6.111 Final Report DDR Whack-A-Mole

Ara Adhikari, Victoria Ouyang, Davis Tran

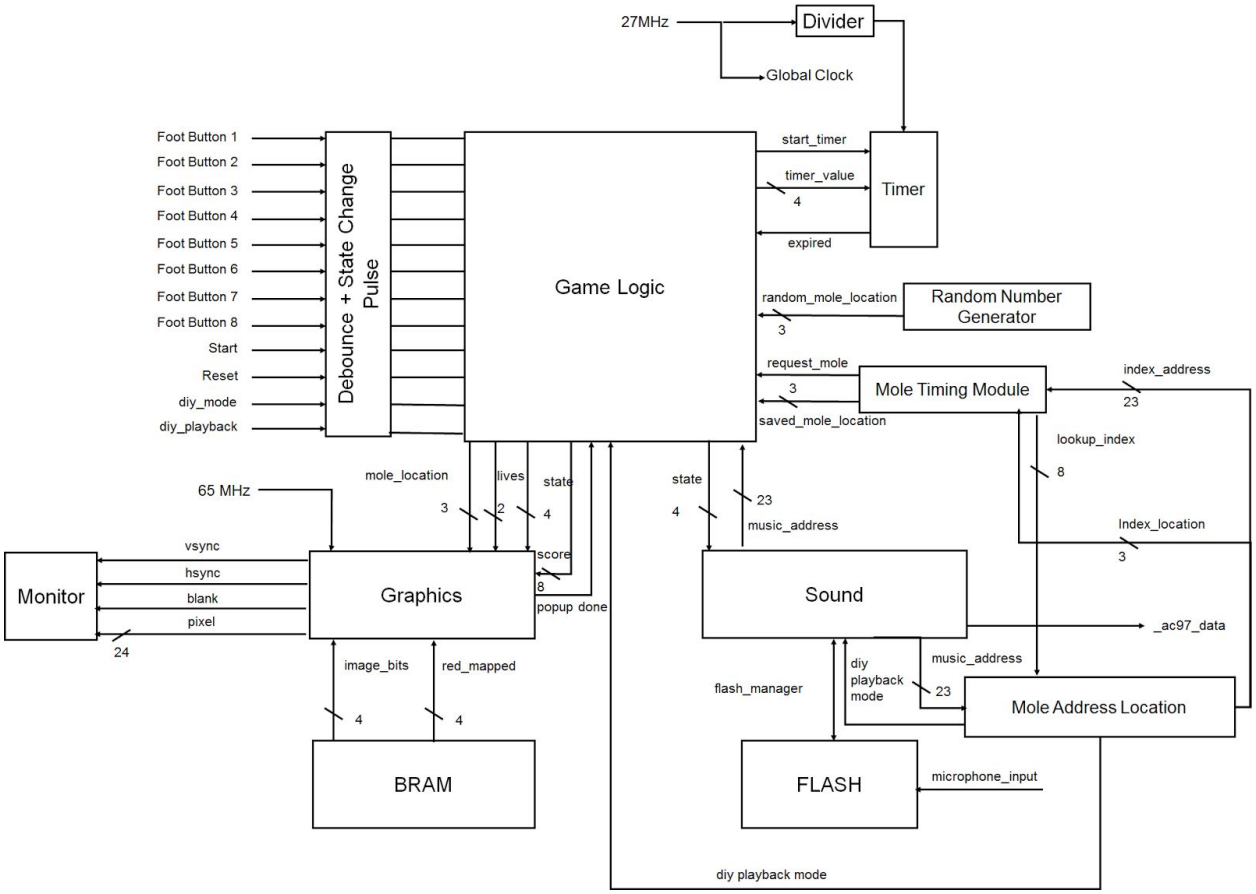
Table of Contents

6.111 Final Report DDR Whack-A-Mole

1.0 Project Overview

For our final project, we built a DDR-style Whack-A-Mole game. We setup a 3 x 3 square "board" with 8 foot buttons (the center square is for the player to stand on) that each correspond to a location on the screen where a mole could potentially pop up. The location of the mole is generated randomly. The goal of the game is for the player to step on the corresponding button when a mole pops up in a certain allotted amount of time. Each time the player successfully whacks a mole, the scoring mechanism will increase. As the score increases, the difficulty of the game also increases because the amount of time the player has to step on the corresponding button decreases. If a mole is missed, the player will lose a life, and when all lives are lost, the game is over. There is also a make-your-own-game mode where the player can press buttons to pre-program mole pop-up locations as the background music plays through once, and play the game based on their button presses instead of the normal gameplay. Overall, there are five big aspects to this project: foot buttons, game logic, graphics, sound, and make-your-own-game mode.

2.0 Block Diagram *Davis*



3.0 Project Implementation Details and Challenges

3.1 Foot Buttons *Davis*

Considering the game was a DDR game, the original intention was to build a full-scale DDR board. The main reason for using a team-made board rather than the original DDR Pad was that the drivers for the original board were not readily available. Instead, the "board" was made out of a set of 8 buttons arranged in a square. The buttons were unintuitively push-on, push-off but this was accounted for in our final design and discussed below. The buttons were selected to be rugged to endure the test of time (a month) as we designed the game. One extra button was dissected to determine proper usage and another was broken during play-testing.

3.1.1 Decoding Push-On Push-Off Button

Buttons included on the LabKit were simple buttons were HIGH when unpressed and went LOW when pressed. However, the buttons selected would switch states and remain in the a new state when pressed. To account for this we created a module that output a pulse whenever a one bit signal changed states. The module included a delay similar to debounce modules to account for rapid and unintended state changes, and the buttons were all wired to this module.

