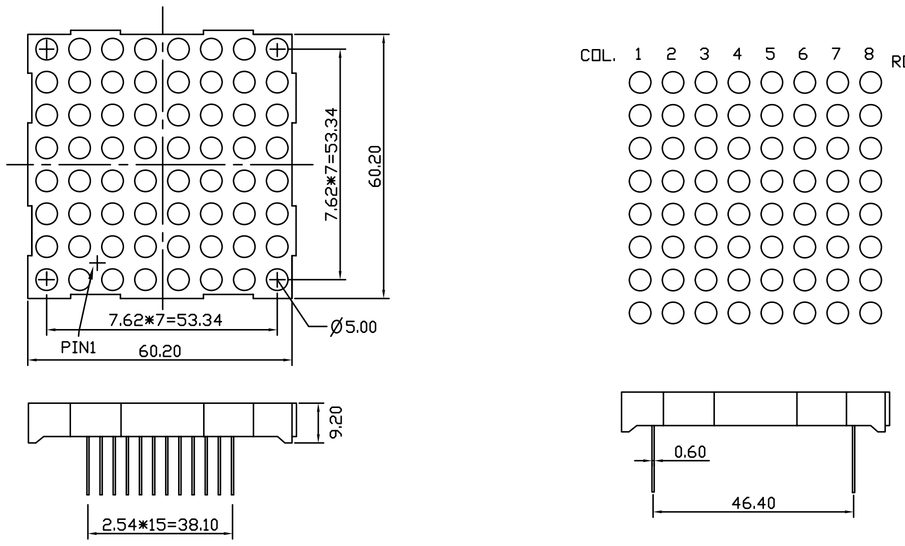
* Soldering Bath: not more than 5 seconds @260℃.The bottom ends of the plastic reflector should be at least 2mm above the solder surface
Soldering Iron: not more than 3 seconds @300℃ under 30W

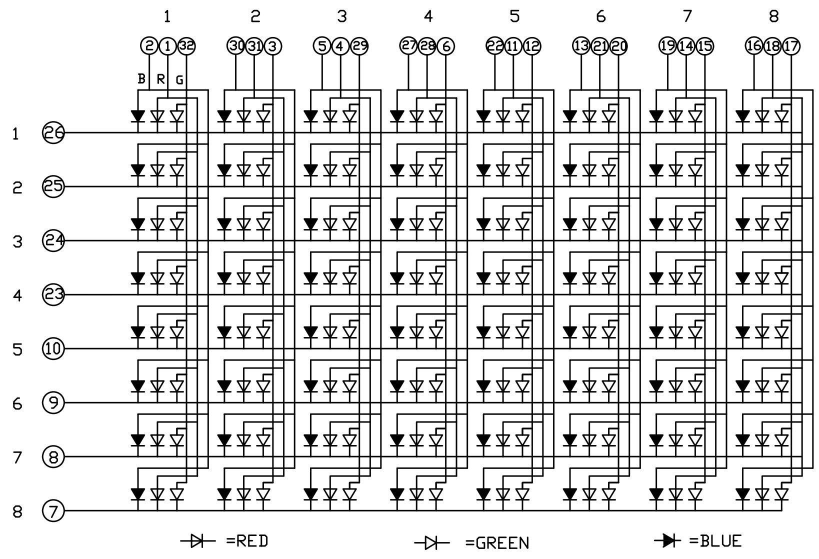
LED Chip Typical Electircal & Optical Characteristics: (Ta=25℃)

ITEMS	Color	Symbol	Condition	Min.	Typ.	Max.	Unit
Forward Voltage	Red	V_F	I_F=20mA	1.9	2.0	2.2	V
	Green			3.2	3.3	3.5	
	Blue			3.2	3.3	3.5	
Luminous Intensity	Red	I_V	I_F=20mA	120	150	160	mcd
	Green			250	300	330	
	Blue			180	200	250	
Wavelenength	Red	Δ λ	I_F=20mA	620	---	625	nm
	Green			515	---	517.5	
	Blue			465	---	467.5	
Light Degradation after 1000 hours	Red	-4.68% ~ -8.27%					
	Green	-11.37% ~ -15.30%					
	Blue	-12.47% ~ -16.81%					

Mechanical Dimensions:

All dimension are in mm, tolerance is ±0.2mm unless otherwise noted
An epoxy meniscus may extend about 1.5mm down the leads.

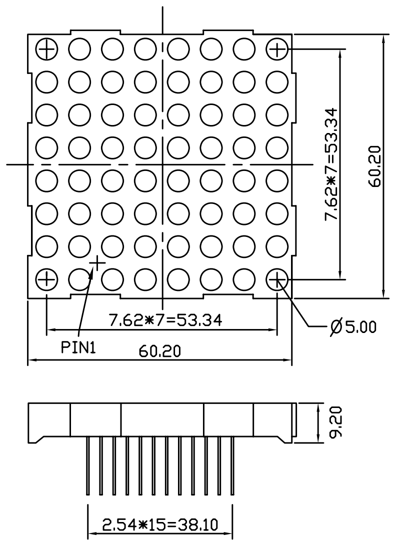


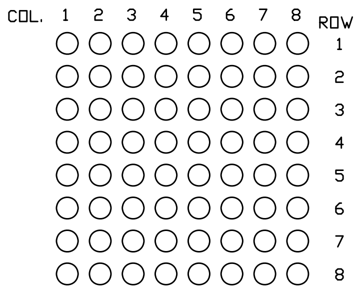


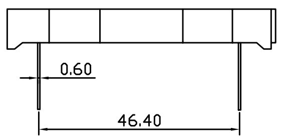
Unit: mm

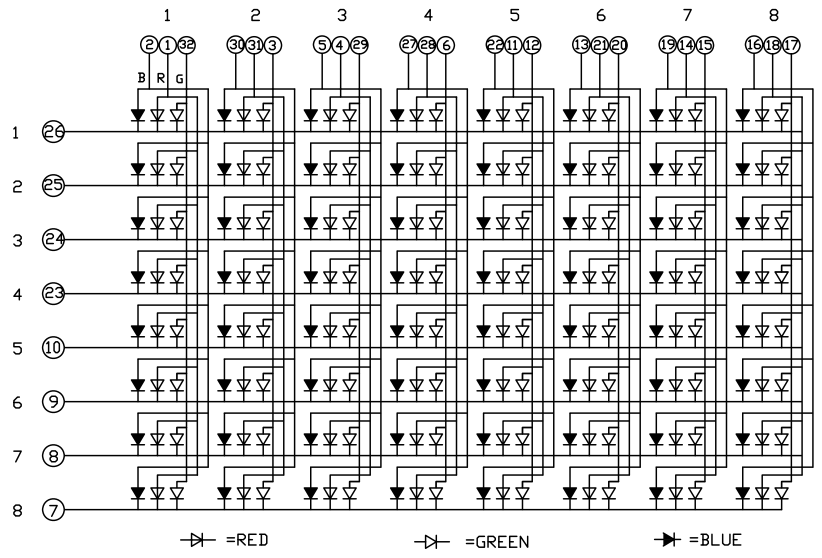
Mechanical Dimensions:

- All dimension are in mm, tolerance is ±0.2mm unless otherwise noted
- An epoxy meniscus may extend about 1.5mm down the leads.









Unit: mm