M152A Project: Mastermind

Ryan Stenberg Rodrigo Valle

March 22, 2017

Introduction

Our project is based entirely around the 1970s board game Mastermind. This game can be designed as a single player game with the help of a pseudo-random number generator. The idea is simple: the board will generate a random "code" as a sequence of colors, and it is the player's job to guess this code exactly. The player has 8 chances to guess correctly, and after each guess is made, some feedback is given to the player about their guess. Feedback is given in the following form: - For every color in the guess sequence that is both in the correct place and of the correct color, the seven segment display will show an "X". - For every color in the guess sequence that is one of the correct colors but not in the correct place, the seven segment display will show an "O". - Otherwise, the seven segment display will be blank. - These Xs and Os are not shown in any particular order so as to avoid revealing any more information to the player.

In Mastermind, the player has the ability to inspect their previous guesses and the feedback that was provided visually, in order to help them make their next guess. Due to a restriction on RGB LEDs, we implemented a history mode that allows players to flip through their previous guesses and see the feedback that was provided for those guesses. The details and difficulties of all of these features will be laid out in more detail.

Design

We designed our Mastermind implementation as modularly possible, so that the focus of each module remained small. Each of these modules is intended to stand on it's own as a part of the greater Mastermind game, and each was tested separately with its own testbench.

Modules

Our design was modularized into the following components:

- **prng.v**: This is a standalone module that (pseudo)randomly generates 4 3-bit numbers (reprsenting 4 colors), and stores theme. It takes an input that, when set high, will cause the prng to re-generate the numbers. Internally, it uses a Linear Feedback Shift Register (LSFR) as a means of generating pseudo random codes.
- **ssd_conveter.v**: Four instances of this module exist in our project. Each one takes a 2-bit code representing blank, an "X", or an "O", and converts it to an 8-bit binary number representation that will display the corresponding graphic when given to the ssd_driver. This module is entirely combinational.
- **ssd_driver.v**: This module takes 4 8-bit integers as input and a high-speed block (400 Hz). It will display each of these integers on the seven segment display by cycling across all 4 digits rapidly.
- **led_driver.v**: This module provides an interface for selecting colors for the LEDs and making a particular LEDs blink. It takes four 3-bit inputs from the guess module and four 3-bit inputs from the history module, and decides which to display depending on the currently selected mode. It also takes a blink selector integer, which specifies which LED to blink, and will also decide whether blinking should be enabled or disabled based on the mode (the selected LED only blinks in guess mode). Each of the four 3-bit outputs are sent to the FPGA's GPIO ports, where they power an RGB LED in a circuit to display the proper color. This is the main visual interface for the player.
- **history.v**: This module takes an enable switch, button up, button down, 4 3-bit guess wires, and a submit input. Essentially, history has access to the interface for when it needs to cycle up/down in history, and it has access to the player's current guess so that when the submit button is pressed, history can store the guess. By default in guess mode (indicated when the "enable" input is set low), the most recent guess along with the corresponding feedback created by the feedback module are sent to led_driver and sdd_driver modules so they can be shown to the player. Also, the current guess is output to thet turn modules so it can be displayed by illuminating an LED above the FPGA board's switches corresponding to the current turn. When in history mode (indicated when the "enable" input is set high), the history module begins accepting input from the up/down buttons and outputs the selected turn to the led_driver module. It also shows which turn is being viewed by sending the number of the current turn to the turn module. The final output of history is a single wire sent to feedback that indicates when the last turn has been taken. This is used for ending the game and resetting the game state to begin a new game.
- **guess.v**: This module takes the left/right/up/down buttons, and an enable wire. The up and down buttons allow users to cycle through the choices of

colors, and the left an right buttons allow players to change which LED they are currently choosing a color for. All of these changes are made to an internal register, whose contents are being sent to led_driver to be displayed (along with the currently selected LED, so that led_driver can blink it). The current guess is also sent to the history, so that it may record it when the user submits their guess. When the enable wire is set low, all guess-editing functionality is disabled so that the history module can display history.