This properly integrated all the buttons and only resulted in one minor issue. When the FPGA turns on, the inputs are all assumed or reset to be low. Thus if any of the buttons were in the HIGH state, the state change module would send a pulse immediately after the FPGA turned on. While this generally did not cause problems, the game was designed to start whenever the *up* button was pressed. Thus the game would sometimes automatically start when we turned on the FPGA. A somewhat hacky solution was to simply press the button so that it changed to LOW state and then restart the FPGA and reload the program. Notably, this was not an issue after the FPGA was already running and a player wanted to play a new game after an old one ended.

The 8 buttons were wired to a protoboard provided by the lab and held at LOW with pull-down resistors. The signal was wired to the lab kit at *user[31:24]*.

3.2 Game Logic *Davis*

The structure of the game was centered around a large game state module. The main module as well as each supporting module is outlined in detail below. Each module receives a 27 MHz clock as input (typically *clk*) and a *reset* signal.

3.2.1 Game State FSM

The game state module determines state transitions based on inputs and manages player lives and player score. Table 1 illustrates all the states built into the FSM and a description of events that occurred in each state. Issues and clarifications for particular states are outlined in later sections.

State Name	State Function
IDLE	Player can press *up* to move to begin the game or flip *switch[7]* to begin recording a game. The start screen is displayed.
GAME_START_DELAY	Two second delay to allow player to prepare for the game to begin and moles to appear.
GAME_ONGOING	Main state for gameplay. This state constantly checks lives. If lives has reached zero, transition to GAME_OVER state.
REQUEST_MOLE	This state is entered from GAME_ONGOING when the mole management module requests a mole. This state is used to send a pulse to other modules to indicate a mole is about to be summoned.
MOLE_ASCENDING	A normal mole image ascends in one of eight prescribed locations.
MOLE_COUNTDOWN	The mole remains on the screen for a prescribed amount of time or until a button is pressed. Additionally a mole sound effect is played.
MOLE_WHACKED	A state used as a pulse to signal that a mole been correctly stepped on. Score is incremented by 1.
MOLE_MISSED	A mole was not whacked in time or the correct space was stepped on. Lives is decremented by 1.
MOLE_WHACKED_SOUND	Time for the whacked sound effect to play. Additionally is a safe period where the player cannot accidentally hit another button.
MOLE_MISSED_SOUND	Time for the missed sound effect to play. Is a safe period similar to above state.
DEAD_MOLE_DESCENDING	After a mole is whacked and the sound is played, the mole descends back down into nothingness.
HAPPY_MOLE_DESCENDING	After a mole is missed and the sound is played, the mole descends back down into nothingness.
GAME_OVER	Lives has reached zero and the game is over. The user may press up to move to the start screen.

RECORD_MODE_IN_PROGRESS	Control transfers to sound modules to manage recording a full game.
DIY_DONE_RECORD	Moves to playing the newly recorded game when player input is received.

Table 1: A description of the function of each state.

Note that a single timer instance is used for all states except for MOLE_COUNTDOWN. Whenever a state change occurs, a *start_timer* signal is sent to the timer but the *expired* signal is only used in one state.

For due diligence, this next paragraph provides a summary of all the inputs and outputs, which are elaborated upon in the next sections where necessary. The *start* signal is used to determine when to start a game or when to start recording. The *whacked* and *misstep* signals are used to determine the state to move to after the MOLE_COUNTDOWN phase. The *request_mole* signal indicates a transition should occur from GAME_ONGOING to the MOLE_ASCENDING state. The *expired* signal determines when to transition out of the GAME_START_DELAY. Previously, the *expired* signal was a consistent two-second timer that also determined when to transition out of the MOLE_COUNTDOWN state, but this is now controlled by the *variable_expire* signal to add a changing difficulty level feature to the game. The *diy_mode*, *diy_playback*, and *ready_to_use* signals are interacting with the game recording feature. The *popup_done* signal is used to transition from ascending and descending mole states. Additionally, *[2:0] random_mole_location* and *[2:0] saved_mole_location* dictate where the mole pops up on the screen whenever *request_mole* is asserted. Finally, *[3:0] display_state*, *[2:0] mole_location*, *[1:0] lives*, and *[7:0] score* are output from the game state module to the sound and graphics project components.

3.2.2 Dividers and Timer

The divider module asserts a *one_hz_enable* signal HIGH (the same as the one in the car alarm lab) once per second. This allows the timer value to be readable on the led display. The timer asserts *expired* as high for one clock cycle each time its internal counter reaches zero. The internal counter decrements by one each time it receives the *one_hz_enable* pulse.

An additional 10 Hz divider was created to run a timer which decremented a tenth of a second each time the *ten_hz_enable* signal was pulsed. This divider was created to implement difficulty levels in the game. When a mole pops up, it stays on screen for a limited amount of time before disappearing. For increased difficulty, the amount of screen time is decreased by a 0.2 seconds (down to some minimum) every time the score hits a multiple of four, which was not possible with the simple *one_hz_enable* signal.

3.2.3 Random Number Generator

In the normal game mode, the mole pop-up location is randomly generated. Random number generation is handled by a four-bit linear-feedback shift register (LFSR). Since there were eight locations, only 8 numbers (0-7) were required but because 0 is not a possible state for a three-bit LFSR, the last three bits were used in a four-bit LFSR. This did create a non-uniform probability distribution for the locations but was largely unnoticeable during gameplay.

3.2.4 Button Interpretation

Whenever a mole arose, button input was received from the player. When a location was generated as a number 0-7, it was converted into a one-hot representation of each location. The buttons were also concatenated into a one-hot representation of those inputs. The two inputs were compared to output a *whacked* signal when the player signal matched the mole location, and a *misstep* signal when the player missteps.

3.2.5 Mole Timing

A critical feature of the game was synchronizations of moles to music. Background music was played from FLASH memory and the current read address, *[22:0] music_address*, was constantly sent to the mole timing module. The module would compare the address to a prescribed set of addresses for moles to pop up and each time there was a match, the *request_mole* signal was asserted high for one clock cycle.