- **feedback.v**: This module takes four 3-bit inputs from the history module and four 3-bit inputs from the prng module. Using combinatinoal logic, it compares them to determine how many corresponding Xs and Os to display on the SSD. This is sent to the ssd_converter modules in the form of four 2-bit wires. This module also handled the end of the game. When the last turn is signaled by the history module, a 5 second countdown begins. At the end of the countdown, the game auto resets.
- **turn.v**: This module takes a 3-bit input, and converts it into an 8-bit representation such that only one bit is on – its place corresponding to the number of the current turn. This is channeled directly to the LEDs above the switches on the FPGA, so that the user can have feedback on which turn they are taking (in guess mode), or which turn they are viewing (in history mode).
- **debouncer.v**: This module takes in a high speed clock and helps to sync the input buttons with the clock cycles on the board. It does so by requiring the button to be held down for several clock cycles and then only registers the press once that button is released. This module is used as an intermediary between each button and the rest of the modules.
- **clock_div.v**: This module takes in the built in clock from the FPGA board and divides it into several different speeds. Our game used a 1 Hz clock to orchestrate the countdown to reset at the end of a game, a 4 Hz clock to handle blinking, a 200 Hz clock to handle debouncing, and a 400 Hz clock to mutliplex the cathodes for the seven segment display.
- **mastermind.v**: Contains all the other modules and pairs them together correctly. Takes the up, down, left, right, and center buttons along with a single switch (the mode switch) as input. Outputs to the LEDs above the switches, GPIO pins, and the seven segment display.

Module Schematic

We've labeled the components on this schematic to give a better sense of how these modules fit together:

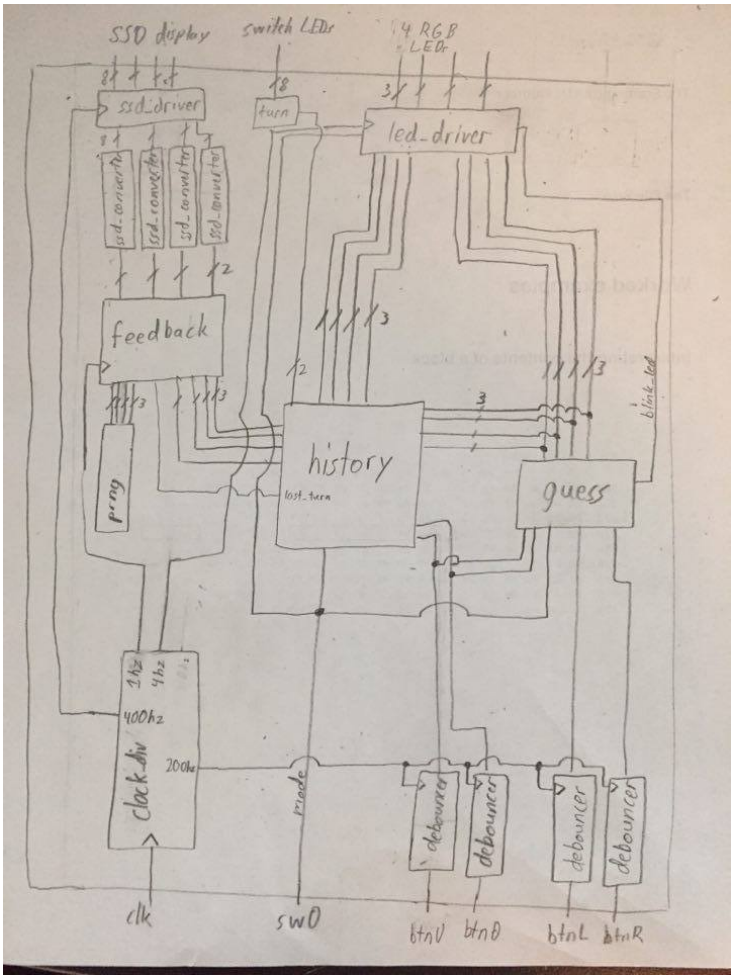

Figure 1: The inputs and outputs of our Mastermind game along with how it interacts with the modules that it's composed of.

Bugs

- **blinking bug**: There was a bug where the user would be presented with a sequence of all blank LEDs. The user could scroll through and select a color, but when right was pressed to select the next color all of the LEDs would begin blinking at a very fast rate. After one of the right or left buttons were pressed again, the blinking would stop, and the LEDs would be left with some seemingly random colors. This issue was fixed in our debouncer module; we had reused the debouncers from the last project, which tracked the button state so that buttons would act as "toggles". It turns out that when we were pressing left/right/up/down, our guess module thought we were holding the button down and quickly scrolling through several locations in the sequence at once attempting to change the pattern. We simply took the button state out of our debouncers and the problem was resolved.
- **general bug**: We were told that using initial blocks could cause strange behavior inside modules and were told to avoid it. We briefly changed our modules so that they all supported a reset input to initialize state within our module, but began having several issues with the change. We realized that initial blocks had not caused us any problems in the past and the Xilinx synthesizer seemed to support them well enough, so we decided to roll back the changes we had made and our synthesis issues were resolved. Our issue was that some modules were not designed with reset inputs in mind, so dropping them into our existing code caused issues with a) when the module would be initialized and b) how it would be initialized. A quick fix was to just set all reset inputs to a constant "high" signal, so the modules would be initialized on boot, but this proved cumbersome and we made the decision to roll back shortly after.
- **history bug**: Our history module would not display history further back than a few moves. We couldn't find the cause of this issue because we briefly modified our testbench to check that history was indeed being stored, and it certainly was, so we were baffled about why history would be losing data while in use. We suspect that maybe history could have been overwriting the same location in memory repeatedly rather than storing it into a new location each time. The guess module also holds guess state which is sent directly to led_driver to be displayed, and this could have also played a role in this bug.

Simulation Documentation

We tested all basic functionality of our game using testbenches, in order to make sure that these modules would work well when they were put together. Mostly, our testing focused around the interfaces that each module exposes, making sure that each module responds correctly when given input that simulates how another module would interact with it.

We created testbenches for every individual module which you can view in our project's source code. Below we've selected the testbenches we feel to be the most important and provided a short description along with the code (we've also removed some redundant code and replaced the deletion with ellipsis to make it easier to read).

feedback_tb.v

This module tests the feedback module's capability of creating correct response output given a guess and a code to check it against. Normally the code would be supplied by the PRNG, but for testing purposes the code will be 0, 1, 2, 3. Not that these numbers actually correspond to colors when given to the RGB LEDs. Our feedback module passed this test.

```
module feedback_tb;
    reg clk;
    reg last_turn;
    reg [2:0] code0;
    reg [2:0] code1;
    reg [2:0] code2;
    reg [2:0] code3;
    reg [2:0] history0;
    reg [2:0] history1;
    reg [2:0] history2;
    reg [2:0] history3;
    wire [1:0] ssd0;
    wire [1:0] ssd1;
    wire [1:0] ssd2;
    wire [1:0] ssd3;
    wire game_over;

    integer cnt = 0;

    initial begin
        clk = 0;
        last_turn = 0;
        code0 = 0;
        code1 = 1;
```

```
    code2 = 2;
    code3 = 3;
    history0 = 0;
    history1 = 0;
    history2 = 3;
    history3 = 0;

    #10 $finish;
end

always begin
    #0.5 clk = ~clk;
end

always @(posedge clk) begin

    $display("---------\n",
             "SSD: %d-%d-%d-%d\n", ssd0, ssd1, ssd2, ssd3,
             "Cnt: %0d", cnt);

    // Set history1 to correct
    if (cnt == 0)
        history1 = 1;

    // Set history2 to correct
    if (cnt == 1)
        history2 = 2;

    // Set history3 to correct
    if (cnt == 2)
        history3 = 3;

    cnt = cnt + 1;

end

feedback response(
    .clk(clk),
    .last_turn(last_turn),
    .code0(code0),
    .code1(code1),
    .code2(code2),
    .code3(code3),
    .history0(history0),
    .history1(history1),
    .history2(history2),
```

```
        .history3(history3),
        .ssd0(ssd0),
        .ssd1(ssd1),
        .ssd2(ssd2),
        .ssd3(ssd3),
        .game_over(game_over)
    );
endmodule
```