3.3 Graphics *Victoria*

The graphics portion of the project involved three different mole images representing specific states of the game and various text screens that displayed start, lives, score, and game over. A normal mole image was displayed when moles popped up on the screen. Once the mole popped up, if the mole is successfully whacked, then the image turns into a dead mole image and shrinks back down. Alternatively, if the mole is missed (either by stepping on the wrong button or not stepping on the correct button in time), the image turns into a happy mole image and then descends.

3.3.1 Pre-processing Images

The images for the moles were pre-processed using the MATLAB script provided in the tools section on the course website. The default script takes in a BMP image file with 256 colors, and outputs COE files for the image and 8-bit RGB values. However, since the mole images used did not need 256 colors, the images were pre-processed using Gimp, a photo editor program, to downsize the file to 16 colors. The MATLAB script needed to be modified to output 4-bit RGB values (comments in the script indicate where to replace 8-bits with 4-bits).

3.3.2 Generating Image Block Memory Modules

The COE files were loaded into BRAM of the 6.111 Labkit following the instructions in the Displaying Images PDF in the tools section of the course website. For each image, generate block ROMs for each of the COE files, and call the modules inside a module to display the image. The XVGA module was adapted from Lab 3 (pong game), and appropriate hcount and vcount values are generated from that module. The values in the COE file for the overall image represent image addresses that map to specific colors in the RGB COE files. The image address is calculated according to (hcount - x) + (vcount - y) * WIDTH in pixels of the image, and passed into the image ROM. The *image_bits* output of this ROM is then inputted to the RGB ROMs to get the bits values for *red_mapped*, *green_mapped*, and *blue_mapped*. Since *pixel* is a 24-bit color value, and the image colors are only 4-bit numbers, the 4-bit values became the most significant bits of each field (*red_mapped*, *green_mapped*, and *blue_mapped*), and 0's were added in the least significant bits of each field. Check to make sure that when instantiating the ROMs, the fields are in the correct order/or include the .*field_name*(wire) (i.e. .*image_bits*(image_bits)) for all the fields to avoid issues. This was a significant roadblock in getting images to be properly displayed in our project.

Once modules were created for each image, the graphics code was integrated with the game logic module to display the moles whenever the game indicated a new mole popup at a given location. Locations were calculated based off of the image sizes and screen size (1024 x 756 pixels) to split the screen into 9 sections evenly. Moles only pop up in 8 sections, and the center section was used to display the number of lives remaining and the player's current score. Different states of the game determined which of the three image modules to display at a given time.

3.3.3 Smooth Image Transitions

After verifying that images were appearing, changing, and disappearing properly, smooth mole transitions were implemented. Rather than having the moles simply appear and disappear on the screen, the moles are able to ascend from the bottom of the specific screen section (imagine an actual mole coming up from the ground), and then descend back down after it has been displayed for a moment.

To do so, an additional input was added to each of the image modules. Instead of only having a y_value input, there is a *y* that changes and *y_permanent* value that represents the non-changing border of the image. Since the 27 MHz clock used in the module is too fast to see noticeable transitions when updating *y*, a new divider module mole_divider was created to generate the clock that changes *y* by 1 on every pulse of the divider output, *mole_popup_clock*. This pulse happens less frequently, but quickly enough that the transition looks smooth. Note that only the normal mole image ascends, and only the happy mole image and the dead mole image descend.

To make the mole ascend, *y* (referred to as *y_change* for clarity) is initialized to 255 + *y_permanent*. The height of each image is 256 pixels (which divides the screen evenly into three

rows), so the bottom border of the image would be located at 256 + *y_permanent* (the anchor location of the image). While *y_change* is within the bounds of *y_permanent* and 256 + *y_permanent*, decrease *y_change* by 1 each time there is a pulse from the divider described above. Once *y_change* equals *y_permanent*, the popup is complete, so a *popup_done* signal is set to 1 and sent to the game logic module.