guess_tb.v

This testbench is meant to prove the abilities of our guess module. We test basic functionality such as moving the current selection left and right, changing the color of a selection by pressing up and down (the testing of enable/disable was done earlier, before in a previous version of this testbench). On the screen we would see the module's output, so we knew that the module was behaving correctly. We checked to see that - The module correctly chose which LED to blink - The module handled left/right correctly - The module allowed us to select colors in a cyclic fashion, repeating the same sequence of colors without skipping any. - If we scrolled through all possible colors, the selection would loop back on itself, repeating the same cycle of colors. The guess module passed this testbench.

```
module guess_tb;
    reg enable;
    reg left;
    reg right;
    reg up;
    reg down;
    wire [2:0] led_zero;
    wire [2:0] led_one;
    wire [2:0] led_two;
    wire [2:0] led_three;
    wire [1:0] blink_led;

    initial begin
        enable  = 1;
        left    = 0;
        right   = 0;
        up      = 0;
        down    = 0;

        $monitor("----\n",
                 "LEDs: %d-%d-%d-%d\n", led_zero, led_one, led_two, led_three,
                 "Sel_led: %d, time %d\n", blink_led, $time);

        #1 right = 1;
        #1 right = 0;
        #1 up = 1;
        #1 up = 0;
        #1 right = 1;
        #1 right = 0;
        #1 right = 1;
        #1 right = 0;
        #1 up = 1;
```

```
        #1 up = 0;
        #1 up = 1;
        #1 up = 0;
        #1 up = 1;
        #1 up = 0;
        ... // we kept going to test the "max out all colors" edge case
        #1 $finish;
    end

    guess guess_test(
        ...
    );

endmodule
```

history_tb.v

This module proves the capability of our history module, arguably the most complex module in our design. It was important that this testbench be thorough and that our history module function correctly. This test exercises the history module's ability to - Record a selection - Store a selection - Store multiple selections - Enable and disable itself - Switch into history mode and review all of our previous selections - Display the correct currently selected/currently guessing turn

All of the outputs of this module were displayed on screen in a well formatted fashion, so we could confirm that the history module was behaving as designed. The history module passed this testbench.

```
module history_tb;
    reg clk;
    reg mode;
    reg reset;
    reg up;
    reg down;
    reg select;
    reg [2:0] guess[3:0];
    wire [2:0] selection[3:0];
    wire [2:0] selected_turn;
    wire last_turn;

    integer i;
    integer cnt = 0;

    initial begin
        clk    = 0;
        mode   = 0;
        reset  = 0;
        up     = 0;
        down   = 0;
        select = 0;
        for (i = 0; i < 4; i = i+1)
            guess[i] = 'b000;
        #40 $finish;
    end

    always begin
        #0.5 clk = ~clk;
    end

    always @(posedge clk) begin
```

```
if (cnt == 0) begin
    $display("making selection");
    // start in guess mode
    guess[0] <= 'b001;
    guess[1] <= 'b000;
    guess[2] <= 'b000;
    guess[3] <= 'b000;
end

if (cnt == 1) begin
    $display("confirming selection");
    select <= 1;
end

if (cnt == 2) begin
    $display("checking selection");
    select <= 0;
    guess[0] <= 'b000;
    guess[1] <= 'b000;
    guess[2] <= 'b000;
    guess[3] <= 'b000;
end

if (cnt == 6) begin
    $display("Another selection");
    guess[0] <= 'b000;
    guess[1] <= 'b001;
    guess[2] <= 'b000;
    guess[3] <= 'b000;
    select <= 1;
end

if (cnt == 7)
    select <= 0;

if (cnt == 12) begin
    $display("History");
    mode <= 1;
end

if (cnt == 16) begin
    $display("Press down");
    down <= 1;
end

cnt = cnt + 1;
```

```
    end

    always @(negedge clk) begin
        $display("---------\n",
                 "Selection: %d-%d-%d-%d\n", selection[0], selection[1], selection[2], selec
                 "Selected Turn: %d\n", selected_turn,
                 "End Game: %d  Cnt: %0d", last_turn, cnt - 1);
    end

    history history_test(
        ...
    );

endmodule
```