Descending mole images applies the opposite logic. When the mole descends, the *y_change* value should already be at *y_permanent* initially because it finished ascending first. Then, while *y_change* is within the bounds, on every pulse from the divider, increase *y_change* by 1 until it reaches 256 + *y_permanent* and again set *popup_done* signal to 1.

3.3.4 Displaying Text

The text graphics in this game are displayed by assigning pixel colors (either white or black) using case statements. The code was adapted from Weston's example Mario puck. Pixel words were designed and hardcoded in the case statements. Each of the ten digits are also implemented as their own modules. At the beginning of the game, the start screen displays "Whackamole" and the instruction "Press up to start" to instruct the player to begin. While the game is ongoing, "Lives" and "Score" is displayed at the center of the screen. The number of lives remaining and score is taken in as an input from the game logic module, and depending on the number, the correct digits are displayed. Once the number of lives becomes 0, the ending screen displays "Game Over" and the total score.

3.4 Sound *Ara*

For our project, we decided to incorporate background music plus three different sound effects. A pop-up sound effect for when a mole pops up, a "yay" sound effect for when player hits the mole, and "oh no" for when a player misses the mole. We decided to use FLASH memory to store our game audio data. This way, we had enough space to store a full length song, along with the benefit of not having to reload music and sound effects into the 6.111 Labkit everytime we made changes to our project.

3.4.1 Recording Audio to Memory

To get the audio data to store in FLASH memory, we used the ac97 (similar process to the audio lab, except we are storing to FLASH memory instead of BRAM or ZBT memory). FLASH memory can be tedious to work with, so before writing anything to it, erasing the entire FLASH memory is the easiest way to go. After FLASH memory has been erased, everything stored in it are 1's, and FLASH memory is ready to write to.

To handle recording music to FLASH memory, we made a separate project file from the one for the rest of our game/project. This project includes the Verilog code provided under the 6.111 Tools section for working with FLASH memory, making an instance of the flash_manager and controlling the input signals to the flash_manager module accordingly. Two notable changes

were made to the provided code under the tools section: starting flash_manager.v in state = 2, and using MAX_ADDRESS parameter to determine the limit on how high the FLASH address go to. Inconsistency in use of MAX_ADDRESS in the provided code resulted in a lot of debugging when determining where in FLASH memory data was being written to.

Here is an overview of how to interact with the FLASH memory using the flash_manager module. Setting reset signal to HIGH means FLASH will erase. To write to FLASH memory, *writemode* should be HIGH, *doread* should be LOW, and everytime *dowrite* is HIGH, the data in 16-bit *flash_data* input is written to FLASH memory. Using the flash_manager as it is, we do not have control over which address to write to; the address starts at 0 and increments sequentially as more writes happen. For reading from FLASH memory, that is not the case. There is a 23-bit *raddr* input through which we can select where in FLASH memory to read from. For reading, *writemode* should be LOW and *doread* should be HIGH. Read data will be available in the 16-bit *frdata* output. Note that *flash_address* and *faddr* are not the same. You should use the address that goes to *faddr* for most (if not all cases), as to access the data in the first address location of FLASH memory, you set *faddr* to zero, not *flash_address*. The *flash_address* should directly connect to the *flash_address* in the labkit module.

There are a lot of signals to set when working with FLASH memory. It helped to read Diana and Lorenzo's Fall 2016 project report to get familiar with FLASH memory. As for the logic to assign flash_manager signals to get a specific action to execute, we followed a similar logic structure to one Lorenzo wrote for their project, except we were writing to FLASH memory when it's *busy* signal was LOW and ac97 was *ready*, instead of writing data based on a FIFO (first-in-first-out) queue. The data incoming through *from_ac97_data* was also filtered through the fir module (same one from the audio lab). When every ready 8-bit *from_ac97_data* value was written to FLASH memory, the playback was slower than normal. To fix this problem, every 8th data coming from *from_ac97_data* was written to FLASH memory instead.

3.4.2 Playback of Audio from Memory

As for the playback itself, we set FLASH memory to read mode, incremented the *fraddr,* and sent the lower 8-bits of *frdata* through the *to_ac97_data* every time it was ready to receive data. However, even with the use of the FIR filter, the music was not clear. There was some noise, and human voice in the music sounded slightly distorted (off pitch) as well. We tried different sample rates for recording, but we could not get any better results than using every 8th sample. We thought it might have been a problem with the microphone, so we tried to record through auxiliary audio cable instead. With stereo sound, all we got was noise. But when we changed the sound being played from stereo to mono, we were able to get slightly better recording than with the microphone, but the issues mentioned above still remained. We did not really know how else to fix the audio anymore, and because it was functional for the purpose of our game, we decided it was good enough. If we were to redo the project, we would definitely look into changing the structure of reading and writing to flash so that it does not depend on the

busy signal, by using some sort of FIFO. This would probably help improve the sound quality, because we cannot always rely on the timing of the *busy* signal (this was mentioned in one of the previous projects using FLASH memory).

3.4.3 Sound Module Details

In the actual game project file, the sound_module handles all of the playback of the audio. To have the sound effects play together with the background music (overlapping), this module first loads all the sound effects into BRAM before it starts playing any music. Once BRAM is loaded, based on the *game_state* input and *diy_state* (on most states), the background music plays in repeat. Every time the ac97 is *ready*, the filter output (which is filtered FLASH memory data) is sent out to *to_ac97_data* and address is incremented (until it has reached the end of background music, in which case it start over from the beginning). As for the sound effects, if the game_state is in MOLE_COUNTDOWN , MOLE_WHACKED_SOUND, or MOLE_MISSED_SOUND, a corresponding sound effect will be played (added together with the background filtered output to be send to *to_ac97_data*). The sound effects are not sent through a FIR filter because we couldn't hear a big difference in filtered sound effect vs. not filtered sound effect when it was played together with the filtered background music.

3.4.4 FLASH Memory Difficulties

There are a lot of input and output signals when working with the flash_manager module, so it is very important to go back and check to make sure you have all the input and output connections, and they are spelled correctly. We had a lot of issues in first getting an erase/write/read from FLASH memory because we forgot to send the *flash_data* input to the flash_manager. We thought a missing connection would result in an error while the project compiled, but that was not the case, and the project would compile without any problems. Because of this, we thought the problem was elsewhere in our code, and it took us a very long time to debug. While working with FLASH memory, we recommend using the 6.111 Labkit display and/or the 8 LEDs so you can check if things are working as expected. It also helps with debugging.

3.5 Make Your Own Game Mode *Ara + Davis*

In addition to the normal game play, we also added a mode to our game where players can choose when and where the moles pop up in the music instead of randomly generating mole locations. Most of this mode's functionalities is handled by the mole_address_locations module. When this mode starts, the background music plays once. While the music is playing, every button press (every 2 seconds because that's the minimum time we decided on between mole pop-ups) is recorded. When a button is pressed, the *music_address* the FLASH memory is reading from (coming from the sound_module, this is the *raddr* not *flash_address*) along with the button number (each button has a number associated with it) is stored in arrays. There are

two arrays: one for the button location and one for the *music_address* of when the button is pressed. Once the song has reached the end, or the array has reached capacity (whichever comes first, but mostly likely end of the song because capacity of the array is very high), the module waits for the *diy_playback* signal to go HIGH. If *diy_playback* is HIGH, then it sets *diy_playback_mode* signal to HIGH. The *diy_playback_mode* is what the game_logic and sound_module uses to determine if a game is ready to play in the player customized mode instead of normal game. If so then the audio will play from the sound_module like a normal game module. In the mole timing module, this is managed by keeping a running counter of how many moles have been shown. This is sent to the sound modules as a *[3:0] lookup_index* with appropriate delays and the corresponding *[22:0] index_address* and *[2:0] index_location* is received. Thus in *diy_playback_mode* the *music_address* is matched with addresses stored during the game recording session rather than from the pre-written collection of mole addresses.