led_driver_tb.v

This testbench exercised the LED driver's ability to switch between guess and history mode, and blink the correct LED when in guess mode (but not in history mode). All output was printed to the screen, and this module passed this testbench when we ran it.

```
module led_driver_tb;
    reg clk;
    reg blink_enable;
    reg [1:0] blink_led;
    reg [2:0] guess_rgb0;
    reg [2:0] guess_rgb1;
    reg [2:0] guess_rgb2;
    reg [2:0] guess_rgb3;
    reg [2:0] history_rgb0;
    reg [2:0] history_rgb1;
    reg [2:0] history_rgb2;
    reg [2:0] history_rgb3;
    wire [2:0] rgb0_out;
    wire [2:0] rgb1_out;
    wire [2:0] rgb2_out;
    wire [2:0] rgb3_out;

    initial begin
        clk     = 0;
        blink_enable = 0;
        blink_led = 1;
        guess_rgb0 = 1;
        guess_rgb1 = 1;
        guess_rgb2 = 1;
        guess_rgb3 = 1;
        history_rgb0 = 2;
        history_rgb1 = 2;
        history_rgb2 = 2;
        history_rgb3 = 2;

        $monitor("----\n",
                 "RGBs: %b-%b-%b-%b\n", rgb0_out, rgb1_out, rgb2_out, rgb3_out,
                 "Blink_enable: %d   Blink_led: %d  Time: %0d", blink_enable, blink_led, $ti

        #1 blink_enable = 1;
        #4 blink_led = blink_led + 1;
        #4 blink_led = blink_led + 1;
        #1 $finish;
    end
```

```
    always begin
        #0.5 clk = ~clk;
    end

    led_driver led_dr(
        ...
    );

endmodule
```

prng_tb.v

This is a very simple testbench meant to demonstrate the abilities of the linear feedback shift register that the PRNG module is based on. The PRNG would essentially start with a seed that we hardcoded in, and while this would mean that the code would be the same every time we reset the FPGA, it was good enough for our purposes. This module simply clocks the PRNG's "regenerate" input (labeled 'clk') and asks it to generate a bunch of pseudo random numbers, which we output to the screen. This testbench showed that the PRNG module worked well.

```
module prng_tb;
    reg clk;
    wire [2:0] code0;
    wire [2:0] code1;
    wire [2:0] code2;
    wire [2:0] code3;

    initial begin
        clk = 0;
        #10 $finish;
    end

    always begin
        #1 clk = ~clk;
        $display("code: %b %b %b %b\n", code0, code1, code2, code3);
    end

    prng prng_uut(
        .clk(clk),
        .code0(code0),
        .code1(code1),
        .code2(code2),
        .code3(code3)
    );
endmodule
```

Conclusion

We broke this project down into as many clean and independent submodules as possible, so as to keep our design simple and manageable. Unfortunately, some of our modules came with a high level of "interlock", where modules depended on each other quite a bit and the interfaces they presented were not as separated as we would've liked. In particular, the history module became the focus of our design, as every input essentially went into and came out of the history module. If we had the opportunity to do this project again, we might consider taking some time to further separate out the functionality of the history module. Perhaps turns will be counted separately, another module will be responsible for storing guesses, and another module can be responsible for accessing those guesses and displaying them.

The major features that are in our project are all implemented as separate modules. First, the button inputs to our FPGA are debounced by being sent through our debouncer module. The left/right/up/down inputs are then sent to our guess and history modules, which implement the guess and history modes. Guess implements the "guess editor" and history keeps track of all guesses that have been made. The PRNG module is simply responsible for generating and storing the winning code, and the feedback module compares some guess that you supply it (could be the current guess or a previous guess from history) and provides feedback according to the game rules (with Xs and Os). On the display side of things, the led_driver is responsible for powering the color LEDs, ssd_converter allows us to output to the four seven segment displays, and turn will power the LEDs above the switches.

We felt that this design made the most sense because each feature could be easily mapped to a particular module, but we didn't suspect that the history module would play such a major role in our design. The history module, in essence, became the "manager" module of our design, it would essentially drive the flow of our game by hijacking control of other modules when it needed to display history. The interaction between the history module and the guess module was also not so clearly laid out – the guess module would properly turn off when it wasn't in guess mode, but the history module never switched off. It still had to listen for input even in guess mode, because when the user presses the submit button it has to store it. This made our design more monolithic than we would have preferred.

This project was difficult because of the large amount of interlocking parts. Module interactions were essentially what drove this project, and that's what we struggled with the most. We did our best to test every one of our modules individually to make sure that each one would operate correctly when given input from other modules, but unfortunately we found that it's much harder to predict in which other ways modules would interact with each other. We tested extensively to avoid these interactions, but our slightly monolithic design could be what kept us from ironing out all of the bugs. We also think that the

interactions could be the result of test benches not properly aligning with the synthesized version of the code, but we're unsure of this.

Ultimately, we did our best to iron out as many bugs as possible, and we ended up with a project that demoed the majority of the purported features.