4.0 Lessons Learned

1. We spent a lot of time debugging to get FLASH memory working in the beginning. We thought it was an issue with writing to FLASH memory because the read data displayed were all 16‘hFFFF (meaning all 1’s). So, we thought the only issue was with writing to it. However, we later realized the issue behind why FLASH was not working was because the *flash_data* from labkit module was not wired to flash_manager module. So, it is super important to double check to make sure all signals to your modules are wired. If there is an issue with a module not working, always go back and check the module instantiation.
2. The led display on the labkit is not always reliable. If the display does not work, half the time, restarting labkit and reprogramming fixes the issue. Otherwise, it might be because the number of bits being displayed might be off. There were also cases where displaying values inside a module worked, but when we tried to see the same module’s output signals from the labkit module, where it was instantiated, the values never changed. We thought it was something wrong with the instantiation, but it was just the display not working as expected.
3. When the led display does not seem to be working, the 8 led lights are a great way to debug, especially when you are trying to check values that are 8 bits or less. When using this as a debugging tool, just note that the leds turn on when LOW.
4. When working with FLASH memory, note that *flash_address* is not the same as *raddr*. When working with the mole_address_locations, we sent in *flash_address* to use as addresses instead of the *raddr.* This resulted in the music and game in Make-Your-Own-Game mode being cut off halfway through the background music. Using *raddr* instead of *flash_address* in the mole_address_locations module fixed this issue.

5. Displaying graphics was difficult at the beginning of the project. It is important to understand the MATLAB script, and what information the COE files provide. When things were not working, it was helpful to actually look at the values in the COE files to ensure that it makes sense and generated correctly. As mentioned above in the report, always check to make sure the ROM modules are instantiated correctly and that the signals are in the correct order.
6. Hardware can run into issues when resetting, and often times a software reset is not enough. We were puzzled with the graphics display when sometimes running the exact same code would result in oddly colored images displayed (i.e. patchy colors, skewed graphics). Turning the labkit off and on, and then reprogramming generally solved this issue.
7. The text display graphics was implemented at the end and quite tedious. Originally we wanted to just display images of the text similar to how mole graphics were being displayed. However, we ran out of BRAM space on the labkit. One alternative solution to long case statements assigning pixel colors was to shrink the coe files to use only two colors (since we only need black and white). However, due to time constraints and the fact that we were already halfway done implementing the case statements, we decided to continue with that method even though it was tedious.
8. Since the image modules were implemented using a 65 MHz clock and the rest of the game ran on a 27 MHz clock, we experienced weird behavior in which moles were not displaying in the correct locations when requested. Making all of the signals being passed between graphics and game logic modules registers and having graphics take in two clock inputs fixed this problem.
9. This project is based on knowledge of digital systems. One component of the project that was slightly stressful was physically constructing the board. Since this was a mechanical aspect rather than an electrical aspect, we found a way to make the game work without the full wooden board. The final arrangement of the buttons was enough to play the game.
10. In situations where we dealt with numerous signals, being able to quickly create test fixtures was extremely helpful. Many times when signals were not performing correctly, a test fixture helped easily identify what exactly was wrong. Additionally, a whole section of the code was dedicated to creating outputs for the led display to facilitate debugging.
11. It is very important to double check that all signals are accounted for whenever a module is instantiated. There was once a bug that was resolved adding a missing wire as an input to an instance of the game state module.
12. Staying organized early on and over-commenting was extremely helpful for finding and understanding how bits of code worked later on.